

# Scripting avancé avec Windows PowerShell

**Une référence pour  
l'administrateur et le développeur**

Automatisation • XML • Services et logs • Sécurité  
• COM • WMI • CIM • Pipeline • Expressions régulières  
• Modules • Remoting • Workflow • Active Directory • .NET

**Kais Ayari**



# Scripting avancé avec Windows PowerShell



La complexité des infrastructures Microsoft (Windows Server, Active Directory, Exchange Server, SCCM, SCOM, etc.) rend nécessaire de disposer d'un ensemble d'outils cohérents pour aider les administrateurs et les ingénieurs en charge d'automatiser les processus en œuvre dans le système d'information. C'est chose faite grâce à PowerShell dont l'arsenal, très vaste, comprend à la fois un langage de script et un environnement en ligne de commande, et dépasse largement les fonctionnalités proposées par VBScript.

## Une référence pour l'administrateur en environnement Microsoft

À jour de la dernière version de PowerShell, cet ouvrage avancé met en pratique le langage dans de très nombreux cas concrets de tâches d'administration.

Après avoir exposé les connaissances fondamentales à l'utilisation de PowerShell (environnement en ligne de commande, environnement de scripting, syntaxe, principe du pipeline, format des données, manipulation des objets, sécurité), puis chaque élément du langage (variables, opérateurs, tableaux et dictionnaires, boucles, contrôle de flux, fonctions, modules et gestion d'erreur), cet ouvrage met en pratique ces connaissances théoriques avec WMI, les services, les logs, la base de registres, COM et XML. Le lecteur verra ensuite comment administrer des machines à distance, utiliser les workflows, manipuler les expressions régulières, automatiser la gestion de l'Active Directory, mieux connaître le framework .NET, et développer des interfaces graphiques facilitant l'utilisation des scripts écrits.

## Au sommaire

**Les bases** • Commandes natives Windows • Comparer sur la base de propriétés d'objets • Filtrer les objets • Effectuer des opérations sur chaque objet • Créer des objets • Variables automatiques • Opérateurs de comparaison • Opérateurs d'affectation • Opérateurs de redirection • Opérateurs de fractionnement et de jointure • Opérateurs spéciaux • **Tableaux et dictionnaires** • **Boucles** • for • foreach • do...while • do...until • while • **Instructions conditionnelles** • if...else • if...elseif...else • switch • **Fonctions** • Déclarer • Paramètres • \$args • Documenter • Syntaxe de l'aide • Description des mots-clés • **Modules** • Écrire un module en langage PowerShell • PSSnapins • **PowerShell et la gestion d'erreur** • Différencier l'erreur de l'exception • Anatomie d'une erreur • Paramètre -ErrorAction et la variable \$ErrorActionPreference • Instruction trap • Instruction try...catch...finally • **PowerShell en pratique** • **L'infrastructure WMI et CIM** • Les cmdlets WMI • WMI en pratique • Les cmdlets CIM • CIM en action • **Gérer les services** • Lister les services • Démarrer, arrêter, redémarrer et interrompre l'exécution d'un service • Configurer le mode de démarrage d'un service •Modifier le compte de connexion d'un service • **Gérer les logs** • Lister • Configurer et sauvegarder • Supprimer • **Gérer la base de registre** • Naviguer dans la base de registre • Chercher, ajouter et effacer un élément de registre • La base de registres à distance • **PowerShell et COM** • Manipuler un objet COM • Automatiser Internet Explorer • Mapper un lecteur réseau • Utiliser l'objet FileSystem • Créer un fichier texte • Lire un fichier texte • **PowerShell et XML** • Manipuler un fichier XML • Les cmdlets Export-Clixml et Import-Clixml • Rechercher des informations à l'aide de la cmdlet Select-Xml • **Configurer l'accès à distance** • Connexions temporaires et persistantes • Quel type de connexion privilégié? • Se déconnecter d'une session pour se reconnecter plus tard • **Workflows sous PowerShell** • Workflows comme synonyme de productivité • Relation PowerShell-WWF • Différence workflow/fonction • Paralléliser des opérations • Effectuer des opérations de manière séquentielle • Instruction InlineScript • **Expressions régulières** • Classe [Regex] • Opérateurs -match et -notmatch • cmdlet Select-String • **Automatiser l'Active Directory** • Administrer les utilisateurs et groupes de l'Active Directory • Créer un domaine Active Directory • Zones DNS de recherches directes et inversées • **Utiliser le framework .NET** • Ajouter un type • Envoyer un e-mail • Collecter des informations DNS • **Développer une interface graphique avec PowerShell** • Windows Forms ou Windows Presentation Foundation • PowerShell Studio 2012 • Créer une interface graphique • Écrire le code • **PowerShell sous Linux : rêve ou réalité ?** • Gérer les services • Gérer les modules kernel • Gérer les disques.

## À qui s'adresse cet ouvrage ?

- Aux ingénieurs et administrateurs systèmes et réseaux Microsoft souhaitant automatiser efficacement leurs opérations d'administration
- Aux programmeurs désirant automatiser le déploiement et la configuration de leurs applications
- Aux ingénieurs et administrateurs venant du monde Unix/Linux et évoluant dans des environnements hétérogènes

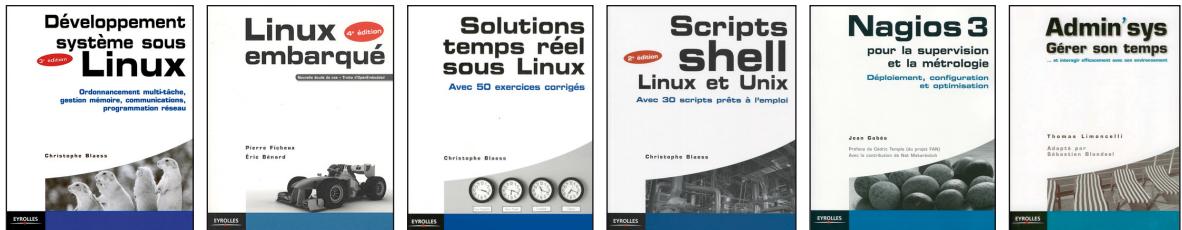
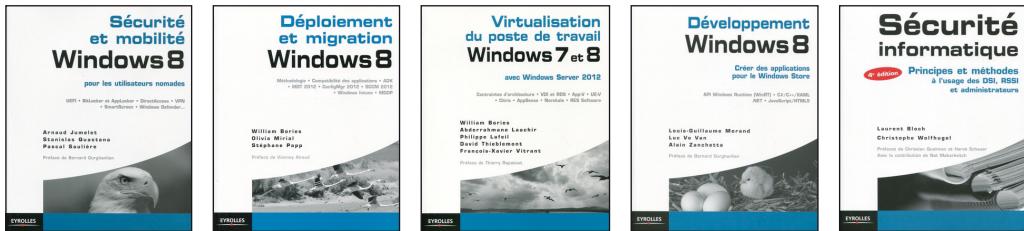
## K. Ayari

**Kais Ayari**, PowerShell Guru et Scripting Specialist, a notamment travaillé chez Microsoft en tant qu'expert et référent technique PowerShell. L'auteur a pensé et conçu l'architecture d'une version de PowerShell sous Unix/Linux, ce qui fait de lui l'un des meilleurs experts au monde sur cette technologie. Il collabore également avec les éditions Manning en apportant son expertise technique sur PowerShell.

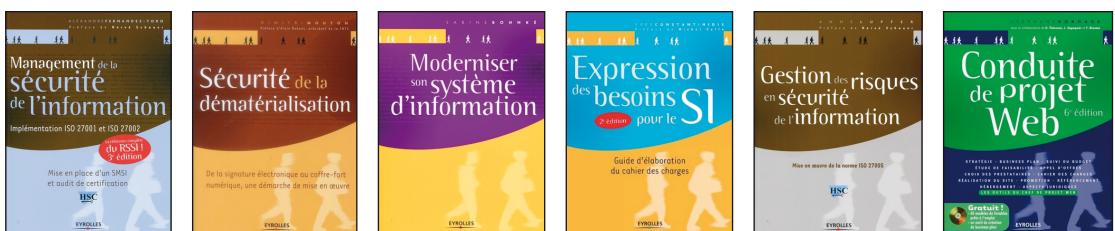
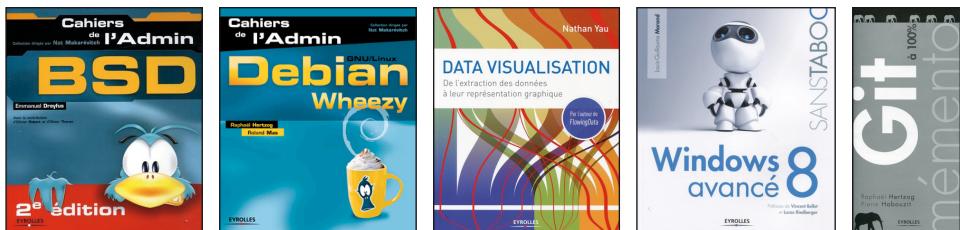
# **Scripting avancé avec Windows PowerShell**

**Une référence pour  
l'administrateur et le développeur**

## DANS LA MÊME COLLECTION



## CHEZ LE MÊME ÉDITEUR



Retrouvez aussi nos livres numériques sur  
<http://izibook.eyrolles.com>

# **Scripting avancé avec Windows PowerShell**

**Une référence pour  
l'administrateur et le développeur**

Automatisation • XML • Services et logs • Sécurité  
• COM • WMI • CIM • Pipeline • Expressions régulières  
• Modules • Remoting • Workflow • Active Directory • .NET

**Kais Ayari**

**EYROLLES**

ÉDITIONS EYROLLES  
61, bd Saint-Germain  
75240 Paris Cedex 05  
[www.editions-eyrolles.com](http://www.editions-eyrolles.com)

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage,  
sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie,  
20, rue des Grands Augustins, 75006 Paris.

© Groupe Eyrolles, 2014, ISBN : 978-2-212-13788-0

# Avant-propos

---

Depuis ses débuts, Windows n'a jamais véritablement eu d'interpréteur de commandes, mis à part le classique (et très rudimentaire) `cmd.exe` qui émule partiellement MS-DOS. En effet, Microsoft n'a jusqu'à présent jamais promu la gestion du système d'exploitation en ligne de commande, l'idée ayant toujours été de recourir à une interface graphique, pour que les outils soient utilisables à la souris. Le shell fournissant un nombre très réduit de commandes, les utilisateurs (administrateurs, ingénieurs...) ont privilégié les outils dotés d'interface graphique.

Cette politique a eu, on ne peut le nier, des conséquences dévastatrices sur le comportement des utilisateurs, dans la mesure où elle les a rendus à leur insu très réticents à la ligne de commande, sinon rebelles. Le hiatus avec le monde Unix/Linux s'est profondément accentué au fil des années ; il n'a jamais été dans la culture de l'administrateur Windows d'utiliser les langages de scripts (à quelques exceptions près), alors qu'au contraire, les informaticiens administrant des plates-formes Unix/Linux ont intégré ce réflexe depuis bien longtemps. C'est là une différence quasi culturelle.

Toutefois, Microsoft a pris conscience du problème depuis plusieurs années et a apporté avec PowerShell une réponse à la hauteur de ce qui est fait par la concurrence. Cette technologie d'automatisation, très riche, dépasse à mon sens toutes celles du même genre développées jusqu'à nos jours. Les administrateurs Windows peuvent désormais être fiers de l'outillage dont ils disposent : il leur est possible de choisir d'utiliser le shell ou l'interface graphique. Et on sait l'importance extrême, pour tout informaticien, d'avoir le choix.

## Automatiser l'administration de systèmes

Aujourd'hui, les ingénieurs et administrateurs système et réseau effectuent énormément de tâches quotidiennes, le plus souvent si répétitives qu'à force, ils finissent par travailler en perdant en précision. C'est pourquoi il est nécessaire qu'ils se dotent de

moyens pour développer des processus d'automatisation – l'objectif étant, autant que faire se peut, d'automatiser les tâches rébarbatives pour se concentrer sur d'autres volets de son travail tels que la veille technologique, la rédaction des documentations, le transfert de ses connaissances, et surtout, le traitement des urgences, qui surviennent hélas souvent lorsqu'on les attend le moins.

Évidemment, on ne peut qu'adhérer à cette belle théorie, mais dans les faits, le monde de l'administration système et réseau oppose une sorte de réticence à ce type de processus. La raison en est qu'automatiser des tâches d'administration complexes demande de connaître au moins un langage de script, si ce n'est plusieurs. Or, dans le monde de Microsoft, un immense fossé sépare l'administrateur du programmeur, en raison de la façon dont les systèmes d'exploitation Windows (et, en réalité, tous les produits Microsoft) sont faits.

Car administrer un système Windows se fait la plupart du temps, et malheureusement, avec la souris. Ainsi vous pouvez, par exemple, gérer vos objets dans l'Active Directory via une console MMC (*Microsoft Management Console*). Depuis des années, cet outil très important est l'élément le plus utilisé dans le monde de l'administration Windows car on peut y ajouter des composants logiciels enfichables, pour gérer quasi tout le système. Toute personne qui a déjà utilisé une console MMC sait qu'il n'est pas nécessaire de connaître tel ou tel type de langage pour agir efficacement ; la souris suffit. Au contraire, un système d'exploitation Linux favorise, de par sa structure, une administration en ligne de commande, poussant les administrateurs, même juniors, à se familiariser avec le shell, et donc à intégrer dans leur quotidien le réflexe de l'automatisation.

Avec PowerShell, Microsoft a véritablement changé la donne. En effet, cet outil, qui s'adresse en premier lieu aux administrateurs (mais pas seulement), a pour objectif d'ouvrir enfin des perspectives d'administration plus importantes que celles offertes en son temps par VBScript. Ce dernier n'est qu'un langage de script, créé en tant que tel, alors que PowerShell est bien plus que cela : c'est une technologie d'automatisation comprenant un langage de script *et* un environnement en ligne de commande. Il dispose d'un service web et est à la base de beaucoup de plates-formes serveurs comme Exchange 2007, 2010, et plus.

L'arsenal offert à l'administrateur par PowerShell est très vaste, et dépasse de loin les fonctions proposées par VBScript – quoique, il faut le souligner, ce dernier ait conservé toute son efficacité et reste largement utilisé aujourd'hui.

## À qui s'adresse ce livre ?

Ce livre s'adresse à des professionnels recherchant l'efficacité dans leur travail quotidien. Il se veut un manuel de référence sur la technologie PowerShell. Il peut être lu de manière linéaire, mais aussi en consultation ponctuelle pour traiter un problème particulier.

Le public visé par ce livre est essentiellement constitué d'administrateurs et d'ingénieurs système et réseau, mais un programmeur peut tout à fait le lire.

#### PRÉCAUTION

Ce livre ne fera pas de vous un expert en PowerShell, car ce n'est pas sa finalité. Je ne vous y enseignerai pas comment écrire vos propres *providers* ou fournisseurs pour, par exemple, naviguer à travers une base de données SQL Server via le shell comme si vous navigiez à travers un système de fichiers. Vous n'y apprendrez pas non plus à écrire votre propre shell de A à Z, ou des modules particulièrement complexes pour créer des graphiques en 3D, ou encore à embarquer PowerShell de manière sécurisée dans votre application web.

## Les prérequis

Un administrateur qui n'a jamais fait de scripting dans sa vie peut très bien lire ce livre, car la pédagogie proposée ici part de zéro pour aller vers un niveau plus avancé. Néanmoins, une connaissance de langages de scripts comme VBScript serait un plus indéniable, notamment pour comprendre le langage en tant que tel. De même, savoir utiliser l'invite de commande (le classique *Command Prompt* fourni par Microsoft) pourra vous aider.

Par ailleurs, bien comprendre le fonctionnement du système d'exploitation Windows est capital pour pouvoir administrer cette plate-forme comme il se doit. Cela implique surtout de bien se documenter avant de procéder à l'automatisation d'une tâche particulière, mais cela vaut pour toutes les technologies supportant le scripting.

Chaque exemple menant à une pratique particulière a été volontairement simplifié de sorte que le lecteur puisse s'y retrouver facilement. Il n'est pas du tout question de surcharger le lecteur avec des exemples compliqués qui ne l'aideront pas à avancer. Ce qui compte avant tout est la philosophie et l'esprit mis en évidence à travers la simplicité de l'exemple.

Enfin, des notions de programmation orientée objet peuvent aider le lecteur à comprendre la philosophie de PowerShell.

## Structure de ce livre

Le livre s'articule autour de quatre parties :

- **Partie 1 : PowerShell, les bases**

Couvre les connaissances fondamentales pour appréhender PowerShell correctement, notamment une description de l'environnement en ligne de commande et de l'environnement de scripting, les bases de la syntaxe, le principe du pipeline, le format des données, la manipulation des objets, ainsi que la sécurité.

- **Partie 2 : PowerShell en tant que langage**

Cette partie explique de manière claire chaque élément de langage de PowerShell : variables, opérateurs, tableaux et dictionnaires, boucles, contrôle de flux, fonctions, modules, et la gestion d'erreur.

- **Partie 3 : PowerShell en pratique**

Dans cette troisième partie, on met en pratique les connaissances acquises dans les deux premières parties. Les sujets abordés sont : WMI, les services, les logs, la base de registres, COM et XML.

- **Partie 4 : Aller plus loin avec PowerShell**

Cette dernière partie explique au lecteur comment administrer des machines à distance, utiliser les workflows, manipuler les expressions régulières, automatiser la gestion de l'Active Directory, mieux connaître le framework .NET, et comment développer des interfaces graphiques facilitant l'utilisation des scripts écrits. Le dernier chapitre de ce livre s'attardera quelque peu sur la relation non évidente entre PowerShell et Linux. Il esquissera quelques ébauches de mon travail visant à créer une version stable de PowerShell sous Linux.

## Remerciements

Tout d'abord, j'aimerais remercier les éditions Eyrolles pour leur soutien et leur suivi concernant le développement de ce projet, en particulier Muriel Shan Sei Fan et Laurène Gibaud pour leur accompagnement ainsi que leur disponibilité, ainsi qu'Anne Bougnoux, Sophie Hincelin, Géraldine Noiret et Gaël Thomas.

Mes premiers pas dans le monde de l'informatique n'auraient pas été possibles sans l'aide de Ahmed Atallah et Raouf Ben Yahyaten. Leur soutien précieux lors de mes débuts a été une source fondamentale de motivation.

Enfin, j'aimerais aussi remercier les membres de ma famille pour leurs encouragements au quotidien, ainsi que Mark James, Hideyoshi Nakagawa, Jonathan Lowell, Shinji Komano, Shams Eddine El Andalousi, Cédric Duchel, Sofian Ben Yedder, Abdelilah Alioui, Shakira Alhamwi, Deha Alhamwi, Katia Denat, Sarah Nielsen et Magalie Anderson.

Kais Ayari  
kais.ayari.psx@gmail.com

# Table des matières

---

## PARTIE 1

### **PowerShell, les bases ..... 1**

#### CHAPITRE 1

##### **Qu'est-ce que PowerShell ? ..... 3**

Découvrir le shell ..... 4

    Comment lancer la console ? ..... 4

    Les opérations d'édition de la console ..... 5

L'environnement intégré d'écriture de scripts ..... 6

    Accéder à ISE ..... 7

    Les raccourcis clavier ..... 9

L'aide intégrée de PowerShell ..... 10

    La cmdlet Get-Help ..... 10

    Les cmdlets Update-Help et Save-Help ..... 13

Les nouveautés de PowerShell v3 ..... 14

    L'ajout automatique de modules ..... 14

    Les tâches planifiées ..... 14

    Windows PowerShell Web Access ..... 14

    Les sessions et connexions persistantes ..... 15

    PowerShell Workflow ..... 15

    Syntaxe simplifiée pour les cmdlets Where-Object et Foreach-Object ..... 15

    Amélioration de PowerShell ISE ..... 15

    La possibilité de mettre l'aide à jour ..... 16

    À propos des nouveautés ..... 16

#### CHAPITRE 2

##### **Les bases de la syntaxe ..... 17**

Les commandes ..... 18

    Les cmdlets ..... 18

    Les fonctions ..... 19

    Les scripts ..... 20

Les commandes natives Windows .....	21
<b>Les paramètres .....</b>	<b>22</b>
Les modes d'analyse syntaxique .....	24
<i>Le mode expression</i> .....	24
<i>Le mode commande</i> .....	25
<i>Quelques exemples</i> .....	25
<b>Les alias .....</b>	<b>26</b>
<b>Les blocs de script .....</b>	<b>29</b>
Syntaxe d'un bloc de script .....	30
Utilisation des blocs de script .....	30

**CHAPITRE 3****Le pipeline..... 33**

Principe et fonctionnement du pipeline .....	34
Qu'est-ce qu'un pipeline ? .....	34
Le pipeline un peu plus dans le détail .....	35
<b>Le processus de liaison des paramètres .....</b>	<b>37</b>
Mode de fonctionnement .....	37
Liaison de paramètres par valeur .....	38
Liaison de paramètres par nom de propriété .....	39
<b>Analyser les erreurs de pipeline .....</b>	<b>40</b>
Examiner les erreurs de pipeline avec Trace-Command .....	40

**CHAPITRE 4****Manipuler les objets avec PowerShell..... 45**

PowerShell est purement orienté objet .....	46
Qu'est-ce qu'un objet ? .....	46
Utiliser un objet .....	47
<b>Maîtriser les flux d'objets .....</b>	<b>49</b>
Sélectionner parmi un jeu d'objets .....	49
Sélectionner certaines propriétés d'objets .....	50
Extraire le contenu d'une propriété .....	51
<b>Trier les objets .....</b>	<b>52</b>
Trier les objets en fonction des valeurs de propriétés .....	52
Trier les objets dans l'ordre décroissant .....	53
Trier les objets en éliminant les doublons .....	53
<b>Grouper les objets .....</b>	<b>54</b>
Grouper les objets en s'appuyant sur une valeur de propriété .....	54
<b>Comparer les objets .....</b>	<b>55</b>
Comparer sur la base de collections d'objets .....	55

Comparer sur la base de propriétés d'objets .....	56
<b>Filtrer les objets .....</b>	<b>59</b>
L'approche classique .....	59
L'approche simplifiée .....	60
<b>Effectuer des opérations sur chaque objet .....</b>	<b>61</b>
L'approche classique .....	62
L'approche simplifiée .....	62
<b>Créer des objets .....</b>	<b>63</b>
Créer un objet .NET .....	63
Ajouter des membres aux objets créés .....	64
<b>CHAPITRE 5</b>	
<b>PowerShell et la mise en forme des données .....</b>	<b>65</b>
Afficher les données de sortie sous forme de tableau .....	67
La cmdlet Format-Table .....	67
Les propriétés personnalisées .....	69
Afficher les données de sortie sous forme de liste .....	70
La cmdlet Format-List .....	70
Personnaliser l'affichage des données de sortie .....	73
La cmdlet Format-Custom .....	73
Afficher une seule propriété sous forme de tableau .....	75
La cmdlet Format-Wide .....	75
<b>CHAPITRE 6</b>	
<b>PowerShell et la sécurité .....</b>	<b>77</b>
Les stratégies d'exécution PowerShell .....	78
Fonctionnement des stratégies d'exécution .....	78
Les types de stratégies d'exécution .....	79
Obtenir un certificat pour signer les scripts .....	80
Les différents moyens d'obtenir un certificat .....	80
Créer un certificat autosigné .....	81
Signer numériquement des scripts .....	83
<b>PARTIE 2</b>	
<b>PowerShell en tant que langage .....</b>	<b>87</b>
<b>CHAPITRE 7</b>	
<b>Les variables .....</b>	<b>89</b>
Définir ce qu'est une variable .....	90
Les variables d'environnement .....	91

Les variables automatiques .....	93
Les variables de préférence .....	94

**CHAPITRE 8****Les opérateurs ..... 97**

Les opérateurs arithmétiques .....	98
Les opérateurs logiques .....	100
Les opérateurs de comparaison .....	101
Les opérateurs d'égalité .....	102
Les opérateurs de relation .....	102
Les opérateurs de correspondance .....	102
L'opérateur -replace .....	103
Les opérateurs -like et -notlike .....	103
Les opérateurs -in et -notin .....	104
Les opérateurs d'affectation .....	104
Les opérateurs de redirection .....	107
Les opérateurs de fractionnement et de jointure .....	111
L'opérateur -split .....	111
L'opérateur -join .....	112
Les opérateurs de type .....	112
Les opérateurs spéciaux .....	114
L'opérateur d'appel .....	114
L'opérateur de déréférencement .....	114
L'opérateur de type « dot sourcing » .....	115
L'opérateur de membre statique .....	115
L'opérateur de plage .....	115
L'opérateur de mise en forme .....	115
L'opérateur de sous-expression .....	116
L'opérateur de sous-expression de tableau .....	116
L'opérateur virgule .....	116
L'opérateur de pipeline .....	116
L'opérateur de transtypage .....	117
L'opérateur d'index .....	117

**CHAPITRE 9****Les tableaux et dictionnaires ..... 119**

Les tableaux .....	120
Créer un tableau .....	120
Utiliser les tableaux .....	121
Les dictionnaires .....	124

Créer un dictionnaire .....	124
Utiliser les dictionnaires .....	125
<b>CHAPITRE 10</b>	
<b>Les boucles.....</b>	<b>129</b>
La boucle for .....	130
La boucle foreach .....	131
La boucle do..while .....	133
La boucle do..until .....	134
La boucle while .....	135
<b>CHAPITRE 11</b>	
<b>Les instructions conditionnelles .....</b>	<b>137</b>
L'instruction conditionnelle if..else .....	138
L'instruction conditionnelle if..elseif..else .....	139
L'instruction switch .....	140
<b>CHAPITRE 12</b>	
<b>Les fonctions.....</b>	<b>145</b>
Comprendre les fonctions .....	146
Déclarer une fonction .....	146
Les paramètres .....	147
<i>La variable \$args</i> .....	147
<i>Les paramètres nommés</i> .....	148
<i>Les paramètres positionnels.</i> .....	149
Explorer un peu plus les fonctions .....	149
La syntaxe dans le détail .....	149
Un exemple concret .....	151
Documenter les fonctions .....	152
Syntaxe de l'aide .....	152
Description des mots-clés .....	153
Exemple d'une fonction documentée .....	153
Les fonctions avancées .....	156
Principe des fonctions avancées .....	156
Les paramètres avancés .....	157
<b>CHAPITRE 13</b>	
<b>Les modules .....</b>	<b>161</b>
Qu'est-ce qu'un module PowerShell? .....	162
Se familiariser avec les modules PowerShell .....	162

Créer un module PowerShell .....	165
Écrire un module en langage PowerShell .....	166
Un mot sur les PSSnapins .....	169
<b>CHAPITRE 14</b>	
<b>PowerShell et la gestion d'erreur .....</b>	<b>171</b>
Différencier l'erreur de l'exception .....	172
Comprendre l'anatomie d'une erreur .....	173
Le paramètre <code>-ErrorAction</code> et la variable <code>\$ErrorActionPreference</code> .....	176
L'instruction <code>trap</code> .....	178
L'instruction <code>try..catch..finally</code> .....	180
<b>PARTIE 3</b>	
<b>PowerShell en pratique .....</b>	<b>183</b>
<b>CHAPITRE 15</b>	
<b>L'infrastructure WMI et CIM .....</b>	<b>185</b>
Définir ce qu'est WMI .....	186
Utiliser WMI .....	189
Les cmdlets WMI .....	190
WMI en pratique .....	190
Utiliser les cmdlets CIM .....	193
Les cmdlets CIM .....	194
CIM en action .....	194
<b>CHAPITRE 16</b>	
<b>Gérer les services .....</b>	<b>197</b>
Lister les services .....	198
Démarrer, arrêter, redémarrer et interrompre l'exécution d'un service .....	200
Démarrer un service .....	200
Arrêter un service .....	201
Redémarrer un service .....	201
Interrompre un service en cours d'exécution .....	201
Configurer le mode de démarrage d'un service .....	202
Modifier le compte de connexion d'un service .....	203
<b>CHAPITRE 17</b>	
<b>Gérer les logs .....</b>	<b>207</b>
Lister les logs .....	208
Configurer et sauvegarder les logs .....	212

Configurer les logs .....	212
Sauvegarder les logs .....	213
Supprimer les logs .....	214
<b>CHAPITRE 18</b>	
<b>Gérer la base de registres .....</b>	<b>215</b>
Naviguer dans la base de registres .....	216
Chercher, ajouter et effacer un élément de registre .....	219
Chercher des informations dans la base de registres .....	219
Ajouter un élément dans la base de registres .....	221
Effacer un élément de la base de registres .....	222
La base de registres à distance .....	223
<b>CHAPITRE 19</b>	
<b>PowerShell et COM .....</b>	<b>227</b>
Manipuler un objet COM .....	228
Automatiser Internet Explorer .....	230
Accéder automatiquement à une page web .....	230
S'authentifier automatiquement sur un site web .....	232
Mapper un lecteur réseau .....	233
Utiliser l'objet FileSystem .....	235
Créer un fichier texte .....	236
Ouvrir et lire un fichier texte .....	237
<b>CHAPITRE 20</b>	
<b>PowerShell et XML .....</b>	<b>239</b>
Manipuler un fichier XML .....	240
Les cmdlets Export-Clixml et Import-Clixml .....	244
Rechercher des informations à l'aide de la cmdlet Select-Xml .....	247
<b>PARTIE 4</b>	
<b>Aller plus loin avec PowerShell .....</b>	<b>249</b>
<b>CHAPITRE 21</b>	
<b>L'administration à distance .....</b>	<b>251</b>
Configurer l'accès à distance .....	252
Les connexions temporaires .....	254
Les connexions persistantes .....	257
Quel type de connexion privilégié? .....	260
Se déconnecter d'une session pour se reconnecter plus tard .....	261

**CHAPITRE 22****Les workflows sous PowerShell ..... 265**

Qu'est-ce qu'un workflow ? .....	266
Les workflows comme synonyme de productivité .....	266
La relation PowerShell-WWF .....	267
La différence entre un workflow et une fonction .....	271
La syntaxe .....	272
Paralleliser des opérations .....	272
Effectuer des opérations de manière séquentielle .....	273
L'instruction InlineScript .....	274
Les workflows en pratique .....	275

**CHAPITRE 23****Les expressions régulières ..... 279**

Un peu de syntaxe .....	280
La classe [Regex] .....	282
Les opérateurs -match et -notmatch .....	284
La cmdlet Select-String .....	286

**CHAPITRE 24****Automatiser l'Active Directory ..... 289**

Comprendre ce qu'est l'Active Directory .....	290
Qu'est-ce que l'Active Directory ? .....	290
Le module Active Directory .....	291
Les utilisateurs et groupes Active Directory .....	294
Administre les comptes d'utilisateurs .....	295
Administre les groupes .....	302
Créer un domaine Active Directory .....	305
Les zones DNS de recherches directes et inversées .....	307
Créer une zone DNS de recherche directe .....	308
Créer une zone DNS de recherche inversée .....	309

**CHAPITRE 25****Utiliser le framework .NET ..... 311**

Comprendre le framework .NET .....	312
Ajouter un type .NET à une session PowerShell .....	313
Envoyer un e-mail .....	316
Collecter des informations DNS .....	318

## CHAPITRE 26

**Développer une interface graphique avec PowerShell..... 323**

Windows Forms ou Windows Presentation Foundation ? .....	324
PowerShell Studio 2012 .....	325
Créer une interface graphique .....	325
Écrire le code .....	327

## CHAPITRE 27

**PowerShell sous Linux : rêve ou réalité ?..... 337**

PowerShell sous Linux est une réalité .....	338
Gérer les services .....	340
Gérer les modules kernel .....	341
Gérer les disques .....	342
Utiliser le framework .NET .....	344

**Index..... 347**



## PARTIE 1

# **PowerShell, les bases**



# 1

## Qu'est-ce que PowerShell ?

---

*Avant toute étude des particularités de PowerShell, il est indispensable de faire connaissance avec cette technologie. Sa richesse et sa complexité requièrent une approche graduée. Nous verrons donc comment accéder au shell, qui est l'interface en ligne de commande. Puis, nous explorerons l'environnement de scripting, ou ISE (Integrated Scripting Environment). PowerShell fournit une aide extrêmement utile, qui évite de recourir systématiquement à Internet pour résoudre des problèmes ; apprendre à l'utiliser est donc un élément capital pour être plus efficace dans une démarche où le temps est très précieux. Enfin, nous énumérerons les nouveautés de la dernière version de PowerShell, la version 3.*

### SOMMAIRE

- ▶ Découvrir le shell
- ▶ L'environnement de scripting
- ▶ L'aide intégrée
- ▶ Les nouveautés liées à PowerShell v3

## Découvrir le shell

Ce que l'on nomme *shell* dans la perspective de cet ouvrage est l'interface en ligne de commande, ou CLI (*Command-Line Interface*). Cela dit, un shell peut aussi être de type graphique. Lorsqu'un utilisateur navigue dans l'explorateur Windows, il utilise un shell graphique. Mais beaucoup d'utilisateurs réduisent la signification du mot shell au sens d'interface en ligne de commande, ignorant que ce mot a d'autres acceptations.

### Comment lancer la console ?

PowerShell est disponible d'emblée dans Windows 8 (sans vouloir évoquer les autres versions de ce système). Pour y accéder, nous pouvons au moins citer deux méthodes.

Pour la première méthode :

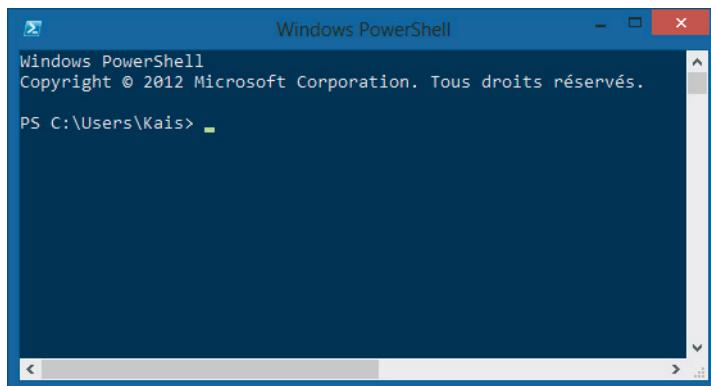
- 1 Écran *Démarrer*.
- 2 Taper *PowerShell* avec votre clavier.
- 3 Cliquer sur l'icône *Windows PowerShell* qui apparaît.

Pour la seconde méthode :

- 1 Déplacer le curseur de la souris jusqu'au coin en haut à droite du Bureau.
- 2 Cliquer sur *Rechercher*.
- 3 Taper *PowerShell* avec votre clavier.
- 4 Cliquer sur l'icône *Windows PowerShell* qui apparaît.

La console s'affiche alors à l'écran.

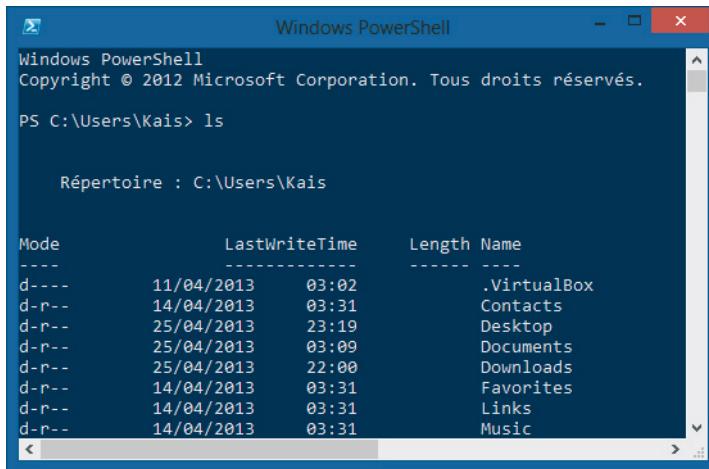
**Figure 1–1**  
Console PowerShell lancée depuis Windows 8



Comme on le voit, l'interface est tout à fait classique et ne présente pas de particularités esthétiques évidentes. D'ailleurs tel n'est pas son but. Essayons simplement d'obtenir la liste des fichiers et répertoires.

**Figure 1–2**

Lancement d'une commande de base



A screenshot of a Windows PowerShell window titled "Windows PowerShell". The window shows the command "PS C:\Users\Kais> ls" followed by the output of the "ls" command. The output lists several folders in the current directory (C:\Users\Kais) with their mode, last write time, length, and name. The output is as follows:

Mode	LastWriteTime	Length	Name
d----	11/04/2013 03:02		.VirtualBox
d-r--	14/04/2013 03:31		Contacts
d-r--	25/04/2013 23:19		Desktop
d-r--	25/04/2013 03:09		Documents
d-r--	25/04/2013 22:00		Downloads
d-r--	14/04/2013 03:31		Favorites
d-r--	14/04/2013 03:31		Links
d-r--	14/04/2013 03:31		Music

En tapant la commande `ls`, on obtient une liste ordonnée de répertoires. C'est donc du texte. C'est évident me direz-vous, mais en réalité, nous avons ici toute une série d'objets (que nous étudierons plus tard dans cet ouvrage).

Voilà donc à quoi ressemble la console PowerShell. Surtout, il ne faut pas se fier aux apparences. En effet, PowerShell est extrêmement robuste, permettant d'accéder à quasiment tout le système d'exploitation. Le framework .NET est le noyau central de cette technologie, offrant des perspectives de programmation très grandes et satisfaisant aussi bien les administrateurs que les programmeurs.

**NOTE Concernant le nom Windows PowerShell**

Le nom originel n'est pas Windows PowerShell, mais Monad. Ce terme nous vient du philosophe allemand Leibniz. Il désigne un atome insécable faisant partie des éléments des choses. Pour lui, tout l'univers est constitué de monades. Donc, ce que nous voyons est un ensemble de monades, autrement dit des agrégations d'atomes constituant une réalité. Pour faire le lien avec Windows PowerShell, les cmdlets sont des monades ; invoquées les unes avec les autres, elles produisent une réalité concrète.

## Les opérations d'édition de la console

Le tableau suivant donne la liste des touches permettant les opérations de déplacement et d'édition – qui seront familières aux utilisateurs sous Unix/Linux.

**Tableau 1–1** Opérations d'édition de la console PowerShell

Touches de clavier	Opération d'édition
<i>Flèches haut/bas</i>	Permet de se déplacer dans l'historique des commandes.
<i>Flèches gauche/droite</i>	Déplacent le curseur vers la gauche et la droite sur la ligne de commande.
<i>Insertion</i>	Bascule en mode insertion.
<i>F7</i>	Affiche l'historique des commandes saisies.
<i>Espace arrière</i>	Supprime le caractère se trouvant juste avant le curseur.
<i>Tab</i>	Complète les commandes tapées à l'écran.
<i>Ctrl + flèche gauche</i>	Déplace le curseur vers la gauche, mais mot à mot.
<i>Ctrl + flèche droite</i>	Déplace le curseur vers la droite, mais mot à mot.
<i>Début</i>	Déplace le curseur au tout début de la ligne de commande.
<i>Fin</i>	Déplace le curseur à la fin de la ligne de commande.

Nous pouvons constater que ces opérations d'édition sont identiques dans d'autres shells. Elles sont une sorte d'acquis universel et intemporel. À présent, passons à un environnement un peu plus convivial, PowerShell ISE.

## L'environnement intégré d'écriture de scripts

Avant l'arrivée de PowerShell version 2, les utilisateurs écrivaient leurs scripts sur des éditeurs de texte classiques comme Notepad, car il n'y avait à l'époque pas d'environnement spécifique, à l'exception des solutions commerciales comme PrimalScript et autres PowerShell Plus. Depuis la version 2, un environnement de scripting dédié est disponible : PowerShell ISE (*Integrated Scripting Environment*), qui améliore considérablement l'ergonomie du scripting.

ISE est une application embarquant l'environnement d'exécution de PowerShell, c'est-à-dire que toutes les commandes tapées, à quelques exceptions près, fonctionnent comme si elles étaient lancées à partir d'une console. La différence réside essentiellement dans le type d'expérience. Si nous désirons écrire de longs scripts, il vaut mieux utiliser des éditeurs comme ISE, ne serait-ce que pour la coloration syntaxique. De plus, cela facilite la maintenance des scripts.

## Accéder à ISE

Pour accéder à ISE, nous évoquerons ici encore deux méthodes, toujours à partir de Windows 8, mais il en existe d'autres.

Première méthode :

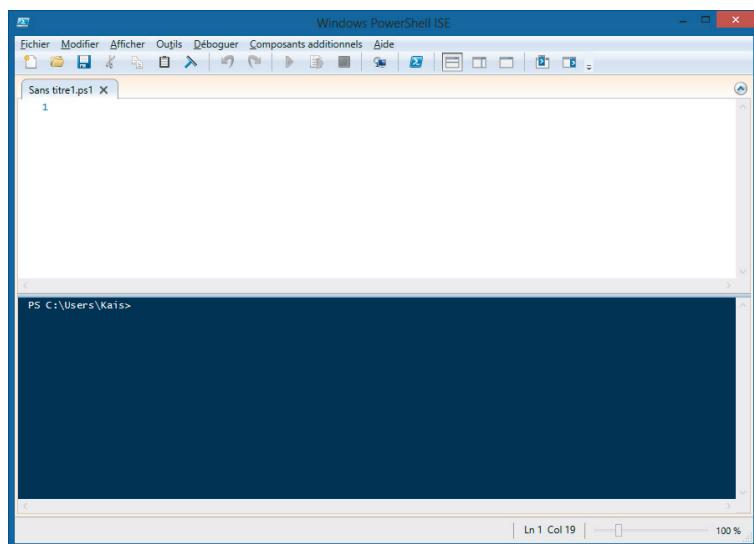
- 1 Écran *Démarrer*.
- 2 Taper *PowerShell* avec votre clavier.
- 3 Cliquer sur l'icône *Windows PowerShell* qui apparaît.
- 4 Depuis la console, tapez *ise* ; normalement, l'application ISE devrait apparaître.

Deuxième méthode :

- 1 Déplacer le curseur de la souris jusqu'au coin en haut à droite du Bureau.
- 2 Cliquer sur *Rechercher*.
- 3 Taper *powershell\_ise* avec votre clavier.
- 4 Cliquer sur l'icône *powershell\_ise.exe* qui apparaît.

**Figure 1–3**

L'affichage par défaut de PowerShell ISE



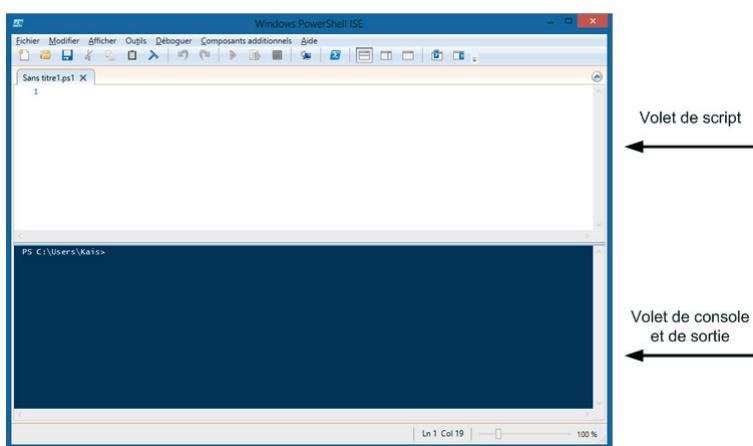
À partir de cet affichage, nous pouvons observer qu'il y a par défaut deux volets :

- le volet de script ;
- le volet de console, qui est aussi un volet de sortie.

La figure suivante fait bien la distinction entre les deux.

**Figure 1–4**

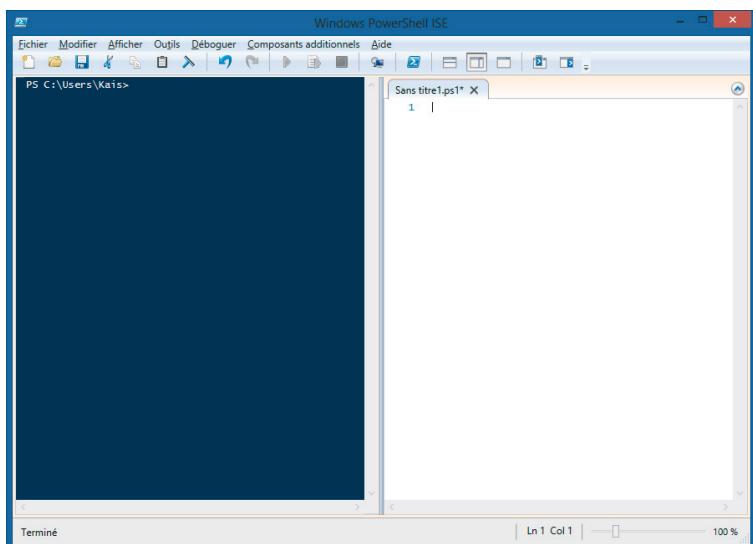
L'affichage est divisé en deux parties : l'une pour l'écriture de scripts et l'autre pour l'utilisation de la console ainsi que l'analyse des sorties.



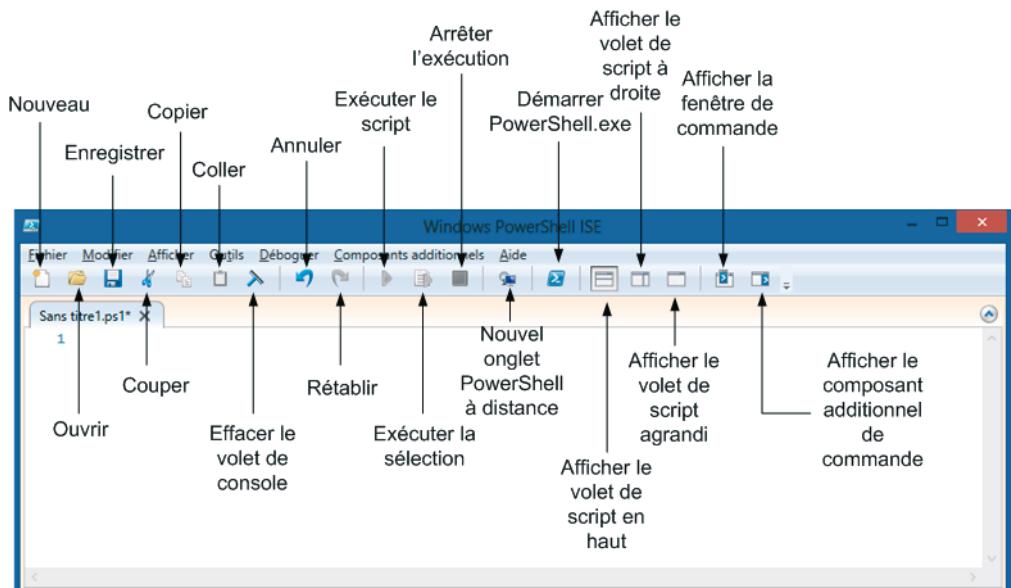
Ici, la disposition des volets est horizontale, mais elle peut très bien être verticale comme le montre la figure suivante. En fait, il n'y a pas de meilleure disposition ; l'utilisateur choisit en fonction de ses goûts.

**Figure 1–5**

Affichage vertical des deux volets



ISE fournit une barre d'outils permettant d'éditer ses scripts rapidement (figure ci-contre).



**Figure 1–6** La barre d'outils permet un certain nombre d'actions essentielles pour l'édition de scripts.

ISE est donc un environnement de développement de scripts à part entière. Il fournit l'essentiel pour écrire des scripts de bonne qualité.

## Les raccourcis clavier

Dans l'optique d'une édition rapide et efficace des scripts, PowerShell ISE propose un certain nombre de raccourcis clavier, indispensables pour gagner du temps. Nous donnons ci-dessous les plus utiles au quotidien.

**Tableau 1–2** Raccourcis clavier dans ISE

Raccourci clavier	Type d'action
<i>Ctrl + S</i>	Enregistre le fichier ou script actuellement utilisé par l'utilisateur.
<i>Ctrl + N</i>	Crée un nouveau fichier ou script.
<i>Ctrl + C</i>	Copie la sélection de texte.
<i>Ctrl + V</i>	Colle le texte précédemment copié.
<i>Ctrl + F</i>	Exécute une recherche rapide.
<i>Ctrl + H</i>	Effectue un remplacement dans le fichier actuellement utilisé.
<i>Ctrl + G</i>	Déplace le curseur vers une ligne spécifique.

**Tableau 1–2** Raccourcis clavier dans ISE (suite)

Raccourci clavier	Type d'action
<i>Ctrl + T</i>	Crée un nouvel onglet PowerShell.
<i>Ctrl + R</i>	Gère la visibilité du volet de script.
<i>F1</i>	Ouvre une rubrique d'aide consacrée à ISE.
<i>F8</i>	Exécute la sélection de texte.
<i>F5</i>	Exécute le fichier actuellement utilisé. Avant l'exécution, le fichier est enregistré.
<i>F3</i>	Lorsque le raccourci <i>Ctrl + F</i> est utilisé, passe à la prochaine occurrence.
<i>Shift + F3</i>	Lorsque le raccourci <i>Ctrl + F</i> est utilisé, retourne à l'occurrence précédente.
<i>Ctrl + Shift + P</i>	Démarre une nouvelle instance de PowerShell. L'intérêt est d'y recourir lorsque certaines commandes nécessitent une véritable console.
<i>Alt + F4</i>	Utilisé lorsque l'utilisateur veut quitter l'application.
<i>Entrée</i>	Insère une ligne (dans le volet de script) ou exécute la ligne de commande (dans le volet de console).

Évidemment, il y en a bien d'autres documentés, voire non documentés. En réalité, ISE nécessiterait un chapitre à lui tout seul. Nous n'avons abordé ici que l'essentiel nous permettant d'être productifs rapidement.

## L'aide intégrée de PowerShell

Une des grandes forces de PowerShell réside dans sa documentation. Microsoft a en effet pris soin de documenter l'ensemble des commandes, mais aussi tous les éléments qui concernent le langage, l'administration à distance et bien d'autres sujets. La commande d'obtention de l'aide s'appelle `Get-Help`.

### La cmdlet `Get-Help`

Dans le monde de Linux, pour ne citer que celui-là, les administrateurs ont pour habitude depuis longtemps d'utiliser la commande `man` pour obtenir de l'aide sur des sujets précis ; elle est sans doute la plus utilisée, à moins d'avoir une mémoire phénoménale et de mémoriser toute la syntaxe. De même, avec PowerShell, les administrateurs disposent de la cmdlet `Get-Help` pour lire la documentation. L'utilisateur peut même utiliser les fonctions `man` ou `help`, qui sont basées sur `Get-Help`.

**À SAVOIR Lien entre Get-Help, help et man**

Il faut savoir que `Get-Help` est une véritable cmdlet. La commande `help` est quant à elle une fonction utilisant `Get-Help`. Enfin, `man` est l'alias de la fonction `help`.

Toute l'aide est stockée dans des fichiers XML. La cmdlet `Get-Help` y a recours à chaque fois qu'elle est invoquée et la sortie produite par cette commande est extraite de ces fichiers d'aide. Il faut donc faire attention à ne pas les effacer, car dans ce cas, la documentation sera perdue et il faudra la télécharger de nouveau.

Pour illustrer l'utilisation de cette commande, essayons d'obtenir de l'aide concernant la cmdlet `Get-Process` :

PS> **Get-Help Get-Process**

**NOM**

Get-Process

**RÉSUMÉ**

Obtient les processus qui s'exécutent sur l'ordinateur local ou un ordinateur distant.

**SYNTAXE**

```
Get-Process [[-Name] <string[]>] [-ComputerName <string[]>]
[-FileVersionInfo] [-Module] [<CommonParameters>]
```

```
Get-Process -Id <Int32[]> [-ComputerName <string[]>]
[-FileVersionInfo] [-Module] [<CommonParameters>]
```

```
Get-Process -InputObject <Process[]> [-ComputerName <string[]>]
[-FileVersionInfo] [-Module] [<CommonParameters>]
```

**DESCRIPTION**

L'applet de commande `Get-Process` obtient les processus présents sur un ordinateur local ou distant.

Sans paramètre, `Get-Process` obtient tous les processus présents sur l'ordinateur local. Vous pouvez également spécifier un processus particulier en indiquant son nom ou son identificateur de processus, ou passer un objet processus à `Get-Process` via le pipeline.

Par défaut, `Get-Process` retourne un objet processus qui possède des informations détaillées sur le processus et prend en charge des méthodes qui vous permettent de démarrer et d'arrêter le processus. Vous pouvez également utiliser les paramètres de `Get-Process` pour obtenir des informations sur la

version de fichier du programme qui s'exécute dans le processus et obtenir les modules qui ont été chargés par le processus.

#### LIENS CONNEXES

Online version: <http://go.microsoft.com/fwlink/?LinkID=113324>  
[Get-Process](#)  
[Start-Process](#)  
[Stop-Process](#)  
[Wait-Process](#)  
[Debug-Process](#)

#### REMARQUES

Pour consulter les exemples, tapez : "get-help Get-Process -examples".  
Pour plus d'informations, tapez : "get-help Get-Process -detailed".  
Pour obtenir des informations techniques, tapez : "get-help Get-Process -full".

Comme nous pouvons le voir, la sortie est structurée en parties très explicites :

- *Nom* : donne le nom de la cmdlet.
- *Résumé* : décrit succinctement le fonctionnement de la cmdlet.
- *Syntaxe* : expose la syntaxe de la cmdlet avec ses paramètres.
- *Description* : décrit le fonctionnement de la cmdlet de manière plus détaillée.
- *Liens connexes* : dirige vers la version en ligne de l'aide et indique des cmdlets en lien avec la cmdlet en question.
- *Remarques* : donne des exemples pour obtenir plus de détails.

Ce n'est pas tout : la cmdlet [Get-Help](#) dispose de paramètres fournissant plus de détails.

**Tableau 1-3** Paramètres de la commande Get-Help

Paramètre	Description
<a href="#">-Category</a>	Affiche de l'aide sur des éléments se trouvant dans une catégorie spécifique. Les catégories sont <i>Alias</i> , <i>Cmd</i> , <i>Provider</i> et <i>HelpFile</i> .
<a href="#">-Detailed</a>	Ajoute à l'affichage d'aide de base des descriptions de paramètres et des exemples.
<a href="#">-Examples</a>	Affiche uniquement le nom, le résumé et les exemples liés à la cmdlet.
<a href="#">-Full</a>	Affiche toutes les rubriques d'aide d'une cmdlet.
<a href="#">-Name</a>	Fournit de l'aide au sujet d'une commande, rubrique conceptuelle, fournisseur, alias, script ou fonction.
<a href="#">-Online</a>	Affiche la version en ligne de l'aide.
<a href="#">-Parameter</a>	Affiche l'aide détaillée du paramètre spécifié en argument.
<a href="#">-ShowWindow</a>	Affiche l'aide dans une nouvelle fenêtre. Facilite la consultation de l'aide tout en continuant à utiliser la console.

**NOTE Paramètres de la cmdlet Get-Help.**

`Get-Help` propose d'autres paramètres que ceux présentés dans le tableau. Nous n'avons listé ici que les plus utilisés. Pour plus d'informations, tapez : `help Get-Help -full`.

À présent que nous avons parcouru la commande `Get-Help`, dont nous avons constaté l'importance, passons à deux autres commandes tout aussi importantes, nommées `Update-Help` et `Save-Help`.

## Les cmdlets Update-Help et Save-Help

La commande `Update-Help` est une nouveauté apportée avec PowerShell version 3, qui est installé d'office avec Windows Server 2012 et Windows 8. Le fait est que l'aide disponible n'est pas fournie ; elle doit donc être téléchargée, grâce à la cmdlet `Update-Help`.

Cette commande offre une véritable souplesse, car elle peut télécharger les fichiers d'aide sur Internet, mais aussi dans un endroit spécifique comme un chemin d'accès vers un serveur. Un scénario intéressant est de placer l'aide dans un serveur de fichiers et de faire pointer une fois par mois toutes les machines concernées vers ce serveur pour la télécharger. De cette façon, la documentation sera à jour. Cette commande est une excellente innovation de la version 3 de PowerShell.

Pour mettre l'aide à jour, lancez une session PowerShell en mode administrateur et tapez la ligne de commande suivante :

```
PS> Update-Help -force
```

Une bonne pratique consiste à lancer cette ligne de commande une ou deux fois par mois.

Pour les administrateurs préférant centraliser la documentation ou l'aide, une autre commande est à leur disposition : `Save-Help`. Cette cmdlet stocke les fichiers d'aide dans un endroit sécurisé. Cela aide effectivement à un meilleur contrôle, mais surtout permet à des machines n'ayant pas accès à Internet de disposer d'une documentation à jour. Voici comment l'utiliser :

```
PS> Save-Help -DestinationPath \\NTFileServer\PowerShellDocs
```

Dans cet exemple, nous téléchargeons la documentation pour tous les modules vers un serveur de fichiers. Une fois cette étape franchie, `Update-Help` peut être invoquée

avec le paramètre `-SourcePath` pour télécharger la documentation depuis le serveur de fichiers, et non pas Internet :

```
PS> Update-Help -SourcePath \\NTFileServer\PowerShellDocs
```

Cette combinaison `Save-Help/Update-Help` démontre bien l'amélioration qui a été apportée à PowerShell concernant la gestion de la documentation.

## Les nouveautés de PowerShell v3

PowerShell apporte avec la version 3 un certain nombre de nouveautés. Nous allons présenter les plus importantes.

### L'ajout automatique de modules

Avec la version 2, il était nécessaire d'appeler la commande `Import-Module` lorsqu'un script ou une autre ligne de commande faisait appel à des cmdlets qui n'étaient pas chargées automatiquement lors du lancement d'une session. Maintenant, il n'est plus nécessaire de passer par cette étape, car les modules sont ajoutés automatiquement lorsqu'une commande est appelée.

### Les tâches planifiées

Les tâches en arrière-plan (ou *jobs*) peuvent à présent être planifiées. Pour rappel, elles s'exécutent de manière asynchrone, ce qui permet de les lancer et de continuer à travailler. Attention à ne pas confondre les jobs et les jobs planifiés.

### Windows PowerShell Web Access

Windows PowerShell Web Access est sans doute l'une des plus brillantes nouveautés sur cette nouvelle version. Elle concerne Windows Server 2012 et permet de créer le rôle de passerelle PowerShell, basé sur le Web. L'objectif pour un administrateur est, par exemple, d'accéder à une console PowerShell via un client qui peut être un ordinateur portable, un ordinateur de bureau, un téléphone portable, en fait n'importe quel périphérique client qui prend en charge JavaScript et accepte les cookies. Dans ce contexte, toutes les commandes envoyées le sont via le navigateur, donc il n'est pas nécessaire d'installer PowerShell.

## Les sessions et connexions persistantes

Les PSSessions, qui sont des sessions spécifiques parmi d'autres dans PowerShell, ont à présent la caractéristique de pouvoir être persistantes. Par exemple, un informaticien créant une session à distance peut, s'il le souhaite, se déconnecter puis se reconnecter à un moment ultérieur, depuis le même poste ou un autre. La contrainte essentielle est que PowerShell version 3 soit installé des deux côtés.

## PowerShell Workflow

Cette nouveauté est à mon sens la plus importante de la version 3 : la possibilité d'utiliser simplement la puissance d'une technologie complexe comme *Windows Workflow Foundation*. Il n'y a plus besoin de passer par Visual Studio pour écrire ses propres workflows. Cette nouvelle fonctionnalité offre la perspective d'écrire des tâches visant des cibles de manière séquentielle ou parallèle, pouvant être interrompues puis relancées à nouveau, etc. Elle offre aussi une grande flexibilité dans le type d'action recherchée.

## Syntaxe simplifiée pour les cmdlets Where-Object et Foreach-Object

Ces deux commandes ont vu leur syntaxe simplifiée. L'utilisation des *scriptblocks* et de certains symboles a été repensée. Une approche un peu moins verbeuse est à présent possible. Ceci dit, il s'agit d'un plus et non pas d'une sorte de correction. On peut en effet toujours utiliser l'ancienne syntaxe. Prenons un exemple vraiment très simple pour illustrer notre propos, exemple qui peut avoir plusieurs variantes selon le type d'optimisation que l'on recherche.

Voici une syntaxe classique :

```
PS> Get-Service | Where-Object {$_.Status -eq 'Stopped'}
```

Et voici la syntaxe simplifiée :

```
PS> Get-Service | Where-Object -Property Status -eq -Value Stopped
```

Ces deux méthodes fonctionnent, chacun choisira celle qui lui convient le mieux.

## Amélioration de PowerShell ISE

ISE dispose de nouvelles fonctionnalités comme :

- l'autocomplétion des commandes ;
- la possibilité d'appeler des snippets ou extraits pour faciliter l'écriture de scripts ;

- la création de régions ;
- l'ajout d'un composant additionnel de commandes ;
- une grande marge de manœuvre dans la gestion des composants additionnels ;
- un débogueur amélioré ;
- un menu *option* pour gérer notamment tout ce qui est couleurs et polices, le volet de script, l'autocomplétion, ainsi que d'autres paramètres.

## La possibilité de mettre l'aide à jour

Cette fonctionnalité permet de télécharger les fichiers d'aide depuis Internet, mais aussi depuis d'autres sources comme un serveur de fichiers. La façon dont l'aide peut être gérée est donc plus souple, car des processus de vérification, correction et validation peuvent être mis en place.

## À propos des nouveautés

Évidemment, il existe d'autres nouveautés liées à la version 3, mais toutes les énumérer ici demanderait deux ou trois chapitres au minimum ! Car il y a celles que l'on pourrait qualifier de visibles et qui concernent l'utilisateur, et puis celles qui le sont moins et qui relèvent du fonctionnement interne de l'environnement d'exécution.

# 2

## Les bases de la syntaxe

---

*Dans sa syntaxe, PowerShell dérive de plusieurs langages. Sa grammaire s'est structurée essentiellement à partir de Korn Shell, à la base également d'autres langages de scripts. Par ailleurs, on reconnaît dans certaines structures les langages C, C#, Perl, PHP ainsi que Tcl, pour ne citer que les plus connus.*

*Dans ce chapitre, nous allons aborder les éléments de base qui constituent le fondement de PowerShell. D'abord, nous essaierons de définir ce que regroupe le mot commande, puis nous expliquerons ce que sont les paramètres et les modes d'analyse syntaxique, nous évoquerons les alias et enfin les blocs de script.*

### SOMMAIRE

- ▶ Comprendre les commandes
- ▶ Les paramètres
- ▶ Les modes d'analyse syntaxique
- ▶ Les alias
- ▶ Les blocs de script

## Les commandes

En PowerShell, le terme « commande » a plusieurs acceptations. Il peut signifier :

- les cmdlets ;
- les fonctions ;
- les scripts ;
- les commandes natives Windows.

Définir ces termes est une démarche fondamentale pour acquérir les bases.

### Les cmdlets

Une cmdlet ou applet de commande (appelée *commandlet*) est un ensemble de données structurées dans une classe .NET. Ces données regroupent le nom de la cmdlet, les types de ses paramètres, sa description, son comportement vis-à-vis des autres cmdlets, etc. Lorsque cette classe est implémentée, elle est compilée dans une DLL (*Dynamic Link Library*). Cette dernière peut ensuite être importée lors d'une session en cours, ou alors être chargée au démarrage d'une session.

#### NOTE Sur le mode de fonctionnement des cmdlets

Lorsqu'une session PowerShell est démarrée, les bibliothèques (ou DLL) sont chargées automatiquement, ce qui rend l'exécution des cmdlets plus rapide que celle des autres catégories de commandes.

Les cmdlets respectent des règles de nommage bien précises. Le format qu'elles adoptent est de type *Verbe-Nom*.

- Le verbe décrit le genre d'action à mener.
- Le nom décrit l'objet sur lequel nous voulons accomplir cette action.

Par exemple, la cmdlet `Get-Service` respecte parfaitement ce format.

- Le verbe `Get` signifie « obtenir ».
- Le nom `Service` signifie un objet ou une liste d'objets représentant les services.

Cela signifie en clair : « Lister un ou plusieurs objet(s) représentant les services. »

#### À SAVOIR Deux façons d'écrire des cmdlets

Depuis la version 2 de PowerShell, il est possible, et c'est une bonne chose pour les administrateurs qui ne sont pas programmeurs, d'écrire des cmdlets en langage PowerShell sous la forme de fonctions avancées (voir chapitre 12), et non plus seulement en langage .NET comme C#.

**RESSOURCE** Liste des verbes approuvés► <http://msdn.microsoft.com/en-us/library/windows/desktop/ms714428%28v=vs.85%29.aspx>**Figure 2-1**

Liste des verbes approuvés pour écrire des commandes PowerShell

The screenshot shows a web browser window displaying the Microsoft Dev Center - Desktop page. The URL in the address bar is [msdn.microsoft.com/en-us/library/windows/desktop/ms714428\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms714428(v=vs.85).aspx). The page title is "Approved Verbs for Windows PowerShell Commands". Below the title, there is a note: "Windows PowerShell uses a verb-noun pair for the names of cmdlets and for their derived Microsoft .NET Framework classes. For example, the **Get-Command** cmdlet provided by Windows PowerShell is used to retrieve all the commands that are registered in Windows PowerShell. The verb part of the name identifies the action that the cmdlet performs. The noun part of the name identifies the entity on which the action is performed." A "Note:" section follows, stating: "Windows PowerShell uses the term *verb* to describe a word that implies an action even if that word is not a standard verb in the English language. For example, the term *New* is a valid Windows PowerShell verb name because it implies an action even though it is not a verb in the English language." At the bottom, there is a "Verb Naming Rules" section with a bulleted list:

- When you specify the verb, we recommend that you use one of the predefined verb names provided by Windows PowerShell (aliases for these predefined verbs are included in the following tables). When you use a predefined verb, you ensure consistency between the cmdlets that you create, the cmdlets that are provided by Windows PowerShell, and the cmdlets that are designed by others.
- Use the predefined verbs to describe the general scope of the action, and use parameters to further refine the action of the cmdlet.
- To enforce consistency across cmdlets, do not use a synonym of an approved verb.

Il n'est absolument pas nécessaire d'utiliser des verbes de cette liste mais cela est fortement recommandé, car des problèmes peuvent sinon survenir. Par exemple, la commande **Get-Command** peut avoir des difficultés à détecter de nouvelles cmdlets ne respectant pas les règles de nommage.

## Les fonctions

Les fonctions constituent un autre type de commandes, aussi importantes que les cmdlets. Une fonction est une liste d'instructions portant un nom et résidant en mémoire. Elle peut être créée dynamiquement pendant la session, mais aussi écrite dans des fichiers de façon à pouvoir la réutiliser. À la base, la structure d'une fonction est identique à celle dans d'autres langages comme Bash, mais cette structure a évolué au fur et à mesure des versions de PowerShell.

### Fonction de base

```
function Write-Object{
    param($FirstParam)
    Write-Output "Le paramètre indiqué est: $Firstparam"
}
```

Les fonctions ressemblent beaucoup aux cmdlets, en ce sens qu'elles peuvent avoir des paramètres nommés ou positionnels, booléens ou dynamiques. Elles bénéficient des mêmes règles de nommage que les cmdlets, peuvent retourner des valeurs en sortie à l'écran, affecter ces dernières à des variables ou même les passer à d'autres commandes. Depuis la version 2 de PowerShell, les fonctions, en plus de pouvoir conserver leurs propres caractéristiques, peuvent fonctionner de manière très précise comme des cmdlets. Pour plus de détails, voir le chapitre 12.

## Les scripts

PowerShell considère les scripts comme un type de commande à part entière. Ils sont constitués d'au moins une commande (ou groupe de commandes) et sont contenus dans un fichier texte, qui doit avoir l'extension `.ps1`. Le principe du script est de pouvoir automatiser un certain nombre de tâches. En effet, lorsqu'un administrateur effectue des tâches d'administration de manière récurrente, la meilleure solution pour gagner du temps et éviter des erreurs est de scripter ces tâches. De plus, un script peut être exécuté non seulement par celui qui l'a écrit, mais aussi par d'autres personnes, et ce même si elles ne connaissent pas le contenu du script. Encore plus, un script, à partir du moment où il a été écrit et testé comme il se doit, peut être lancé en tâches planifiées. On voit donc l'immense gain de productivité procuré par ce type de commande.

L'exemple suivant liste l'ensemble des services en cours d'exécution.

### Exemple d'un script simple enregistré dans un fichier `svcrun.ps1`

```
get-service | where-object {$_.Status -eq "Running"}
```

Pour exécuter ce script, il faut d'abord modifier la stratégie d'exécution (voir chapitre 6). Puis, selon l'endroit où le script est enregistré (`C:\devscripts` pour l'exemple), tapons cette ligne de commande :

```
PS> c:\devscripts\svcrun.ps1
```

Si l'on se trouve dans le répertoire `C:\devscripts`, il n'est pas nécessaire de préciser tout le chemin d'accès vers le script :

```
PS> .\svcrun.ps1
```

#### À SAVOIR L'exécution des scripts et la sécurité

À l'époque de VBScript, il était relativement facile d'écrire des virus, car une fois le code écrit et enregistré dans un fichier `.vbs`, il suffisait à la victime de double-cliquer sur ce dernier pour que le code soit exécuté. Si vous essayez de double-cliquer sur un script PowerShell, cette action lancera un éditeur de texte ou de scripts. Ceci est simplement une correction en termes de sécurité, même si cette mesure est largement contournable.

Les scripts peuvent être très simples, mais aussi très complexes avec des paramètres, une documentation intégrée et la possibilité d'être lancés à distance.

## Les commandes natives Windows

Ce que l'on désigne par le terme de « commandes natives Windows » est constitué de tous les exécutables ou commandes qui ne relèvent pas de l'environnement de PowerShell, comme les cmdlets. Voici une petite liste de commandes considérées comme natives :

- `ipconfig.exe` ;
- `cacls.exe` ;
- `calc.exe` ;
- `fsutil.exe` ;
- `find.exe` ;
- `hostname.exe` ;
- `robocopy.exe`.

PowerShell ne les analyse pas comme des cmdlets ou fonctions. Ces commandes natives sont exécutées comme des processus séparés. Leur exécution est donc plus lente que celle des autres commandes, à cause de la séparation des processus. De plus, le traitement de leurs paramètres est différent, ce qui implique que leurs syntaxes doivent être étudiées de nouveau selon que l'on se trouve dans un environnement Command Prompt ou PowerShell.

Il est fortement recommandé de les éviter si l'objectif recherché peut être accompli via PowerShell, et ce, pour des raisons qui tiennent essentiellement à la performance, mais pas seulement.

## Les paramètres

Après avoir étudié ce que recouvrait le concept de commandes avec PowerShell, concept riche car regroupant de multiples notions, il est indispensable de comprendre ce que sont les paramètres, notion intimement liée aux commandes. Ces dernières reposent en effet sur des paramètres, qui modifient leur comportement en fournissant des entrées. Les paramètres ont la syntaxe suivante.

### Syntaxe d'un paramètre

1 -<nom du paramètre> 2 <valeur du paramètre> 3

Dans cette syntaxe, on observe le trait d'union 1, qui signale la présence d'un paramètre à PowerShell. Vient ensuite le nom du paramètre 2, qui est en réalité son identifiant. Enfin, le paramètre a souvent une valeur 3, même si ce n'est pas toujours le cas.

#### NOTE Sur le nom des paramètres

Les paramètres ne peuvent pas contenir d'espaces, ce qui veut dire que tous les mots les constituant doivent être attachés. De plus, il doit y avoir au moins une espace entre le nom du paramètre et la valeur correspondante.

Les paramètres peuvent être de plusieurs types et avoir de multiples contraintes, donc leurs comportements sont variables d'une commande (au sens large du terme) à une autre. Chaque paramètre a sa propre définition, c'est-à-dire un ensemble d'informations les caractérisant. Prenons l'exemple du paramètre `Name` de la cmdlet `Get-Service` :

```
PS> Get-Help Get-Service -Parameter Name

-Name <String[]>
Spécifie les noms des services à récupérer. Les caractères génériques sont autorisés. Par défaut, Get-Service obtient tous les services présents sur l'ordinateur.

Obligatoire ?          false
Position ?             1
Valeur par défaut      All services
Accepter l'entrée de pipeline ? true (ByPropertyName, ByValue)
Accepter les caractères génériques ? True
```

On voit ici la syntaxe du paramètre, sa description, ainsi qu'un certain nombre de détails très utiles pour l'utiliser.

- **Le caractère obligatoire ou non du paramètre**

Indique si le paramètre est obligatoire (valeur `true`) ou pas. Dans le cas où la valeur est `true` et où le paramètre n'est pas présent dans la ligne de commande, PowerShell invitera à fournir une valeur. Si la valeur est `false`, la commande pourra être exécutée sans la présence du paramètre.

- **La position du paramètre**

Cette valeur détermine clairement si le paramètre est positionnel ou pas. Si la valeur est `0` ou `named`, cela veut dire que le nom du paramètre est requis, et dans ce cas, le paramètre est nommé. Si la valeur est autre que les deux susmentionnées, par exemple `1` ou `2`, cela veut dire que le paramètre est positionnel et que le nom n'est pas obligatoire. Dans ce cas précis, la valeur indique la position par rapport à la commande.

- **La valeur par défaut**

Indique la valeur que le paramètre doit prendre si aucune valeur n'est spécifiée.

- **Accepte l'entrée de pipeline**

Indique si le paramètre accepte des valeurs via l'opérateur `|`. Les valeurs possibles sont :

- `False` : aucune valeur ne peut être redirigée vers le paramètre.
- `True (by Value)` : des valeurs peuvent être redirigées vers le paramètre en question à condition qu'elles soient acceptées par le paramètre ou qu'elles puissent être converties dynamiquement.
- `True (by Property Name)` : ici, des valeurs peuvent aussi être redirigées. La différence avec `by value` est que la valeur redirigée doit comporter une propriété ayant le même nom que le paramètre (ex : valeur avec propriété ayant le nom `Age` vers un paramètre ayant le nom `Age`).

- **Accepte les caractères génériques**

Indique si le paramètre peut contenir des caractères génériques. Cette option offre de plus larges possibilités dans la définition de la valeur du paramètre.

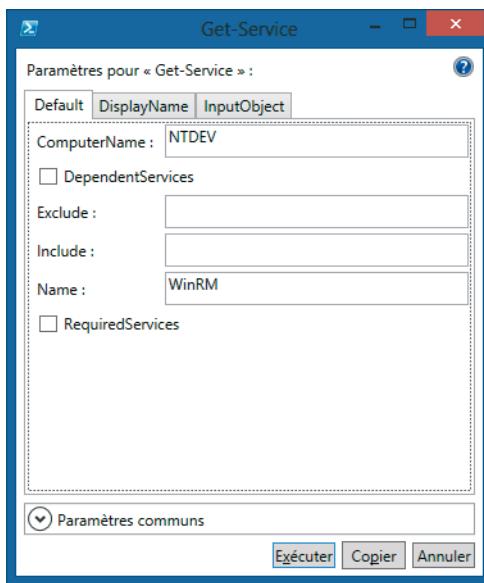
Une nouvelle cmdlet a fait son apparition avec PowerShell version 3 : `Show-Command`. Idéale pour les débutants, elle montre dans une interface graphique la commande analysée ainsi que ses paramètres. Reprenons l'exemple de la commande `Get-Service` :

```
PS> Show-Command Get-Service
```

Après avoir appuyé sur la touche `Entrée`, une fenêtre apparaît.

**Figure 2–2**

Show-Command permet de découvrir une commande avec ses paramètres.



Le but recherché à travers `Show-Command` est de se familiariser avec l'ensemble de la commande analysée, c'est-à-dire aussi ses paramètres. Via cette interface, nous pouvons fournir les entrées aux paramètres de la commande, puis soit l'exécuter directement, soit copier la commande, les paramètres et valeurs correspondants dans la console pour pouvoir analyser le contenu.

Savoir comment utiliser les paramètres est donc essentiel lorsque l'on manipule des commandes.

## Les modes d'analyse syntaxique

Lorsqu'une ligne de commande est entrée, PowerShell en analyse chaque élément. En effet, la ligne de commande, qui représente une unité d'exécution, est décomposée en jetons, un jeton étant un élément de la ligne de commande. Cette analyse est d'une certaine façon la première étape dans l'interprétation de l'unité d'exécution. Et contrairement à d'autres shells ou interpréteurs de commandes, PowerShell ne propose que deux modes d'analyse syntaxique : le mode expression et le mode commande.

### Le mode expression

Ce mode d'analyse est utilisé lorsque le début de la ligne de commande ne contient pas le nom d'une commande, qu'elle soit cmdlet, fonction ou commande native, de même pour les alias et les mots-clés. Dans ce cas de figure, les expressions sont évaluées en

majorité comme des chaînes de caractères ou des valeurs numériques. Les chaînes de caractères doivent être entre guillemets pour être traitées comme telles. Les nombres, pour être traités comme des valeurs numériques, ne doivent pas figurer entre guillemets.

## Le mode commande

Ce mode d'analyse, contrairement au précédent, est utilisé lorsque le début de la ligne de commande contient le nom d'une cmdlet, fonction, commande native ou un alias qui pointe vers une commande. Dans ce mode, les nombres sont traités comme des valeurs numériques et les autres arguments comme des chaînes de caractères, sauf si l'argument commence par l'un des caractères spéciaux suivants :

- le signe dollar \$ ;
- le signe arobase @ ;
- les guillemets simples ' ;
- les guillemets doubles " ;
- une parenthèse ouvrante (.

Si un argument commence avec l'un de ces caractères spéciaux, alors il sera traité comme une expression de valeur, c'est-à-dire que la valeur sera déterminée à partir de l'évaluation de l'argument.

## Quelques exemples

Le tableau suivant énumère un certain nombre d'exemples illustrant les différents modes d'analyse de PowerShell.

Tableau 2-1 Exemples illustrant les différents modes d'analyse

Exemple	Mode d'analyse	Résultat
7+7	Expression	14 (nombre entier)
Write-Output (7+7)	Expression	14 (nombre entier)
\$val=7+7	Expression	\$val=14 (nombre entier)
Write-Output 7+7	Commande	"7+7" (chaîne)
Write-Output \$a/7	Commande	"14/7" (chaîne)

PowerShell utilise ces modes d'analyse de manière transparente pour l'utilisateur. Toutefois, les connaître est important, car cela aide à comprendre la façon dont PowerShell fonctionne, même si un ouvrage tout entier pourrait être consacré au fonctionnement interne de cet outil.

## Les alias

Un alias est un surnom utilisé pour une commande, qu'elle soit cmdlet, script, fonction ou commande native. L'alias est présent dans de nombreux interpréteurs de commandes, il n'est donc pas un élément nouveau. L'utilisation des alias fait gagner beaucoup de temps, surtout lorsqu'on est un administrateur gérant au quotidien des urgences. Il faut donc comprendre l'alias comme un pointeur et non pas comme une véritable commande. Par exemple, la commande [Compare-Object](#) a comme alias le mot [compare](#). La liste des alias est disponible via la ligne de commande suivante :

```
PS> Get-Alias
```

CommandType	Name
Alias	% -> ForEach-Object
Alias	? -> Where-Object
Alias	ac -> Add-Content
Alias	asnp -> Add-PSSnapin
Alias	cat -> Get-Content
Alias	cd -> Set-Location
Alias	chdir -> Set-Location
Alias	clc -> Clear-Content
Alias	clear -> Clear-Host
Alias	clhy -> Clear-History
Alias	cli -> Clear-Item
Alias	clp -> Clear-ItemProperty
Alias	cls -> Clear-Host
Alias	clv -> Clear-Variable
Alias	cnsn -> Connect-PSSession
Alias	compare -> Compare-Object
Alias	copy -> Copy-Item
Alias	cp -> Copy-Item
Alias	cpi -> Copy-Item
Alias	cpp -> Copy-ItemProperty
Alias	cvpa -> Convert-Path
Alias	dbp -> Disable-PSBreakpoint
Alias	del -> Remove-Item
Alias	diff -> Compare-Object
Alias	dir -> Get-ChildItem
Alias	dnsn -> Disconnect-PSSession
Alias	ebp -> Enable-PSBreakpoint
Alias	echo -> Write-Output
Alias	epal -> Export-Alias
Alias	epcsv -> Export-Csv
Alias	epsn -> Export-PSSession
Alias	erase -> Remove-Item
Alias	etsn -> Enter-PSSession

Alias	exsn -> Exit-PSSession
Alias	fc -> Format-Custom
Alias	fl -> Format-List
Alias	foreach -> ForEach-Object
Alias	ft -> Format-Table
Alias	fw -> Format-Wide
Alias	gal -> Get-Alias
Alias	gbp -> Get-PSBreakpoint
Alias	gc -> Get-Content
Alias	gci -> Get-ChildItem
Alias	gcm -> Get-Command
Alias	gcs -> Get-PSCallStack
Alias	gdr -> Get-PSDrive
Alias	ghy -> Get-History
Alias	gi -> Get-Item
Alias	gjb -> Get-Job
Alias	gl -> Get-Location
Alias	gm -> Get-Member
Alias	gmo -> Get-Module
Alias	gp -> Get-ItemProperty
Alias	gps -> Get-Process
Alias	group -> Group-Object
Alias	gsn -> Get-PSSession
Alias	gsnp -> Get-PSSnapin
Alias	gsv -> Get-Service
Alias	gu -> Get-Unique
Alias	gv -> Get-Variable
Alias	gwmi -> Get-WmiObject
Alias	h -> Get-History
Alias	history -> Get-History
Alias	icm -> Invoke-Command
Alias	iel -> Invoke-Expression
Alias	ihy -> Invoke-History
Alias	ii -> Invoke-Item
Alias	ipal -> Import-Alias
Alias	ipcsv -> Import-Csv
Alias	ipmo -> Import-Module
Alias	ipsn -> Import-PSSession
Alias	irm -> Invoke-RestMethod
Alias	ise -> powershell_ise.exe
Alias	iwmi -> Invoke-WmiMethod
Alias	iwr -> Invoke-WebRequest
Alias	kill -> Stop-Process
Alias	lp -> Out-Printer
Alias	ls -> Get-ChildItem
Alias	man -> help
Alias	md -> mkdir
Alias	measure -> Measure-Object
Alias	mi -> Move-Item
Alias	mount -> New-PSDrive

Alias	move -> Move-Item
Alias	mp -> Move-ItemProperty
Alias	mv -> Move-Item
Alias	nal -> New-Alias
Alias	ndr -> New-PSDrive
Alias	ni -> New-Item
Alias	nmo -> New-Module
Alias	npssc -> New-PSSessionConfigurationFile
Alias	nsn -> New-PSSession
Alias	nv -> New-Variable
Alias	ogv -> Out-GridView
Alias	oh -> Out-Host
Alias	popd -> Pop-Location
Alias	ps -> Get-Process
Alias	pushd -> Push-Location
Alias	pwd -> Get-Location
Alias	r -> Invoke-History
Alias	rbp -> Remove-PSBreakpoint
Alias	rcjb -> Receive-Job
Alias	rcsn -> Receive-PSSession
Alias	rd -> Remove-Item
Alias	rdr -> Remove-PSDrive
Alias	ren -> Rename-Item
Alias	ri -> Remove-Item
Alias	rjb -> Remove-Job
Alias	rm -> Remove-Item
Alias	rmdir -> Remove-Item
Alias	rmo -> Remove-Module
Alias	rni -> Rename-Item
Alias	rnp -> Rename-ItemProperty
Alias	rp -> Remove-ItemProperty
Alias	rsn -> Remove-PSSession
Alias	rsnp -> Remove-PSSnapin
Alias	rujb -> Resume-Job
Alias	rv -> Remove-Variable
Alias	rvpa -> Resolve-Path
Alias	rwmi -> Remove-WmiObject
Alias	sajb -> Start-Job
Alias	sal -> Set-Alias
Alias	saps -> Start-Process
Alias	sasv -> Start-Service
Alias	sbp -> Set-PSBreakpoint
Alias	sc -> Set-Content
Alias	select -> Select-Object
Alias	set -> Set-Variable
Alias	shcm -> Show-Command
Alias	si -> Set-Item
Alias	sl -> Set-Location
Alias	sleep -> Start-Sleep
Alias	s1s -> Select-String

Alias	sort -> Sort-Object
Alias	sp -> Set-ItemProperty
Alias	spjb -> Stop-Job
Alias	spps -> Stop-Process
Alias	spsv -> Stop-Service
Alias	start -> Start-Process
Alias	subj -> Suspend-Job
Alias	sv -> Set-Variable
Alias	swmi -> Set-WmiInstance
Alias	tee -> Tee-Object
Alias	trcm -> Trace-Command
Alias	type -> Get-Content
Alias	where -> Where-Object
Alias	wjb -> Wait-Job
Alias	write -> Write-Output

Ces alias sont dits intégrés au démarrage de session et leur nombre varie évidemment en fonction de plusieurs facteurs (ceux créés par l'utilisateur en cours de session, ou même les alias créés dans le profil de l'utilisateur). Le tableau suivant dresse la liste des commandes disponibles pour utiliser les alias.

**Tableau 2–2** Liste des cmdlets permettant de travailler avec les alias

Cmdlet	Description
Get-Alias	Obtient les alias de la session active.
Export-Alias	Exporte vers un fichier les alias actuellement définis.
Import-Alias	Importe une liste d'alias à partir d'un fichier.
New-Alias	Crée un alias.
Set-Alias	Crée ou modifie un alias.

#### RECOMMANDATION Sur l'utilisation des alias

Il est fortement recommandé d'utiliser les alias en mode console et non pas en mode scripting, car s'ils contiennent plusieurs lignes avec des alias, ils risquent d'être mal interprétés lors de la maintenance et le résultat peut être désastreux. De plus, le but d'un alias n'est pas d'être utilisé dans un script.

## Les blocs de script

Les blocs de script (ou *scriptblocks*), présents dans plusieurs langages, représentent un des piliers fondamentaux de PowerShell. Il s'agit d'une collection d'instructions formant une unité d'exécution.

## Syntaxe d'un bloc de script

La syntaxe d'un bloc de script est la suivante.

### Syntaxe d'un bloc de script

```
{ <Liste d'instructions> } ①
```

La liste d'instructions est figurée entre accolades ①. Cette syntaxe ressemble énormément à celle des fonctions, à la différence que les blocs de script ne prennent pas de nom : ils sont ce que l'on appelle des fonctions anonymes. Tout comme les fonctions, un bloc de script peut avoir des paramètres.

### Bloc de script incluant des paramètres

```
{
    param ([type]$param1, [type]$param2) ①
    <Liste d'instructions> ②
}
```

Ici, les paramètres ① sont à l'intérieur du bloc de script. Contrairement aux fonctions, la définition des paramètres ne peut se faire qu'à l'intérieur des deux accolades. Cependant, comme les fonctions, les blocs de script retournent des valeurs qui correspondent à la sortie des commandes de la liste d'instructions ②.

## Utilisation des blocs de script

Pour exécuter un bloc de script, il est nécessaire d'utiliser l'opérateur d'appel & :

```
PS> & { Get-Service }
```

Status	Name	DisplayName
Running	AdobeARMservice	Adobe Acrobat Update Service
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service
Stopped	AeLookupSvc	Expérience d'application
Stopped	ALG	Service de la passerelle de la couc...
Stopped	AllUserInstallA...	Agent d'installation pour tous les ...
Running	AMD External Ev...	AMD External Events Utility
Running	AMD FUEL Service	AMD FUEL Service
Stopped	AppIDSvc	Identité de l'application
Running	Appinfo	Informations d'application
Running	AudioEndpointBu...	Générateur de points de terminaison...
Running	Audiosrv	Audio Windows

Stopped	AxInstSV	Programme d'installation ActiveX (A...)
Stopped	BDESVC	Service de chiffrement de lecteur B...
Running	BFE	Moteur de filtrage de base
Running	BITS	Service de transfert intelligent en...
Running	BrokerInfrastru...	Service d'infrastructure des tâches...

Certaines cmdlets PowerShell utilisent les blocs de script comme valeurs de certains paramètres. C'est le cas des cmdlets `Where-Object` et `Foreach-Object` :

```
PS> Get-Service | Where-Object { $_.Status -eq "Stopped" }
```

Status	Name	DisplayName
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service
Stopped	AeLookupSvc	Expérience d'application
Stopped	ALG	Service de la passerelle de la couc...
Stopped	AllUserInstallA...	Agent d'installation pour tous les ...
Stopped	AppIDSvc	Identité de l'application
Stopped	AxInstSV	Programme d'installation ActiveX (A...)
Stopped	BDESVC	Service de chiffrement de lecteur B...
Stopped	Browser	Explorateur d'ordinateurs
Stopped	bthserv	Service de prise en charge Bluetooth
Stopped	CertPropSvc	Propagation du certificat
Stopped	COMSysApp	Application système COM+
Stopped	defragsvc	Optimiser les lecteurs
Stopped	DeviceInstall	Service d'installation de périphérique
Stopped	dot3svc	Configuration automatique de réseau...
Stopped	DsmSvc	Gestionnaire d'installation de péri...
Stopped	Eaphost	Protocole EAP (Extensible Authentic...
Stopped	EFS	Système de fichiers EFS (Encrypting...

```
PS> 10,20,30 | Foreach-Object -Process { $_ / 2 }
```

```
5
```

```
10
```

```
15
```

Nous pouvons observer à travers tous ces exemples la puissance des blocs de script dans PowerShell. Toutefois, nous nous sommes uniquement familiarisés avec les bases de la compréhension.



# 3

## Le pipeline

---

*Le pipeline est sûrement l'élément le plus partagé entre les différents interpréteurs de commandes. Pourtant, la façon dont il fonctionne avec PowerShell est tout à fait particulière en bien des aspects. De plus, même si cet élément est quasi transparent quant à son fonctionnement, maîtriser ses aspects est une condition sine qua non à la maîtrise de PowerShell.*

*Nous nous familiariserons donc avec le principe de fonctionnement du pipeline, puis nous évoquerons la liaison des paramètres, qui est très directement liée au pipeline. Enfin, nous examinerons les erreurs de pipeline pouvant causer l'échec d'une commande.*

### SOMMAIRE

- ▶ Principe et fonctionnement du pipeline avec PowerShell
- ▶ La liaison des paramètres
- ▶ Examiner le pipeline

## Principe et fonctionnement du pipeline

Comme dans tous les interpréteurs, il est possible d'invoquer plusieurs commandes simultanément dans la même ligne. Toutefois, cela nécessite un moyen puissant pour les mettre en articulation : le pipeline.

### Qu'est-ce qu'un pipeline ?

Un pipeline est un moyen par lequel des commandes peuvent communiquer les unes avec les autres. Le principe est qu'une [commande\\_1](#) envoie sa sortie à la [commande\\_2](#), qui elle-même, après avoir traité le flux entrant, envoie sa propre sortie à la [commande\\_3](#), et ainsi de suite. Cette communication se fait via l'opérateur `|`. Donc, il peut y avoir autant d'opérateurs `|` que nécessaire.

#### Chaîne de commandes sollicitant le pipeline

```
Commande_1 | Commande_2 | Commande_3 | Commande_4
```

Le principe du pipeline est, par conséquent, de réaliser une chaîne de commandes. Il en résulte un flux d'objets à travers l'ensemble des commandes visées.

#### À SAVOIR Ordre dans lequel s'exécutent les commandes

En PowerShell, les commandes sont exécutées de gauche à droite. En outre, l'ensemble de la chaîne est considéré comme une unité d'exécution.

L'architecture des cmdlets est conçue de telle sorte qu'elles soient communicatives. Ainsi, une cmdlet qui a pour verbe [Get](#) est destinée à envoyer des objets à d'autres cmdlets d'action, ayant comme verbes [Stop](#), [Set](#), [Start](#), [New](#), [Suspend](#), [Rename](#)... Par exemple, prenons la liste des commandes liées à la gestion des services :

```
PS> Get-Command -Noun Service
```

CommandType	Name
Cmdlet	Get-Service
Cmdlet	New-Service
Cmdlet	Restart-Service
Cmdlet	Resume-Service
Cmdlet	Set-Service
Cmdlet	Start-Service
Cmdlet	Stop-Service
Cmdlet	Suspend-Service

En observant la sortie, on voit qu'elles partagent toutes le même nom, mais les verbes diffèrent. Cela signifie qu'elles ont un mode d'action spécifique, mais surtout qu'elles peuvent communiquer les unes avec les autres grâce au pipeline. L'exemple suivant cherche puis démarre le service WinRM :

```
PS> Get-Service -Name WinRM | Start-Service
```

Voici un autre exemple pour mieux saisir l'utilisation du pipeline. Essayons d'obtenir les processus consommant plus de 15 Mo de mémoire physique :

```
PS> Get-Process | Where-Object {$_.WorkingSet -gt 15000000}
```

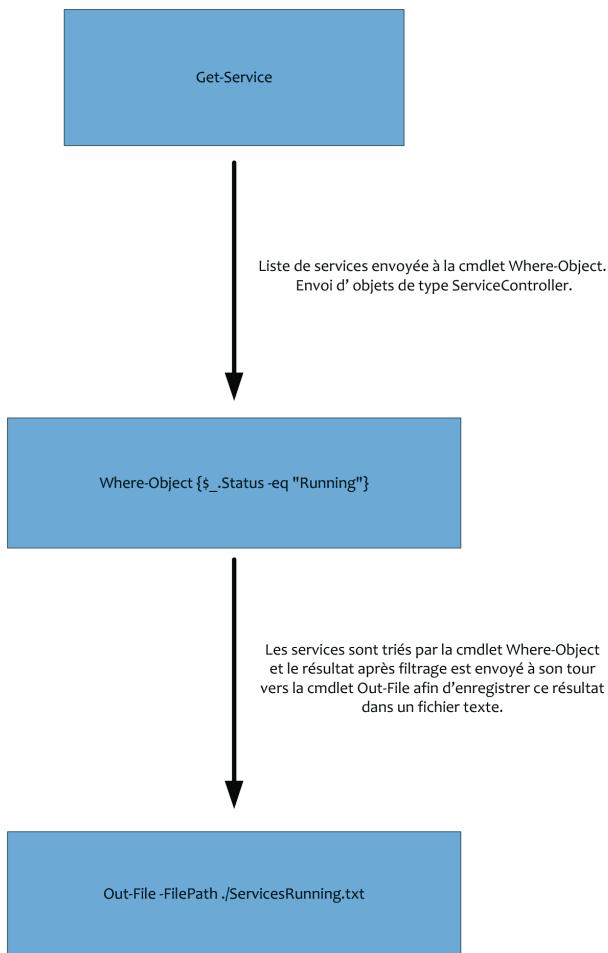
Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
301	29	57936	74932	239	44,51	1856	AcroRd32
284	18	7776	15656	97	1,28	6444	AcroRd32
774	94	80728	22772	846	24,55	4924	CCC
181	21	19648	34640	200		2076	dwm
2111	175	43912	87028	467	248,88	5480	explorer
498	74	137136	169892	493	307,84	4960	firefox
482	82	96072	81288	249		4192	MsMpEng
212	30	30712	28692	620	0,75	1316	pceef4
338	27	122360	132400	613	10,25	6736	powershell
576	48	28904	23040	581		3320	SearchIndexer
518	77	1537840	1271688	1843	179,48	3472	soffice.bin
738	36	9532	16056	102		296	svchost
955	39	77796	74188	159		564	svchost
938	36	19544	20580	113		904	svchost
1814	63	165064	121324	395		984	svchost
490	46	23048	20364	393		1396	svchost
293	46	11380	17124	548	3,49	6304	taskhostex
437	14	12272	27396	126	23,56	3272	VirtualBox
...							

Le résultat a été volontairement tronqué ici, mais nous avons obtenu un résultat probant en une simple ligne de commande. L'utilisation du pipeline conduit donc à un gain de temps incontestable.

## Le pipeline un peu plus dans le détail

Maintenant que nous avons abordé cette notion importante, voici un schéma plus approfondi montrant l'utilisation d'un pipeline de commandes et le flux d'objets qui lui est lié.

**Figure 3-1**  
Schéma de fonctionnement  
du pipeline avec PowerShell



Via ce pipeline, nous obtenons d'abord tous les services présents dans le système avec la cmdlet `Get-Service`. Ensuite, l'ensemble des services collectés sous forme de liste d'objets est envoyé à la cmdlet `Where-Object`, qui ne sélectionne que les services dont la propriété `Status` a la valeur `Running`. Après filtrage, le résultat final est reçu et traité par la cmdlet `Out-File`, qui l'enregistre dans un fichier texte. En ligne de commande, cela donne :

```
PS> get-service | where-object {$_.Status -eq "Running"} | out-file -FilePath ./ServicesRunning.txt
```

Finalement, le pipeline est la langue dans laquelle s'exprime l'ensemble des cmdlets dans PowerShell, ou le socle fondamental de communication.

## Le processus de liaison des paramètres

La liaison des paramètres est le processus par lequel PowerShell redirige les objets envoyés à travers le pipeline depuis une cmdlet vers une autre. En effet, l'objet envoyé depuis la cmdlet **A** est associé à l'un des paramètres de la cmdlet **B** qui reçoit l'objet en question. La prise en compte d'entrées par les cmdlets est définie lors du développement de ces dernières ; PowerShell, qui analyse chacune de ces commandes de manière très précise, sait si elles acceptent ou non les entrées via leurs paramètres.

### Mode de fonctionnement

Pour qu'une cmdlet accepte des entrées, elle doit remplir les critères suivants.

- Au moins un de ses paramètres doit accepter une entrée en provenance du pipeline.
- Si un paramètre accepte une entrée en provenance du pipeline, il doit explicitement accepter le type d'objet envoyé ou alors un type d'objet pouvant être converti.
- Le paramètre ne doit pas être utilisé au préalable dans la cmdlet.

Pour savoir si une cmdlet accepte ou pas des objets en provenance du pipeline, il faut utiliser la commande `Get-Help`. Et là, nous avons le choix entre les paramètres `full` et `parameter`. La ligne de commande suivante utilise `parameter` avec l'argument `*` pour étendre la recherche à tous les paramètres :

```
PS> Get-Help Get-Service -Parameter *
```

-InputObject <ServiceController[]>  
Spécifie les objets ServiceController représentant les services à récupérer.  
Entrez une variable contenant les objets, ou tapez une commande ou une  
expression qui obtient ces objets. Vous pouvez également diriger un objet  
service vers Get-Service.

Obligatoire ?	false
Position ?	named
Valeur par défaut	

① Accepter l'entrée de pipeline ? true (ByValue)  
Accepter les caractères génériques ? false

-Name <String[]>  
Spécifie les noms des services à récupérer. Les caractères génériques sont  
autorisés. Par défaut, Get-Service obtient tous les services présents sur  
l'ordinateur.

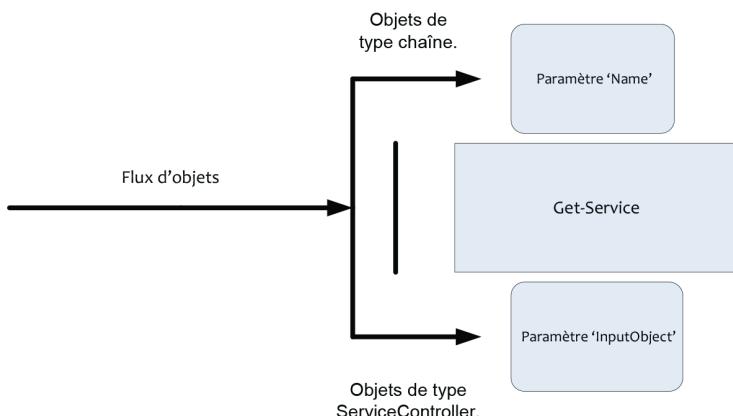
Obligatoire ?	false
Position ?	1

	Valeur par défaut	All services
①	Accepter l'entrée de pipeline ?	true (ByPropertyName, ByValue)
	Accepter les caractères génériques ?	true

La sortie est longue, nous ne retiendrons donc que les paramètres qui nous intéressent, c'est-à-dire `-InputObject` et `-Name`. Ici, l'aide indique qu'ils acceptent l'entrée de pipeline : la ligne ① a une valeur `true` dans les deux cas. Le résultat est que cette cmdlet peut accepter plusieurs types d'objets à travers le pipeline. Un schéma sera peut-être plus explicite.

**Figure 3–2**

La liaison de paramètres avec PowerShell



Si, au cours de l'exécution, la liaison de paramètres ne peut se réaliser car les conditions requises ne sont pas remplies, PowerShell invite à fournir les valeurs de paramètres manquantes.

## Liaison de paramètres par valeur

La liaison par valeur implique qu'un paramètre puisse accepter des objets qui ont le même type .NET que sa valeur. Si l'objet envoyé peut être converti en ce même type, alors il sera accepté. Par exemple, le paramètre `-Name` de la cmdlet `Get-Service` accepte, comme on l'a vu, des objets de type chaîne ou pouvant être convertis en ce type. Voici des exemples d'utilisation de la liaison de paramètres par valeur :

```
PS> 'winrm' | get-service
```

Status	Name	DisplayName
Stopped	winrm	Gestion à distance de Windows (Gest...

```
PS> $winrm = get-service -name winrm
PS> $winrm | get-service
Status      Name            DisplayName
-----      --  -----
Stopped     winrm          Gestion à distance de Windows (Gest...
```

## Liaison de paramètres par nom de propriété

La liaison par nom de propriété implique qu'un paramètre puisse accepter des objets uniquement lorsqu'une propriété de l'objet redirigé porte le même nom que le paramètre. Toujours sur l'exemple de `-Name` de la cmdlet `Get-Service`, on observe via l'aide que ce paramètre accepte aussi des objets ayant une propriété portant le même nom. Dans l'exemple qui va suivre, nous allons importer un fichier `.csv` contenant une liste de trois services et rediriger la sortie vers la cmdlet `Get-Service`. Tout d'abord, commençons par regarder le contenu du fichier en mode console :

```
PS> Get-Content -Path .\services.csv
Name, DisplayName
WinRM, Gestion à distance de Windows
WSearch, Windows Search
TermService, Services Bureau à distance
```

Le fichier est un simple `.csv`. Maintenant, utilisons la cmdlet `Import-Csv` afin d'obtenir des objets avec propriétés :

```
PS> Import-Csv -Path .\services.csv
Name            DisplayName
----            -----
WinRM          Gestion à distance de Windows
WSearch        Windows Search
TermService    Services Bureau à distance
```

En important ce fichier, nous obtenons en sortie trois objets (chaque ligne représentant un objet). Chacun a deux propriétés : `Name` et `DisplayName`. Envoyons maintenant ces objets à la cmdlet `Get-Service` pour vérifier leur statut :

```
PS> Import-Csv -Path .\services.csv | Get-Service
```

Status	Name	DisplayName
Stopped	WinRM	Gestion à distance de Windows (Gest...
Running	WSearch	Windows Search
Stopped	TermService	Services Bureau à distance

Et voilà ! La liaison de paramètres par nom de propriété s'est opérée correctement. PowerShell a, pour chaque objet, lié la propriété `Name` à la cmdlet `Get-Service` via son paramètre `-Name`.

## Analyser les erreurs de pipeline

Lorsque l'exécution d'une commande ne se déroule pas correctement, il faut essayer de résoudre les problèmes qui se posent à nous. PowerShell donne la possibilité à l'utilisateur d'analyser un certain nombre de types d'erreurs, notamment celles en rapport avec le pipeline.

### Examiner les erreurs de pipeline avec Trace-Command

La cmdlet `Trace-Command` est là pour nous aider dans l'analyse de ce type d'erreur. Entre autres, elle trace le processus de liaison des paramètres de PowerShell, directement lié au fonctionnement du pipeline. Tracer un programme est une chose que les programmeurs font depuis longtemps, pour corriger les problèmes et optimiser le code. Même si ce type de commande s'adresse en premier lieu à des programmeurs PowerShell, un administrateur peut très bien l'utiliser, car l'abstraction fournie est importante. L'exemple suivant illustre l'envoi d'une chaîne de caractères (représentant un objet) vers la cmdlet `Get-Process` :

```
PS> "notepad" | get-process
```

get-process : L'objet d'entrée ne peut être lié à aucun paramètre de la commande, soit parce que cette commande n'accepte pas l'entrée de pipeline, soit parce que l'entrée et ses propriétés ne correspondent à aucun des paramètres qui acceptent l'entrée de pipeline.  
Au caractère Ligne:1 : 13  
+ "notepad" | get-process  
+ ~~~~~~  
+ CategoryInfo: InvalidArgument : (notepad:String) [Get-Process], ParameterBindingException  
+ FullyQualifiedErrorId: InputObjectNotBound,Microsoft.PowerShell.Commands.GetProcessCommand

En lisant attentivement la sortie, on observe que PowerShell a signalé une erreur concernant la liaison de paramètres. Si l'utilisateur connaît suffisamment bien la commande en question, il peut par lui-même essayer de réécrire la ligne de commande de sorte qu'elle soit complètement cohérente. Sinon, il peut utiliser la cmdlet `Trace-Command`. Il faut d'abord déterminer ce que nous voulons analyser : la liaison de paramètres, ainsi que la cmdlet `Get-Process`. Pour cela, nous utiliserons le paramètre `-Name`, qui listera les composants à tracer. Ensuite, nous aurons besoin du paramètre `-Expression` pour spécifier la ligne de commande à tracer. Enfin, pour lire la sortie à l'écran, le paramètre `-Pshost` est nécessaire. La sortie est très verbeuse, donc nous extrairons les éléments les plus significatifs :

```
PS> trace-command -name parameterbinding,cmdlet -expression {'notepad' | Get-Process} -pshost

DÉBOUER :ParameterBinding Information: 0 :      BIND NAMED cmd line args [Get-Process]
DÉBOUER :ParameterBinding Information: 0 :      BIND POSITIONAL cmd line args [Get-Process]
DÉBOUER :ParameterBinding Information: 0 :      MANDATORY PARAMETER CHECK on cmdlet [Get-Process]
DÉBOUER :ParameterBinding Information: 0 :      CALLING BeginProcessing
DÉBOUER :ParameterBinding Information: 0 :      BIND PIPELINE object to parameters: [Get-Process]
DÉBOUER :ParameterBinding Information: 0 :      PIPELINE object TYPE =
[System.String]
DÉBOUER :ParameterBinding Information: 0 :      RESTORING pipeline parameter's original values
DÉBOUER :ParameterBinding Information: 0 :      Parameter [InputObject] PIPELINE INPUT ValueFromPipeline NO COERCION
DÉBOUER :ParameterBinding Information: 0 :      BIND arg [notepad] to parameter [InputObject]
DÉBOUER :ParameterBinding Information: 0 :      Binding collection parameter InputObject: argument type [String], parameter type [System.Diagnostics.Process[]], collection type Array, element type [System.Diagnostics.Process], no coerceElementType
DÉBOUER :ParameterBinding Information: 0 :      Creating array with element type [System.Diagnostics.Process] and 1 elements
DÉBOUER :ParameterBinding Information: 0 :      Argument type String is not IList, treating this as scalar
DÉBOUER :ParameterBinding Information: 0 :      BIND arg [notepad] to param [InputObject] SKIPPED
```

Le premier extrait ci-dessus indique qu'une valeur de type chaîne est liée au paramètre `-InputObject` de la cmdlet `Get-Process`. Ce paramètre acceptant uniquement des objets de type `Process`, la liaison de paramètres ne peut pas se faire. Analysons le deuxième extrait :

```
DÉBOGUE :ParameterBinding Information: 0 : Parameter [Name] PIPELINE INPUT
ValueFromPipelineByPropertyName NO
COERCION
DÉBOGUE :ParameterBinding Information: 0 : Parameter [ComputerName]
PIPELINE INPUT
ValueFromPipelineByPropertyName NO COERCION
DÉBOGUE :ParameterBinding Information: 0 : Parameter [Id] PIPELINE INPUT
ValueFromPipelineByPropertyName NO
COERCION
DÉBOGUE :ParameterBinding Information: 0 : Parameter [ComputerName]
PIPELINE INPUT
ValueFromPipelineByPropertyName NO COERCION
DÉBOGUE :ParameterBinding Information: 0 : Parameter [InputObject] PIPELINE
INPUT ValueFromPipeline WITH COERCION
DÉBOGUE :ParameterBinding Information: 0 : BIND arg [notepad] to parameter
[InputObject]
DÉBOGUE :ParameterBinding Information: 0 : COERCE arg to
[System.Diagnostics.Process[]]
DÉBOGUE :ParameterBinding Information: 0 : Trying to convert argument value
from System.Management.Automation.PSObject to System.Diagnostics.Process[]
DÉBOGUE :ParameterBinding Information: 0 : ENCODING arg into collection
DÉBOGUE :ParameterBinding Information: 0 : Binding collection parameter
InputObject: argument type
[PSObject], parameter type [System.Diagnostics.Process[]], collection type
Array, element type
[System.Diagnostics.Process], coerceElementType
DÉBOGUE : ParameterBinding Information: 0 : Creating array with element type
[System.Diagnostics.Process]
and 1 elements
DÉBOGUE : ParameterBinding Information: 0 : Argument type PSObject is not
IList, treating this as scalar
DÉBOGUE : ParameterBinding Information: 0 : COERCE arg to
[System.Diagnostics.Process]
DÉBOGUE : ParameterBinding Information: 0 : Trying to convert argument value
from System.Management.Automation.PSObject to System.Diagnostics.Process
DÉBOGUE : ParameterBinding Information: 0 : CONVERT arg type to param type
using LanguagePrimitives.ConvertTo
DÉBOGUE : ParameterBinding Information: 0 : ERROR: ERROR: COERCE FAILED: arg
[notepad] could not be converted to the parameter type
[System.Diagnostics.Process]
DÉBOGUE :ParameterBinding Information: 0 : Parameter [Name] PIPELINE INPUT
ValueFromPipelineByPropertyName WITH
COERCION
DÉBOGUE :ParameterBinding Information: 0 : Parameter [ComputerName]
PIPELINE INPUT
ValueFromPipelineByPropertyName WITH COERCION
DÉBOGUE :ParameterBinding Information: 0 : Parameter [Id] PIPELINE INPUT
ValueFromPipelineByPropertyName WITH
COERCION
```

```
DÉBOGUER :ParameterBinding Information: 0 :      Parameter [ComputerName]
PIPELINE INPUT
ValueFromPipelineByPropertyName WITH COERCION
```

Cet extrait met en évidence que la liaison de paramètres par nom de propriété n'a pas lieu d'être. En effet, l'objet envoyé n'a pas de propriétés. On voit aussi que l'objet n'a pas pu être converti.

```
DÉBOGUER : ParameterBinding Information: 0 :      MANDATORY PARAMETER CHECK on
cmdlet [out-lineoutput]
DÉBOGUER : ParameterBinding Information: 0 :      BIND NAMED cmd line args
[Format-Default]
DÉBOGUER : ParameterBinding Information: 0 :      BIND POSITIONAL cmd line args
[Format-Default]
DÉBOGUER : ParameterBinding Information: 0 :      MANDATORY PARAMETER CHECK on
cmdlet [Format-Default]
DÉBOGUER : ParameterBinding Information: 0 :      CALLING BeginProcessing
DÉBOGUER : ParameterBinding Information: 0 :      BIND PIPELINE object to
parameters: [Format-Default]
DÉBOGUER : ParameterBinding Information: 0 :      PIPELINE object TYPE =
[System.Management.Automation.ErrorRecord]
DÉBOGUER : ParameterBinding Information: 0 :      RESTORING pipeline parameter's
original values
DÉBOGUER : ParameterBinding Information: 0 :      Parameter [InputObject]
PIPELINE INPUT ValueFromPipeline NO COERCION
DÉBOGUER : ParameterBinding Information: 0 :      BIND arg [L'objet d'entrée ne
peut être lié à aucun paramètre de la commande, soit parce que cette commande
n'accepte pas l'entrée de pipeline, soit parce que l'entrée et ses propriétés
ne correspondent à aucun des paramètres qui acceptent l'entrée de pipeline.] to
parameter [InputObject]
DÉBOGUER : ParameterBinding Information: 0 :      BIND arg [L'objet d'entrée ne
peut être lié à aucun paramètre de la commande, soit parce que cette commande
n'accepte pas l'entrée de pipeline, soit parce que l'entrée et ses propriétés ne
correspondent à aucun des paramètres qui acceptent l'entrée de pipeline.] to
param [InputObject]SUCCESSFUL
```

Le troisième extrait ci-dessus met en évidence que « L'objet d'entrée ne peut être lié à aucun paramètre de la commande... », ce qui signifie clairement que l'objet redirigé ou son type ne correspond à aucun type d'objet que certains paramètres acceptent. Cette information est cruciale, car nous connaissons à présent la nature du problème. Nous avons manifestement utilisé cette cmdlet de la mauvaise façon. Après avoir analysé ces extraits et lu la documentation, on s'aperçoit qu'il n'était pas nécessaire d'utiliser cette méthode, mais plutôt celle qui suit :

```
PS> Get-Process -Name notepad
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
84	7	1280	5428	93	0,12	5172	notepad

Le paramètre `-Name` de la cmdlet `Get-Process` accepte comme valeur une chaîne de caractères, sans passer par le pipeline, et ce, quelles que soient les modalités. Nous avons donc réussi à déboguer notre ligne de commande en passant par `Trace-Command`, qui nous a fourni les informations nécessaires à la correction d'erreurs de pipeline. L'exemple que nous avons pris est simple, mais ce qui compte le plus ici est de comprendre comment l'analyse des erreurs de pipeline peut se faire avec PowerShell.

# 4

## Manipuler les objets avec PowerShell

---

*La particularité ultime de PowerShell est qu'il est orienté (ou basé) objet, et non pas texte comme on pourrait le penser de prime abord. La différence est fondamentale, car cette orientation peut parfois dérouter certains utilisateurs, par exemple ceux venant du monde Unix/Linux.*

*Dans ce chapitre, nous clarifierons ce qu'implique ce concept. Puis nous apprendrons à maîtriser, trier, grouper, comparer, filtrer et créer des objets avec PowerShell.*

### SOMMAIRE

- ▶ PowerShell est purement orienté objet
- ▶ Maîtriser les flux : Select-Object
- ▶ Trier : Sort-Object
- ▶ Grouper : Group-Object
- ▶ Comparer : Compare-Object
- ▶ Filtrer : Where-Object
- ▶ Effectuer des actions sur chaque objet : Foreach-Object
- ▶ Créer : New-Object

## PowerShell est purement orienté objet

La base de développement de PowerShell est le framework .NET, ce qui oriente cette technologie vers le mode objet, même si nous ne voyons à l'écran que du texte. Cette information est importante, car il est essentiel de comprendre le contexte dans lequel nous évoluons.

### Qu'est-ce qu'un objet ?

Un objet est un ensemble de données définissant une entité. À ce titre, on peut considérer un objet comme une entité sur laquelle on peut agir d'une certaine façon et dans un contexte qui lui est propre. La structure d'un objet est définie en trois parties :

- le type de l'objet ;
- ses méthodes ;
- ses propriétés.

Le type d'un objet représente sa nature. Les méthodes d'un objet servent à le manipuler (le modifier, changer son état à un moment donné). Les propriétés d'un objet donnent des informations sur son état. Pour prendre un exemple concret, le tableau suivant (non exhaustif) illustre les différents types de données d'un objet représentant un service Windows.

Tableau 4–1 Structure d'un objet représentant un service

Membre	Type de donnée	Description
Nom du service	Propriété	Obtient le nom qui identifie le service.
Nom de l'ordinateur	Propriété	Obtient le nom de l'ordinateur sur lequel le service réside.
Type de service	Propriété	Obtient le type de service référencé par l'objet.
Statut	Propriété	Donne l'état du service.
Peut être arrêté ?	Propriété	Indique si le service peut être arrêté après avoir démarré.
Démarrer	Méthode	Démarre le service.
Arrêter	Méthode	Arrête le service.
Interrompre	Méthode	Interrompt le fonctionnement d'un service.
Continuer	Méthode	Permet de reprendre le fonctionnement d'un service après qu'il a été suspendu.

Un objet est donc un moyen par lequel il est possible d'agir sur un composant, ici un composant Windows. Par conséquent, manipuler un objet peut être très dangereux.

## Utiliser un objet

Maintenant que nous savons ce qu'est un objet, essayons d'en utiliser un de manière simple et claire. Prenons l'exemple d'une chaîne de caractères que nous encapsulons dans une variable, de telle façon que l'objet constituant la chaîne soit utilisable via la variable :

```
PS> $var = "PowerShell"
```

Ici, nous avons créé une variable nommée `$var` et lui avons affecté la valeur `PowerShell`. Comment la manipuler ? Quels sont les membres que nous avons à notre disposition ? Une cmdlet peut nous aider à répondre : `Get-Member`, qui renvoie les propriétés et méthodes des objets. Interrogeons-la :

```
PS> $var | Get-Member
```

TypeName : System.String		
Name	MemberType	Definition
Clone	Method	System.Object Clone(), Syste
CompareTo	Method	int CompareTo(System.Object
Contains	Method	bool Contains(string value)
CopyTo	Method	void CopyTo(int sourceIndex,
EndsWith	Method	bool EndsWith(string value),
Equals	Method	bool Equals(System.Object ob
GetEnumerator	Method	System.CharEnumerator GetEnu
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCode GetTypeCode(
IndexOf	Method	int IndexOf(char value), int
IndexOfAny	Method	int IndexOfAny(char[] anyOf)
Insert	Method	string Insert(int startIndex
IsNormalized	Method	bool IsNormalized(), bool Is
LastIndexOf	Method	int LastIndexOf(char value),
LastIndexOfAny	Method	int LastIndexOfAny(char[] an
Normalize	Method	string Normalize(), string N
PadLeft	Method	string PadLeft(int totalWidt
PadRight	Method	string PadRight(int totalWid
Remove	Method	string Remove(int startIndex
Replace	Method	string Replace(char oldChar,
Split	Method	string[] Split(Params char[]
StartsWith	Method	bool StartsWith(string value
Substring	Method	string Substring(int startIn
ToBoolean	Method	bool IConvertible.ToBoolean(
ToByte	Method	byte IConvertible.ToByte(Sys

ToChar	Method	char IConvertible.ToChar(Sys
ToCharArray	Method	char[] ToCharArray(), char[]
ToDateTime	Method	datetime IConvertible.ToDateTime
ToDecimal	Method	decimal IConvertible.ToDecimal
ToDouble	Method	double IConvertible.ToDouble
ToInt16	Method	int16 IConvertible.ToInt16(Sys
ToInt32	Method	int IConvertible.ToInt32(Sys
ToInt64	Method	long IConvertible.ToInt64(Sy
ToLower	Method	string ToLower(), string ToL
ToLowerInvariant	Method	string ToLowerInvariant()
ToSByte	Method	sbyte IConvertible.ToSByte(S
ToSingle	Method	float IConvertible.ToSingle(
ToString	Method	string ToString(), string To
ToType	Method	System.Object IConvertible.T
ToUInt16	Method	uint16 IConvertible.ToUInt16
ToUInt32	Method	uint32 IConvertible.ToUInt32
ToUInt64	Method	uint64 IConvertible.ToUInt64
ToUpper	Method	string ToUpper(), string ToU
ToUpperInvariant	Method	string ToUpperInvariant()
Trim	Method	string Trim(Params char[] tr
TrimEnd	Method	string TrimEnd(Params char[]
TrimStart	Method	string TrimStart(Params char
Chars	ParameterizedProperty	char Chars(int index) {get;}
Length	Property	int Length {get;}

La liste des membres comporte presque exclusivement des méthodes. Si nous voulons convertir notre chaîne de caractères en majuscules, il y a la méthode `ToUpper` :

```
PS> $var.ToUpper()
POWERSHELL
```

Pour convertir tous les caractères en minuscules, il y a la méthode `ToLower` :

```
PS> $var.ToLower()
powershell
```

Pour remplacer une chaîne spécifique par une autre, il y a la méthode `Replace` :

```
PS> $var.Replace("Shell", "Devel")
PowerDevel
```

Pour obtenir le nombre de caractères de notre chaîne, il y a la propriété `Length` :

```
PS> $var.Length
10
```

Manipuler un objet avec PowerShell demande donc un certain temps avant de le maîtriser totalement, car chaque membre a ses particularités. Les sections suivantes seront dédiées à l'utilisation des objets de manière plus poussée.

## Maîtriser les flux d'objets

Lors des manipulations, on peut être intéressé par l'objet dans son intégralité, mais aussi par ses propriétés, autrement dit certaines informations. Donc, nous pouvons avoir une approche sélective vis-à-vis d'un objet en particulier. Cette capacité à sélectionner certaines informations plutôt que d'autres nous est offerte par la cmdlet `Select-Object`.

### Sélectionner parmi un jeu d'objets

Pour sélectionner un nombre particulier d'objets parmi un groupe, il y a les paramètres :

- `-First` pour sélectionner au début d'un tableau d'objets ;
- `-Last` pour sélectionner depuis la fin d'un tableau d'objets.

Essayons, par exemple, d'obtenir les 10 processus qui consomment le plus de temps processeur :

```
PS> Get-Process | Sort-Object -Property CPU -Descending | Select-Object -First  
10
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
306	29	21696	31928	211	4 111,22	356	FlashPlayerPlugin
540	17	10968	14300	72	408,16	4128	audiogd
768	103	86036	94800	1466	331,22	3276	VirtualBox
117	10	1964	5980	73	260,44	1692	YCMMirage
556	79	182904	220004	581	227,34	1872	firefox
254	32	131056	163416	332	150,73	5616	AcroRd32
213	20	11888	21436	153	133,65	3176	plugin-container
625	65	22780	47208	270	116,28	644	winamp
2137	177	46440	87612	479	45,82	3676	explorer
393	64	55892	102324	374	39,28	2812	soffice.bin

Ne faites pas attention à la cmdlet `Sort-Object` que nous étudierons dans la prochaine section. Maintenant, listons les 10 processus consommant le moins de mémoire physique :

```
PS> Get-Process | Sort-Object -Property WS -Descending | Select-Object -Last 10
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
96	8	1380	4436	22		4036	svchost
62	7	1004	4288	62	0,09	40	winampa
66	6	936	4020	28		2788	unsecapp
80	8	1012	3960	41		576	wininit
76	7	1092	3852	44		1640	armsvc
52	6	1092	3656	69	0,06	2720	soffice
99	6	760	3184	23		880	atiesrxx
43	5	864	3108	18		1784	Service
36	2	276	956	4		284	smss
0	0	0	20	0		0	Idle

Les objets que nous avons sélectionnés dans ces exemples sont « entiers », mais qu'en est-il si nous voulons extraire uniquement certaines propriétés ?

## Sélectionner certaines propriétés d'objets

Pour choisir certaines propriétés par rapport à un ensemble, il y a le paramètre `-Property` qui sert à spécifier les propriétés à sélectionner.

Toujours sur la base de nos précédents exemples, prenons celui mettant en évidence les processus qui consomment le plus de temps processeur, mais en sélectionnant uniquement les propriétés `ProcessName` et `CPU` :

```
PS> Get-Process | Sort-Object -Property CPU -Descending | Select-Object -Property ProcessName,CPU -First 10
```

ProcessName	CPU
CCC	15,3972987
explorer	11,4192732
soffice.bin	7,7376496
YontooDesktop	4,4928288
powershell	3,5256226
YCMMirage	2,4804159
MOM	1,3416086
LiveComm	0,9204059
pcee4	0,7644049
RuntimeBroker	0,3744024

On peut voir que la sortie montre uniquement les propriétés que nous avons spécifiées, de sorte que nous puissions travailler plus efficacement.

## Extraire le contenu d'une propriété

Certaines propriétés ont des valeurs représentant des collections de données. Le contenu de ces propriétés n'est visible que partiellement, et ce, pour des raisons liées à la façon dont PowerShell formate les données de sortie. Dans l'exemple qui va suivre, nous nous intéresserons au processus « `powershell` » et aux modules (à ne pas confondre avec les modules PowerShell) qu'il charge. Observons d'abord le comportement classique de PowerShell lorsque nous voulons lister des propriétés ayant comme valeur des collections de données ou d'objets :

```
PS> Get-Process -Name powershell | Select-Object -Property ProcessName, Modules

ProcessName Modules
-----
powershell {System.Diagnostics.ProcessModule (powershell.exe), System...
```

En analysant la sortie, on distingue l'affichage partiel du contenu de la propriété `Modules`. Nous ne pouvons donc pas connaître de manière détaillée la liste des modules chargés au démarrage du processus `powershell`. Pour modifier l'affichage, il y a le paramètre `-ExpandProperty` qui développe le contenu d'une propriété d'un objet.

Pour obtenir notre liste de modules, voici comment procéder :

```
PS> Get-Process -Name powershell | Select-Object -ExpandProperty Modules | 
Select-Object -Property ModuleName

ModuleName
-----
powershell.exe
ntdll.dll
KERNEL32.DLL
KERNELBASE.dll
ADVAPI32.dll
msvcrt.dll
ATL.dll
ole32.dll
OLEAUT32.dll
SHLWAPI.dll
USER32.dll
SHELL32.dll
mscoree.dll
sechost.dll
RPCRT4.dll
GDI32.dll
```

```
combase.dll
IMM32.DLL
MSCTF.dll
...
```

La cmdlet `Select-Object` a été utilisée deux fois dans le pipeline car, la propriété `Modules` ayant été développée, l'affichage aurait montré une collection de modules avec plusieurs propriétés. Par conséquent, il a fallu exclure certaines propriétés afin de nous concentrer sur nos objectifs. Le résultat dont l'affichage a été tronqué, car très long, nous montre une liste de modules associés au processus `powershell`. L'objectif a donc été atteint via ce paramètre. La cmdlet `Select-Object` contient encore d'autres paramètres, mais nous nous arrêterons ici concernant les modes de sélection liés aux objets.

#### **NOTE Concernant PowerShell version 3**

Depuis la version 3 de PowerShell, la cmdlet `Select-Object` est dotée d'une fonctionnalité d'optimisation. En effet, lorsque celle-ci est utilisée avec les paramètres `-First` et `-Index`, le nombre d'objets créés est égal au nombre d'objets sélectionnés. Il en résulte une optimisation réelle en termes de performance.

## Trier les objets

Nous avons étudié dans la section précédente différents modes de sélection des objets avec PowerShell. Ici, nous apprendrons à trier les objets en s'appuyant sur les valeurs de propriétés via la cmdlet `Sort-Object`.

### Trier les objets en fonction des valeurs de propriétés

En PowerShell, il est possible de trier des objets sur la base des valeurs de leurs propriétés, grâce au paramètre `-Property` :

```
PS> Get-Process | Sort-Object -Property WS
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
0	0	0	20	0		0	Idle
36	2	276	780	4		284	smss
43	5	852	3100	18		1924	Service
99	6	808	3168	23		880	aticesrxx
52	6	1092	3672	69	0,08	5864	soffice

75	7	1088	3848	44		1648	armsvc
81	8	968	3888	41		576	wininit
67	6	940	4024	28		2544	unsecapp
62	7	1012	4284	62	0,02	1316	winampa
94	8	1308	4348	22		3400	svchost
136	7	1156	4636	46		32	winlogon
446	14	1984	4880	49		480	csrss
...							

Ici, nous avons listé les processus consommant le moins de mémoire physique. Pour cela, nous avons effectué le tri sur la propriété [WS\(WorkingSet\)](#). L'ordre est croissant par défaut (depuis les origines de PowerShell) et le tri peut être fait à partir de plusieurs propriétés.

## Trier les objets dans l'ordre décroissant

Pour effectuer un tri dans l'ordre décroissant, il faut utiliser le paramètre [-Descending](#). Listons les processus les plus consommateurs de temps processeur :

```
PS> Get-Process | Sort-Object -Property CPU -Descending
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
292	26	24780	35284	208	1 037,20	3728	FlashPlayer
759	102	85408	94664	1471	491,37	3152	VirtualBox
403	14	10452	13912	69	378,51	4104	audiogd
495	74	177240	208172	485	339,83	3148	firefox
117	10	1904	5964	73	308,63	2468	YCMMirage
207	20	10984	18712	150	285,97	5232	plugin-cont
351	63	1191592	304192	1457	74,46	1528	soffice.bin
1992	160	40880	79152	451	37,11	2596	explorer
304	34	108420	127368	309	35,91	2928	AcroRd32
956	14	4576	12612	104	35,40	2476	VBoxSVC
789	95	76932	20688	847	28,25	2608	CCC
417	14	11632	25900	125	13,32	3160	VirtualBox
...							

## Trier les objets en éliminant les doublons

Il est possible, lorsqu'une collection de données contient certaines valeurs identiques, de ne lister qu'une seule occurrence de chaque valeur. Pour ce faire, la cmdlet [Sort-Object](#) propose le paramètre [-Unique](#) :

```
PS> 1,2,2,3,3,3,4,4,4,4,5,5,5,5 | Sort-Object -Unique  
1  
2  
3  
4  
5
```

Dans cet exemple, une série de chiffres contenant des entiers identiques est envoyée à la commande `Sort-Object` qui, via son paramètre `-Unique`, ne retourne qu'une seule occurrence de chaque chiffre.

`Sort-Object` est donc un moyen très original pour trier des masses de données ; original, car contrairement au monde Unix/Linux, l'opération de tri s'effectue sur des objets et non pas sur des flux de textes.

## Grouper les objets

Lorsque les objets sont manipulés, il est parfois nécessaire de regrouper certains d'entre eux, par exemple pour des raisons statistiques. En PowerShell, la cmdlet `Group-Object` permet justement de regrouper les objets contenant une même valeur.

### Grouper les objets en s'appuyant sur une valeur de propriété

Il est possible d'organiser en groupes des objets en fonction d'une valeur de propriété spécifique via le paramètre `-Property`. L'exemple suivant met en évidence deux groupes de services. Le premier représente les services actuellement en cours d'exécution et le second ceux qui ne le sont pas :

```
PS> get-service | group-object -property status  
Count Name      Group  
-----  
    77 Running {AdobeARMservice, AMD External Events Utility, AMD...}  
    88 Stopped {AdobeFlashPlayerUpdateSvc, AeLookupSvc, ALG...}
```

Si nous ne sommes pas intéressés par les membres de groupes, nous pouvons utiliser le paramètre `-noelement` :

```
PS> get-service | group-object -property status -noelement  
Count Name  
-----  
    76 Running  
    89 Stopped
```

Le paramètre `-AsHashTable` fournit les groupes sous la forme d'une table de hachage (ce qui peut être idéal dans un script) :

```
PS> get-service | group-object -property status -AsHashTable
```

Name	Value
Running	{AdobeARMservice, AMD External Events Ut...}
Stopped	{AdobeFlashPlayerUpdateSvc, AeLookupSvc, ...}

**NOTE Sur les valeurs rentrées par la cmdlet Group-Object**

Ne renvoie pas des types d'objets identiques à ceux qui lui ont été redirigés.

## Comparer les objets

Pour comparer les objets, c'est-à-dire les mettre en perspective afin de trouver les différences qu'il y a entre eux, PowerShell propose une cmdlet : `Compare-Object`.

### Comparer sur la base de collections d'objets

Cette commande est très utile notamment lorsque nous voulons faire des rapports. Par exemple, sur un serveur à une heure donnée, on collecte l'ensemble des processus en cours d'exécution et on sauvegarde leur état dans une variable qui servira de base de référence :

```
PS> $BaseDeRéférence = Get-Process
```

Une heure plus tard, nous décidons de refaire la même opération afin de comparer avec notre base de référence :

```
PS> $BaseDeDifférences = Get-Process
```

Dans l'heure écoulée, des processus ont certainement été lancés, d'autres arrêtés. Avec ces deux collections, nous pouvons à présent faire notre comparaison :

InputObject	SideIndicator
System.Diagnostics.Process (audiogd)	=>
System.Diagnostics.Process (CCC)	=>
System.Diagnostics.Process (dllhost)	=>
System.Diagnostics.Process (MOM)	=>
System.Diagnostics.Process (MsMpEng)	=>
System.Diagnostics.Process (notepad)	=>
System.Diagnostics.Process (RIconMan)	=>
System.Diagnostics.Process (svchost)	=>
System.Diagnostics.Process (svchost)	=>
System.Diagnostics.Process (taskeng)	=>
System.Diagnostics.Process (VBoxSVC)	=>
System.Diagnostics.Process (VirtualBox)	=>
System.Diagnostics.Process (VirtualBox)	=>
System.Diagnostics.Process (WMIADAP)	=>
System.Diagnostics.Process (AdobeARM)	<=
System.Diagnostics.Process (runonce)	<=

La cmdlet `Compare-Object` est invoquée avec le paramètre `-ReferenceObject`, qui sert à spécifier une base de référence, et le paramètre `-DifferenceObject` pour spécifier un objet ou une collection d'objets qui sera comparé à la base de référence. Le résultat nous montre une liste de processus dans la première colonne, nommée `InputObject`. La seconde colonne, nommée `SideIndicator`, montre les signes :

- `=>` qui met en évidence les valeurs apparaissant uniquement dans la base comparée à la base de référence ;
- `<=` qui met en évidence les valeurs apparaissant uniquement dans la base de référence.

Il existe aussi le signe `==` qui indique qu'une valeur se trouve dans les deux bases. S'il n'apparaît pas dans le résultat, c'est que nous n'avons pas utilisé le paramètre `-IncludeEqual`. L'inclure impliquerait d'avoir des résultats plus longs à analyser. Un rapport ou une investigation peut donc être développé à partir de ces informations.

## Comparer sur la base de propriétés d'objets

En plus de pouvoir comparer des listes d'objets, il est possible via le paramètre `-Property` de comparer des propriétés d'objets, donc de faire des comparaisons plus fines. Dans l'exemple qui va suivre, nous allons comparer les processeurs de deux serveurs, l'un nommé `srv01` et l'autre `srv02`. Cette information qui nous a été demandée par un membre d'une autre équipe peut être trouvée de plusieurs façons. L'objectif,

vous l'aurez compris, est de trouver cette information via PowerShell. En sachant que `srv01` sera notre base de référence, commençons par les interroger l'un après l'autre :

```
PS> $srv01 = Get-WmiObject -Class Win32_Processor -ComputerName srv01
PS> $srv02 = Get-WmiObject -Class Win32_Processor -ComputerName srv02
```

Nous avons utilisé la cmdlet `Get-WmiObject` que nous étudierons dans un autre chapitre, pour utiliser le service WMI et obtenir des informations sur le type de processeur que contient chacune de ces machines. Mais comment allons-nous faire cette comparaison ? En effet, si nous comparons directement ces deux objets, cela n'a aucun sens car chaque propriété est différente. Ce qui nous intéresse ici est le nom de chacun de ces processeurs ; donc, essayons de nous renseigner sur les propriétés dont disposent ces objets :

```
PS> $srv01 | gm
```

TypeName : System.Management.ManagementObject#root\cimv2\...

Name	MemberType	Definition
PSComputerName	AliasProperty	PSComputerName =
Reset	Method	System.Manageme
SetPowerState	Method	System.Manageme
AddressWidth	Property	uint16 AddressWi
Architecture	Property	uint16 Architect
Availability	Property	uint16 Availabil
Caption	Property	string Caption {
ConfigManagerErrorCode	Property	uint32 ConfigMan
ConfigManagerUserConfig	Property	bool ConfigManag
CpuStatus	Property	uint16 CpuStatus
CreationClassName	Property	string CreationC
CurrentClockSpeed	Property	uint32 CurrentCl
CurrentVoltage	Property	uint16 CurrentVo
DataWidth	Property	uint16 DataWidth
Description	Property	string Descripti
DeviceID	Property	string DeviceID
ErrorCleared	Property	bool ErrorCleare
ErrorDescription	Property	string ErrorDesc
ExtClock	Property	uint32 ExtClock
Family	Property	uint16 Family {g
InstallDate	Property	string InstallDa
L2CacheSize	Property	uint32 L2CacheSi
L2CacheSpeed	Property	uint32 L2CacheSp
L3CacheSize	Property	uint32 L3CacheSi
L3CacheSpeed	Property	uint32 L3CacheSp
LastErrorCode	Property	uint32 LastError
Level	Property	uint16 Level {ge

LoadPercentage	Property	uint16 LoadPerce
Manufacturer	Property	string Manufactu
MaxClockSpeed	Property	uint32 MaxClockS
<b>Name</b>	Property	string Name {get
NumberOfCores	Property	uint32 NumberOfC
NumberOfLogicalProcessors	Property	uint32 NumberOfL
OtherFamilyDescription	Property	string OtherFami
PNPDeviceID	Property	string PNPDevice
PowerManagementCapabilities	Property	uint16[] PowerMa
PowerManagementSupported	Property	bool PowerManage
ProcessorId	Property	string Processor
ProcessorType	Property	uint16 Processor
Revision	Property	uint16 Revision
Role	Property	string Role {get
SecondLevelAddressTranslationExtensions	Property	bool SecondLevel
SocketDesignation	Property	string SocketDes
Status	Property	string Status {g
StatusInfo	Property	uint16 StatusInf
Stepping	Property	string Stepping
SystemCreationClassName	Property	string SystemCre
SystemName	Property	string SystemNam
UniqueId	Property	string UniqueId
UpgradeMethod	Property	uint16 UpgradeMe
Version	Property	string Version {
VirtualizationFirmwareEnabled	Property	bool Virtualizat
VMMonitorModeExtensions	Property	bool VMMonitorMo
VoltageCaps	Property	uint32 VoltageCa
__CLASS	Property	string __CLASS {
__DERIVATION	Property	string[] __DERIV
__DYNASTY	Property	string __DYNASTY
__GENUS	Property	int __GENUS {get
__NAMESPACE	Property	string __NAMESPA
__PATH	Property	string __PATH {g
__PROPERTY_COUNT	Property	int __PROPERTY_C
__RELPATH	Property	string __RELPATH
__SERVER	Property	string __SERVER
__SUPERCLASS	Property	string __SUPERCL
PSConfiguration	PropertySet	PSConfiguration
PSStatus	PropertySet	PSStatus {Availa
ConvertFromDateTime	ScriptMethod	System.Object Co
Convert.ToDateTime	ScriptMethod	System.Object Co

La propriété `Name` retient l'attention, qui correspond exactement à ce que nous cherchons :

```
PS> Compare-Object -ReferenceObject $srv01 -DifferenceObject $srv02 -Property Name
```

Name	SideIndicator
Intel Core i3	=>
Intel Core i5	

En analysant le résultat, `srv01` a un processeur Intel Core i5, tandis que `srv02` a un processeur Intel Core i3. Nous pouvons maintenant envoyer l'information à la personne concernée.

La cmdlet `Compare-Object` est donc un excellent outil pour élaborer des rapports comparatifs.

## Filtrer les objets

Depuis le début du chapitre, nous avons présenté plusieurs modes de traitements des objets. Dans cette section, nous étudierons une cmdlet qui est sans doute une des plus puissantes en matière de sélection des objets : `Where-Object`. Cette cmdlet crée un filtre sélectif qui contrôle les flux d'objets à travers le pipeline. De plus, avec la version 3 de PowerShell, un nouveau mode d'approche de cette cmdlet permet d'envisager un autre type de construction.

### L'approche classique

Une des caractéristiques fortes de `Where-Object` est l'utilisation de blocs de script ; c'est grâce à eux que le processus de filtrage s'effectuera. La philosophie de cette commande est finalement simple : pour chaque objet traité, le bloc de script est exécuté. L'opération a deux issues possibles.

- Le résultat de l'exécution est `true` ; dans ce cas, l'objet est retourné.
- Le résultat de l'exécution est `false` ; dans ce cas, l'objet est ignoré.

Évidemment, pour que l'évaluation soit faite, le niveau de critères doit être spécifié à l'intérieur du bloc de script délimité par les accolades `{}`. La ligne de commande suivante illustre l'utilisation des cmdlets `Get-Service` et `Where-Object` :

```
PS> Get-Service | Where-Object -FilterScript { $_.Status -eq 'Stopped' }
```

Status	Name	DisplayName
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service.

Stopped	AeLookupSvc	Expérience d'application
Stopped	ALG	Service de la passerelle de la couc...
Stopped	AllUserInstallA...	Agent d'installation pour tous les ...
Stopped	AppIDSvc	Identité de l'application
Stopped	AxInstSV	Programme d'installation ActiveX (A...
Stopped	BDESV	Service de chiffrement de lecteur B...
Stopped	Browser	Explorateur d'ordinateurs
Stopped	bthserv	Service de prise en charge Bluetooth
Stopped	CertPropSvc	Propagation du certificat
Stopped	COMSysApp	Application système COM+
Stopped	defragsvc	Optimiser les lecteurs
Stopped	DeviceInstall	Service d'installation de périphérique
Stopped	dot3svc	Configuration automatique de réseau...
Stopped	DsmSvc	Gestionnaire d'installation de péri...
Stopped	Eaphost	Protocole EAP (Extensible Authentic...
Stopped	EFS	Système de fichiers EFS (Encrypting...
...		

L'objectif est de lister les services actuellement arrêtés. La cmdlet `Get-Service` liste tous les services présents sur le système, mais sans distinction de leur état. Si nous sommes intéressés uniquement par un état particulier, alors le recours à la cmdlet `Where-Object` devient justifié. Donc, `Get-Service` envoie la collection d'objets à `Where-Object` qui, via son paramètre `-FilterScript`, ne retient que ceux dont l'état est arrêté. Pour atteindre notre objectif, nous avons utilisé la méthode classique, qui peut paraître pour certains utilisateurs un peu trop verbeuse.

## L'approche simplifiée

PowerShell version 3 inaugure une nouvelle approche dans la façon dont on peut utiliser la cmdlet `Where-Object`. Dans l'exemple précédent, nous avons utilisé un opérateur de comparaison, mais à l'intérieur du bloc de script. Avec la nouvelle approche, les opérateurs de comparaison deviennent des paramètres à part entière. Voici l'exemple précédent, mais écrit via cette nouvelle approche :

```
PS> Get-Service | Where-Object -Property Status -eq -Value Stopped
```

Status	Name	DisplayName
-----	-----	-----
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service
Stopped	AeLookupSvc	Expérience d'application
Stopped	ALG	Service de la passerelle de la couc...
Stopped	AllUserInstallA...	Agent d'installation pour tous les ...
Stopped	AppIDSvc	Identité de l'application
Stopped	AxInstSV	Programme d'installation ActiveX (A...
Stopped	BDESV	Service de chiffrement de lecteur B...

Stopped	Browser	Explorateur d'ordinateurs
Stopped	bthserv	Service de prise en charge Bluetooth
Stopped	CertPropSvc	Propagation du certificat
Stopped	COMSysApp	Application système COM+
Stopped	defragsvc	Optimiser les lecteurs
Stopped	DeviceInstall	Service d'installation de périphérique
Stopped	dot3svc	Configuration automatique de réseau...
Stopped	DsmSvc	Gestionnaire d'installation de péri...
Stopped	Eaphost	Protocole EAP (Extensible Authentic...
Stopped	EFS	Système de fichiers EFS (Encrypting...
...		

La sortie est exactement la même. La différence fondamentale réside dans la façon dont nous avons utilisé la cmdlet `Where-Object`. On observe qu'il n'y a plus de blocs de script, mais uniquement des paramètres :

- `-Property` : pour spécifier la propriété à partir de laquelle le filtrage va s'effectuer ;
- `-Value` : pour indiquer la valeur de cette propriété.

Cette syntaxe a le mérite d'être simple, car l'accent est mis sur la propriété servant de base au processus de filtrage des objets. Toutefois, cette nouvelle possibilité ne permet malheureusement pas d'élaborer des critères de filtrage plus ou moins sophistiqués.

## Effectuer des opérations sur chaque objet

La cmdlet `Foreach-Object` sert à effectuer des actions sur chaque objet à travers le pipeline. Elle est, à l'instar de la cmdlet `Where-Object`, une cmdlet de contrôle de flux très puissante et utilise les blocs de script, ce qui n'est pas un hasard. La différence au niveau des blocs de script est que `Foreach-Object` en propose trois, alors que `Where-Object` n'en propose qu'un. Donc, les modes d'évaluation sont multiples et sont justifiés par la finalité de `Foreach-Object`. La façon dont cette dernière traite les objets peut aller d'un à trois temps :

- les actions à exécuter sur chaque objet à travers le paramètre `-Process` ;
- une ou plusieurs instruction(s) à exécuter avant le traitement du premier objet (via le paramètre `-Begin`) ;
- une ou plusieurs instruction(s) à exécuter après le traitement du dernier objet (via le paramètre `-End`).

Un autre point commun avec `Where-Object` est que `Foreach-Object` inaugure un nouveau type d'approche.

## L'approche classique

L'approche classique de la syntaxe de la cmdlet `Foreach-Object` est symbolisée par l'utilisation de blocs de script. Et cette règle est valable en ce qui concerne les trois paramètres essentiels (car il y en a d'autres) cités plus haut. L'exemple que nous allons prendre pour illustrer ceci sera très simple :

```
PS> 1,2,3 | ForEach-Object -Begin {Write-Host "Début du traitement.."} -Process
{$_ * 3} -End {Write-Host "Fin du traitement.."}
Début du traitement..
3
6
9
Fin du traitement..
```

Ici, nous envoyons trois entiers à `Foreach-Object`, qui prétraite les objets via son paramètre `-Begin` en nous donnant une information. Puis, via le paramètre `-Process`, chaque entier est multiplié par trois, et le résultat des trois opérations est affiché en sortie. Une fois que tous les objets sont traités, une information (via le paramètre `-End`) indiquant la fin du traitement nous est donnée.

**NOTE** Sur la variable spéciale `$_`

Vous avez sûrement remarqué depuis le début de l'ouvrage une syntaxe qui ressemble à `$_`. Il s'agit d'une variable spéciale représentant l'objet d'entrée actif, que nous étudierons dans le chapitre consacré aux variables.

Là aussi, comme pour `Where-Object`, l'approche utilisant les blocs de script peut paraître verbeuse, mais PowerShell version 3 propose aussi une simplification de la syntaxe de cette cmdlet.

## L'approche simplifiée

Cette approche, bien que produisant le même résultat que celle utilisant les blocs de script, est idéale lorsque l'utilisateur emploie souvent la console. En effet, l'attention est portée uniquement sur les paramètres :

```
PS> "Windows PowerShell" | ForEach-Object -MemberName 'Split' -ArgumentList ' '
Windows
PowerShell
```

Dans cet exemple, nous divisons une chaîne de caractères en deux sous-chaînes à partir d'un délimiteur donné. La chaîne de caractères séparée par une espace (qui constitue le délimiteur) est envoyée à `Foreach-Object`. Les membres de l'objet à traiter sont invoqués avec le paramètre `-MemberName` et les arguments (lorsqu'il s'agit de méthodes) sont fournis au paramètre `-ArgumentList`.

Ce type de construction est intéressant, mais l'utilisateur peut sur le long terme être dérouté à cause des limites imposées essentiellement par une marge de manœuvre réduite.

## Créer des objets

Nous entrons dans la dernière section de ce chapitre consacré à la manipulation des objets. Le but est d'y apprendre à créer des objets avec PowerShell. La cmdlet par excellence pour réaliser cela se nomme `New-Object`.

### Créer un objet .NET

`New-Object` sert à créer des objets .NET à partir d'une classe .NET, mais aussi des objets COM (que nous verrons dans un prochain chapitre). La philosophie de base est que, lorsque l'objet est créé, sa structure est définie par la classe à partir de laquelle il est issu, mais il est vierge en termes de données. Par exemple, essayons de créer un objet PowerShell simple :

```
PS> $Objet = New-Object -TypeName PSCustomObject
```

`New-Object` est invoquée avec le paramètre `-TypeName` permettant d'indiquer le nom de la classe .NET, et la valeur spécifiée à ce paramètre est `PSCustomObject`. En réalité, cette valeur n'a pas été mise ici de manière hasardeuse. C'est en effet le nom d'une classe .NET particulière, donnant une consistance particulière aux objets dans PowerShell, mais permettant surtout de façonnier ces objets comme l'utilisateur le souhaite, car ici aussi la structure est vierge. Si l'objet que nous venons de créer est appelé, il ne se passe rien :

```
PS> $Objet
```

```
PS>
```

Comme nous pouvons l'observer, la sortie n'affiche rien et ce comportement est tout à fait normal. L'objet est vierge à tous les niveaux ; il faut à présent lui attribuer des membres.

## Ajouter des membres aux objets créés

Nous venons de créer un objet vierge, c'est-à-dire ne contenant pas de données particulières. Pour définir des membres personnalisés, nous utiliserons la cmdlet [Add-Member](#) et ajouterons deux propriétés :

```
PS> $Objet | Add-Member -NotePropertyName Prénom -NotePropertyValue 'Kais'  
PS> $Objet | Add-Member -NotePropertyName Nom -NotePropertyValue 'Ayari'
```

La propriété [Prénom](#) a d'abord été ajoutée grâce au paramètre [-NotePropertyName](#) et la valeur de cette même propriété grâce au paramètre [-NotePropertyValue](#). Le même procédé a été employé pour la propriété [Nom](#). Notez que la variable [\\$Objet](#) qui encapsule l'objet créé est redirigée à travers le pipeline à la cmdlet [Add-Member](#). Donc, nous avons simplement ajouté deux propriétés, mais il est possible d'ajouter d'autres types de membres (voir l'aide de la cmdlet [Add-Member](#)). À présent, appelons de nouveau l'objet :

```
PS> $Objet
```

Prénom	Nom
-----	---
Kais	Ayari

La sortie nous montre les deux propriétés que nous venons de créer, ainsi que leur valeur. Cet objet est extrêmement simple ; nous pourrions le rendre beaucoup plus complexe, mais ce n'est pas l'objet de ce livre.

### NOTE Concernant la durée de vie d'un objet

La durée de vie d'un objet est égale à celle de la session au cours de laquelle il a été créé. Cependant, il pourrait être sauvegardé dans un fichier XML pour être réutilisé au cours d'une session ultérieure.

Créer des objets est une pratique récurrente chez les utilisateurs de PowerShell. Savoir les manipuler est par conséquent une chose importante, car ils donnent sans conteste plus de complexité et de profondeur aux lignes de commandes et aux scripts.

# 5

## PowerShell et la mise en forme des données

---

*Lorsque PowerShell affiche les données, elles peuvent prendre différents aspects, auxquels l'utilisateur ne prête pas forcément attention. La mise en forme des données est un des points centraux du fonctionnement de PowerShell.*

*Dans ce chapitre, nous étudierons les différents systèmes de mises en forme, à savoir comment distinguer une vue sous forme de tableau de celle sous forme de liste, la personnalisation de l'affichage ainsi que l'affichage d'une seule propriété sous la forme d'une large table.*

### SOMMAIRE

- ▶ Afficher les données de sortie sous forme de tableau : Format-Table
- ▶ Afficher les données de sortie sous forme de liste : Format-List
- ▶ Personnaliser l'affichage des données de sortie : Format-Custom
- ▶ Afficher une seule propriété sous forme de table : Format-Wide

PowerShell structure les données de sortie à partir des informations contenues dans une base de fichiers XML : à chaque fois qu'une commande est exécutée, il interroge la base de données interne afin d'afficher les objets correctement. Par défaut, lors de l'installation de PowerShell, la base est placée dans `C:\Windows\System32\WindowsPowerShell\v1.0`.

**ASTUCE Pour éviter d'avoir à écrire souvent ce même chemin d'accès**

Pour simplifier vos lignes de commandes, vous pouvez utiliser la variable `$PSHOME`, qui contient le chemin d'accès complet menant au répertoire d'installation de PowerShell.

La particularité de cette base de données de mise en forme est que chacun des fichiers qui la constituent contient dans son nom le mot `format`. Regardons cette liste de fichiers :

```
PS> Get-ChildItem $PSHOME\*format*
```

Répertoire : C:\Windows\System32\WindowsPowerShell\v1.0

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	02/03/2012 16:31	27338	Certificate.format.ps1xml
-a---	02/03/2012 16:31	27106	Diagnostics.Format.ps1xml
-a---	02/03/2012 16:31	144442	DotNetTypes.format.ps1xml
-a---	02/03/2012 16:31	14502	Event.Format.ps1xml
-a---	02/03/2012 16:31	21293	FileSystem.format.ps1xml
-a---	02/03/2012 16:31	287938	Help.format.ps1xml
-a---	02/03/2012 16:31	97880	HelpV3.format.ps1xml
-a---	02/03/2012 16:31	101824	PowerShellCore.format.ps1xml
-a---	02/03/2012 16:31	18612	PowerShellTrace.format.ps1xml
-a---	02/03/2012 16:31	13659	Registry.format.ps1xml
-a---	02/03/2012 16:31	17731	WSMan.Format.ps1xml

La disposition des objets lors de leur affichage, de l'aide intégrée et de tout ce qui nécessite une mise en forme est contenue par défaut dans cet ensemble de fichiers XML. Le contenu de cet ensemble est complexe, ce qui nécessite de ne pas faire de modifications sans avoir sauvegardé au préalable ces fichiers. Ceci est très important, car en cas de modifications sans contrôle d'un de ces fichiers, des dysfonctionnements au niveau de l'affichage peuvent apparaître et provoquer des situations très problématiques où il sera extrêmement difficile de remonter les changements qui ont été effectués. La prudence est donc une vertu essentielle dans ce genre de situation.

Le fait est que, dans l'immense majorité des cas, la vue proposée sera assez satisfaisante, car l'équipe de développement de PowerShell a fait en sorte que, pour chaque

objet, les propriétés les plus significatives soient affichées. Cela ne veut pas dire que les autres propriétés ne le sont pas, car cela relève en réalité de ce que l'utilisateur recherche en termes d'informations. Il s'agit ici de proposer des vues cohérentes facilitant la compréhension de l'utilisateur.

## Afficher les données de sortie sous forme de tableau

Lors d'une sortie sous forme de tableau, PowerShell affiche les propriétés de l'objet en colonnes. Si l'affichage par défaut de l'objet a une disposition en tableau et si l'utilisateur a voulu ce mode d'affichage, alors il pourra agir sur les propriétés à afficher. Si l'affichage par défaut de l'objet a une disposition en liste de propriétés, alors il pourra aussi, s'il le souhaite, privilégier le mode d'affichage en tableau, via la cmdlet [Format-Table](#).

### La cmdlet Format-Table

[Format-Table](#) est la cmdlet numéro un si un utilisateur souhaite afficher les données à la manière d'un tableau. Dans l'exemple qui va suivre, nous allons lister les processus en cours d'exécution et ne sélectionner que les propriétés [ProcessName](#) et [CPU](#) :

```
PS> Get-Process | Format-Table -Property ProcessName, CPU
```

ProcessName	CPU
AcroRd32	0,9984064
AcroRd32	40,0766569
armsvc	
atiesrxx	
audiogd	440,0164206
CCC	24,3829563
CLMLSvc	0,2964019
conhost	3,9156251
csrss	
csrss	
dasHost	
dllhost	
dwm	
explorer	51,8703325
firefox	244,7811691
Fuel.Service	
hotkey	0,0936006
IdeaTouch.LocalDataServer...	
Idle	
...	

Comme on peut le constater, les deux propriétés affichées correspondent à celles que nous avons sélectionnées avec le paramètre `-Property`. Cependant, l'espace entre les deux colonnes est peut-être un peu trop grand. Le paramètre `-AutoSize` nous permettra d'ajuster automatiquement la taille des colonnes :

```
PS> Get-Process | Format-Table -Property ProcessName, VM -AutoSize
```

ProcessName	CPU
AcroRd32	0,9984064
AcroRd32	40,0766569
armsvc	
atiesrxx	
audiogd	440,0164206
CCC	24,3829563
CLMLSvc	0,2964019
conhost	3,9156251
csrss	
csrss	
dasHost	
dllhost	
dwm	
explorer	51,8703325
firefox	244,7811691
Fuel.Service	
hotkey	0,0936006
IdeaTouch.LocalDataServer...	
Idle	
...	

Ici, les largeurs des colonnes ont été ajustées automatiquement. Les en-têtes de colonnes peuvent être omis avec le paramètre `-HideTableHeaders` pour ne garder que les données :

```
PS> C:\Users\Kais> Get-Process | Format-Table -Property ProcessName, VM -AutoSize -HideTableHeaders
```

AcroRd32	0,9984064
AcroRd32	40,0766569
armsvc	
atiesrxx	
audiogd	440,0164206
CCC	24,3829563
CLMLSvc	0,2964019
conhost	3,9156251
csrss	
csrss	

```
dasHost
d11host
dwm
explorer          51,8703325
firefox           244,7811691
Fuel.Service
...
```

## Les propriétés personnalisées

Dans le cadre d'un affichage en tableau, on peut se poser la question du renommage des colonnes, c'est-à-dire finalement de leur personnalisation. PowerShell offre cette perspective. La syntaxe d'une propriété personnalisée est celle d'une table de hachage (que nous étudierons dans un prochain chapitre).

### Syntaxe d'une propriété personnalisée

```
@{ ❶ Label = <Chaîne> ; ❷ Expression = { <Bloc de script ou chaîne> } }
```

La table de hachage doit comporter au moins deux clés. Ici, la clé ❶ nommée `Label` représente le nouveau nom de la propriété et la clé ❷ nommée `Expression` fournit ou calcule la valeur de cette même propriété. Les clés valides sont au nombre de cinq :

- `Name` ou `Label` spécifie le nom de la nouvelle propriété.
- `Expression` calcule la valeur de la propriété.
- `FormatString` spécifie un certain type de format (comme le nombre de décimales après la virgule).
- `Width` indique une largeur de colonne.
- `Alignment` aligne la valeur de la propriété à gauche, au centre ou à droite.

Reprendons l'exemple précédent et essayons de changer les noms de propriétés `ProcessName` en `Nom du processus` et `CPU` en `Temps processeur` :

```
PS> Get-Process | Format-Table -Property @{Label='Nom du
processus';Expression={$_.ProcessName}}, @{Label='Temps Processeur';
Expression={$_.Cpu}} -AutoSize
```

Nom du processus	Temps Processeur
AcroRd32	1,0140065
AcroRd32	76,3312893
armsvc	
atiesrxx	
audiodg	839,8781838
CCC	24,804159

CLMLSvc	0,5772037
conhost	7,6128488
csrss	
csrss	
dasHost	
dllhost	
dwm	
...	

Le nom des colonnes a changé et nous avons réussi à personnaliser l'affichage. La cmdlet `Format-Table` est donc un outil très puissant pour disposer les données de sortie sous forme de tableau.

**NOTE** Sur les propriétés personnalisées

La cmdlet `Select-Object` accepte aussi les propriétés personnalisées, mais dans une moindre mesure.

## Afficher les données de sortie sous forme de liste

Après avoir vu comment disposer les données sous forme de tableau, nous étudierons dans cette section comment les présenter sous forme de liste de propriétés. En effet, ce type de disposition est à privilégier lorsqu'il faut étudier un objet quelconque en profondeur. La cmdlet `Format-List` affiche les propriétés d'un objet sous forme d'une liste où chaque propriété est disposée sur une ligne distincte.

### La cmdlet Format-List

Dans l'immense majorité des cas, PowerShell affiche seulement certaines des propriétés de l'objet. Pour qu'il les affiche toutes, il faut appeler la cmdlet `Format-List`. L'exemple suivant illustre l'interrogation de la classe WMI [`Win32_Processor`] :

```
PS> Get-WmiObject -Class Win32_Processor
```

Caption	: AMD64 Family 20 Model 2 Stepping 0
DeviceID	: CPU0
Manufacturer	: AuthenticAMD
MaxClockSpeed	: 1400
Name	: AMD E1-1200 APU with Radeon(tm) HD Graphics
SocketDesignation	: Socket FT1

Cette classe n'a-t-elle vraiment que six propriétés ? On pourrait le penser en nous basant simplement sur la sortie. Pour le savoir, nous reprendrons notre ligne de commande, à laquelle nous ferons participer la cmdlet `Format-List` :

```
PS> Get-WmiObject -Class Win32_Processor | Format-List -Property *
```

PSComputerName	:	LENDERK
Availability	:	3
CpuStatus	:	1
CurrentVoltage	:	13
DeviceID	:	CPU0
ErrorCleared	:	
ErrorDescription	:	
LastErrorCode	:	
LoadPercentage	:	65
Status	:	OK
StatusInfo	:	3
AddressWidth	:	64
DataWidth	:	64
ExtClock	:	100
L2CacheSize	:	1024
L2CacheSpeed	:	
MaxClockSpeed	:	1400
PowerManagementSupported	:	False
ProcessorType	:	3
Revision	:	512
SocketDesignation	:	Socket FT1
Version	:	Modèle 2, niveau 0
VoltageCaps	:	
__GENUS	:	2
__CLASS	:	Win32_Processor
__SUPERCLASS	:	CIM_Processor
__DYNASTY	:	CIM_ManagedSystemElement
__RELPATH	:	Win32_Processor.DeviceID="CPU0"
__PROPERTY_COUNT	:	51
__DERIVATION	:	{CIM_Processor, CIM_LogicalDevice, CIM_LogicalElement, CIM_ManagedSystemElement}
__SERVER	:	LENDERK
__NAMESPACE	:	root\cimv2
__PATH	:	
\\LENDERK\root\cimv2:Win32_Processor.DeviceID="CPU0"	:	
Architecture	:	9
Caption	:	AMD64 Family 20 Model 2 Stepping 0
ConfigManagerErrorCode	:	
ConfigManagerUserConfig	:	
CreationClassName	:	Win32_Processor
CurrentClockSpeed	:	1400
Description	:	AMD64 Family 20 Model 2 Stepping 0

```

Family : 71
InstallDate :
L3CacheSize : 0
L3CacheSpeed : 0
Level : 20
Manufacturer : AuthenticAMD
Name : AMD E1-1200 APU with Radeon(tm) HD
Graphics
NumberOfCores : 2
NumberOfLogicalProcessors : 2
OtherFamilyDescription :
PNPDeviceID :
PowerManagementCapabilities :
ProcessorId : 00500F20178BF0FF
Role : CPU
SecondLevelAddressTranslationExtensions : True
Stepping : 0
SystemCreationClassName : Win32_ComputerSystem
SystemName : LENDESK
UniqueId :
UpgradeMethod : 6
VirtualizationFirmwareEnabled : True
VMMonitorModeExtensions : True
Scope : System.Management.ManagementScope
Path :
\\LENDESK\root\cimv2:Win32_Processor.DeviceID="CPU0"
Options : System.Management.ObjectGetOptions
ClassPath : \\LENDESK\root\cimv2:Win32_Processor
Properties : {AddressWidth, Architecture,
Availability, Caption...}
SystemProperties : {__GENUS, __CLASS, __SUPERCLASS,
__DYNASTY...}
Qualifiers : {dynamic, Locale, provider, UUID}
Site :
Container :

```

Cette fois, l'affichage nous montre bien que la classe WMI `[Win32_Processor]` a un nombre de propriétés bien supérieur à celui constaté lors de l'affichage précédent. La valeur \* du paramètre `-Property` indique que nous voulons lister toutes les propriétés de la classe interrogée. Si nous ne sommes intéressés que par certaines d'entre elles tout en voulant conserver ce type de disposition, il suffit de les préciser à la cmdlet `Format-List` :

```
PS> Get-WmiObject -Class Win32_Processor | Format-List -Property Name, Status,
Architecture, NumberOfCores
```

```

Name : AMD E1-1200 APU with Radeon(tm) HD Graphics
Status : OK

```

```
Architecture : 9
NumberofCores : 2
```

L'affichage des données sous forme de liste est idéal lorsqu'un utilisateur souhaite connaître un peu plus l'objet qu'il manie. Il faut souligner qu'il n'y a pas une disposition qui soit meilleure qu'une autre dans l'absolu, mais qu'un affichage particulier est à privilégier dans un contexte particulier.

## Personnaliser l'affichage des données de sortie

Parfois, un utilisateur peut avoir l'envie ou le besoin de disposer les données de sortie dans un format autre que les formats standards. La cmdlet `Format-Custom` permet ce type d'alternative.

### La cmdlet Format-Custom

`Format-Custom` a pour objectif de créer des vues alternatives à celles sous forme de liste ou de tableau, vues qui peuvent être indispensables si l'on veut explorer la complexité des objets qui sont affichés. Par exemple, si nous affichons les propriétés de l'objet correspondant au répertoire `C:\PerfLogs`, nous aurons la sortie suivante :

```
PS> Get-Item C:\PerfLogs
Répertoire : C:\

Mode          LastWriteTime    Length Name
----          -----          ---- 
d---  26/07/2012      09:33          PerfLogs
```

La sortie affiche quatre propriétés et leur valeur respective. Dans le monde du framework .NET, les objets peuvent avoir des propriétés qui constituent d'autres objets et ainsi de suite. Et cette complexité masquée volontairement au niveau de la sortie a pour but d'éviter d'avoir à lire des informations trop verbeuses dans un premier temps. Si nous voulons analyser davantage la structure de l'objet, nous pouvons, et c'est le but de cette section, utiliser la cmdlet `Format-Custom` :

```
PS> Get-Item C:\PerfLogs | Format-Custom -Depth 1
class DirectoryInfo
{
    PSPath = Microsoft.PowerShell.Core\FileSystem::C:\PerfLogs
    PSParentPath = Microsoft.PowerShell.Core\FileSystem::C:\
```

```
PSChildName = PerfLogs
PSDrive =
  class PSDriveInfo
  {
    CurrentLocation = Users\Kais
    Name = C
    Provider = Microsoft.PowerShell.Core\FileSystem
    Root = C:\
    Description = Windows8_OS
    Credential = System.Management.Automation.PSCredential
    DisplayRoot =
    Used = 157191426048
    Free = 814817550336
  }
  PSPrinter =
  class ProviderInfo
  {
    ImplementingType = Microsoft.PowerShell.Commands.FileSystemProvider
    HelpFile = System.Management.Automation.dll-Help.xml
    Name = FileSystem
    PSSnapIn = Microsoft.PowerShell.Core
    ModuleName = Microsoft.PowerShell.Core
    Module =
    Description =
    Capabilities = Filter, ShouldProcess, Credentials
    Home = C:\Users\Kais
    Drives =
    [
      C
      F
      D
      E
      ...
    ]
  }
  ...
}
```

La sortie précédente est un extrait du résultat, car celui-ci est vraiment long. Comme nous pouvons le voir, l'objet analysé est constitué d'une cascade d'objets. Remarquez l'utilisation du paramètre `-Depth` qui permet de choisir le niveau de profondeur : plus il est élevé, plus l'objet sera approfondi, ce qui implique d'avoir un affichage de plus en plus long.

La cmdlet `Format-Custom` est idéale lorsqu'un utilisateur souhaite explorer les objets qu'il manipule, même s'il pourrait privilégier une multitude d'autres moyens à sa disposition. Notez que l'utilisateur peut créer ses propres vues dans des fichiers PS1XML et les passer comme valeur au paramètre `-View` de la cmdlet `Format-Custom`.

## Afficher une seule propriété sous forme de tableau

Nous avons remarqué que les cmdlets `Format-Table` et `Format-List` produisaient des sorties disposant plusieurs propriétés. L'affichage de ces dernières peut être tronqué à cause de la longueur de certaines valeurs en lien avec ce que peut afficher la console. Il est donc parfois nécessaire, si l'on s'intéresse à une unique propriété, de n'afficher que celle-ci.

### La cmdlet `Format-Wide`

`Format-Wide` met l'accent sur une propriété en particulier. Tapons en console la cmdlet `Get-Service` :

```
PS> Get-Service
```

Status	Name	DisplayName
Running	AdobeARMservice	Adobe Acrobat Update Service
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service
Stopped	AeLookupSvc	Expérience d'application
Stopped	ALG	Service de la passerelle de la couc...
Stopped	AllUserInstallA...	Agent d'installation pour tous les ...
Running	AMD External Ev...	AMD External Events Utility
Running	AMD FUEL Service	AMD FUEL Service
Stopped	AppIDSvc	Identité de l'application
Running	Appinfo	Informations d'application
Stopped	aspnet_state	ASP.NET State Service
Running	AudioEndpointBu...	Générateur de points de terminaison...
Running	Audiosrv	Audio Windows
Stopped	AxInstSV	Programme d'installation ActiveX (A...
Stopped	BDESVC	Service de chiffrement de lecteur B...
Running	BFE	Moteur de filtrage de base
Running	BITS	Service de transfert intelligent en...

Si l'on observe la propriété `Name`, on s'aperçoit que certaines valeurs ont un affichage tronqué. Parmi une multitude de possibilités, la cmdlet `Format-Wide` sait afficher la propriété `Name` sous forme de tableau :

```
PS> Get-Service | Format-Wide -Property Name
```

AdobeFlashPlayerUpdateSvc	ALG
AeLookupSvc	AppIDSvc
AMD FUEL Service	aspnet_state
Appinfo	Audiosrv
AudioEndpointBuilder	BDESVC
AxInstSV	BFE
BFE	BITS

BrokerInfrastructure	Browser
bthserv	CertPropSvc
COMSysApp	CryptSvc
DcomLaunch	defragsvc
DeviceAssociationService	DeviceInstall
Dhcp	Dnscache
dot3svc	DPS
DsmSvc	Eaphost
EFS	EventLog
EventSystem	Fax
fdPHost	FDResPub
fhsvc	FontCache
FontCache3.0.0.0	gpsvc
hidserv	hkmsvc
HomeGroupProvider	IconMan_R
IdeaTouch.LocalDataServer.Education	IKEEXT
iphlpvc	JME Keyboard
KeyIso	KtmRm
...	

La sortie nous montre bien que les noms des services s'affichent entièrement. Elle se structure en deux colonnes, ce qui peut être changé via le paramètre `-Column` :

```
PS> Get-Process | Format-Wide -Property Id -Column 3
```

2640	4076	1672
3404	892	3052
3656	5100	5180
468	4580	1772
4020	3432	664
2128	4208	4824
1696	2760	1732
0	2580	2496

Cet exemple illustre en plus du paramètre `-Column`, l'utilisation du paramètre `-Property` spécifiant la propriété à afficher. Plus le nombre de colonnes est élevé et plus l'affichage sera dense.

`Format-Wide` est donc une cmdlet à utiliser dans des cas très précis, c'est-à-dire exceptionnels, car il y a bien d'autres moyens de mettre en perspective une propriété.

# 6

## PowerShell et la sécurité

---

*La sécurité est une composante essentielle de l'écosystème de PowerShell. En effet, il y a eu par le passé avec VBScript d'énormes failles de sécurité qui ont facilité l'apparition de certains virus et autres programmes malveillants dans les années 1999 et 2000. Cela a conduit Microsoft à repenser sa politique de sécurité concernant certaines des technologies que la firme met en avant.*

*Dans ce chapitre, nous commencerons par nous familiariser avec les stratégies d'exécution pouvant être mises en œuvre dans PowerShell. Puis nous verrons comment obtenir un certificat numérique qui nous permettra de signer nos scripts.*

### SOMMAIRE

- ▶ Les stratégies d'exécution
- ▶ Obtenir un certificat pour signer les scripts
- ▶ Signer numériquement des scripts

## Les stratégies d'exécution PowerShell

Une stratégie d'exécution constitue une règle de base à partir de laquelle PowerShell exécute les scripts. Toutefois, cette règle de base peut aisément être contournée, car le but n'est pas de faire de PowerShell un antivirus, mais d'établir, à partir de ces stratégies, un certain nombre de principes de sécurité.

### Fonctionnement des stratégies d'exécution

Une stratégie d'exécution a un mode de fonctionnement particulier selon le contexte dans lequel elle est définie. En fonction de la cible sur laquelle elle s'applique, sa configuration résidera dans un endroit différent.

- **Un utilisateur particulier**

La configuration est stockée dans la base de registres.

- **Un ordinateur local**

La configuration est également stockée dans la base de registres.

- **Une session particulière**

La configuration est stockée en mémoire, ce qui implique son effacement lors de la fermeture de session.

- **Un domaine**

La configuration est stockée dans l'Active Directory.

De plus, elle peut affecter plusieurs types d'étendues.

- **Le processus PowerShell**

La stratégie d'exécution affecte uniquement la session en cours d'exécution. Dans ce cas, la valeur est [Process](#).

- **L'utilisateur actuel**

La stratégie d'exécution influe sur l'utilisateur actuel. Dans ce cas, la valeur est [CurrentUser](#).

- **L'ordinateur local**

La stratégie d'exécution affecte tous les utilisateurs se servant de PowerShell dans l'ordinateur local. Dans ce cas, la valeur est [LocalMachine](#).

Au regard de tous ces éléments, il appartient aux personnes chargées de mettre en place ces stratégies d'exécution de bien penser les types d'usages en fonction des différents contextes.

## Les types de stratégies d'exécution

Vous l'aurez sans doute compris, il n'y a pas qu'un seul type de stratégie d'exécution, mais plusieurs.

- **Restricted**

Cette stratégie d'exécution permet d'exécuter des commandes PowerShell, mais empêche l'exécution de scripts, de modules, de fichiers de configuration, de fichiers de mise en forme ainsi que les profils PowerShell. C'est la stratégie par défaut.

- **AllSigned**

Exige que les scripts et fichiers de configuration soient signés numériquement à l'aide de certificats.

- **RemoteSigned**

Dans cette configuration, les scripts et fichiers de configuration ayant pour origine des sources autres que l'ordinateur local (comme Internet) doivent être signés numériquement. Les scripts et fichiers de configuration écrits à partir de l'ordinateur local ne requièrent pas de signature numérique.

- **Unrestricted**

Les scripts et fichiers de configuration, d'où qu'ils proviennent, ne nécessitent pas de signature numérique. Toutefois, pour ceux provenant d'Internet, l'utilisateur est prévenu avant leur exécution.

- **Bypass**

Les scripts et fichiers de configuration, d'où qu'ils proviennent, ne nécessitent pas de signature numérique. De plus, l'utilisateur n'est pas prévenu avant leur exécution.

- **Undefined**

Aucune stratégie n'est définie. Dans ce cas, PowerShell sera affecté (à condition que toutes les étendues n'aient pas de stratégies particulières) par la stratégie d'exécution **Restricted**.

Chacune de ces stratégies d'exécution peut correspondre à un ou plusieurs contexte(s). Selon que l'on se situe dans un domaine ou dans un groupe de travail, le niveau de sécurité devra être adapté. Par exemple, dans un domaine Active Directory, la meilleure pratique est d'opter pour une stratégie d'exécution **AllSigned** ou **RemoteSigned**. S'il s'agit d'un groupe de programmeurs avec des postes de travail spécifiques à leurs activités, la stratégie d'exécution ne sera peut-être pas issue de celles qui viennent d'être évoquées, mais plutôt des stratégies **Unrestricted** ou **Bypass**. Cependant, des contextes très particuliers peuvent nécessiter pour les uns telle stratégie, et pour les autres telle autre stratégie. Il faut donc encore une fois souligner la nécessité de penser les différents contextes dans lesquels nous évoluons.

Pour connaître la stratégie d'exécution PowerShell qui s'applique à la session en cours, nous pouvons utiliser la cmdlet `Get-ExecutionPolicy` :

```
PS> Get-ExecutionPolicy  
RemoteSigned
```

Si nous sommes intéressés par la stratégie d'exécution s'appliquant à l'utilisateur actuel, nous utiliserons cette même cmdlet avec son paramètre `-Scope` :

```
PS> Get-ExecutionPolicy -Scope CurrentUser  
Undefined
```

Pour modifier la stratégie d'exécution, il faut utiliser la cmdlet `Set-ExecutionPolicy` :

```
PS> Set-ExecutionPolicy AllSigned
```

Pour modifier la stratégie d'exécution d'une étendue particulière :

```
PS> Set-ExecutionPolicy AllSigned -Scope LocalMachine
```

#### À SAVOIR Précéderce des stratégies d'exécution

PowerShell priorise d'abord les stratégies d'exécution définies à partir des stratégies de groupes, puis celles définies en mode local.

## Obtenir un certificat pour signer les scripts

Depuis le début du chapitre, nous avons étudié les différentes stratégies d'exécution comme bases d'un fonctionnement mieux sécurisé de PowerShell. Le but de ces règles de base n'est pas de faire de PowerShell un antivirus en tant que tel, mais d'éviter qu'il soit lui-même une voie d'accès facile au système d'exploitation. À cette possibilité d'établir des règles de base, PowerShell offre celle de signer numériquement les scripts que nous écrivons, et ce à partir de certificats de signature de code.

### Les différents moyens d'obtenir un certificat

La possibilité de signer les scripts que nous avons écrits ne peut se faire qu'après obtention de certificats de signature de code. Un certificat numérique est en quelque sorte une carte d'identité numérique à partir de laquelle une personne physique ou

morale peut être identifiée. Un certificat sert aussi aux chiffrements des échanges entre plusieurs entités. Il existe deux façons d'obtenir des certificats numériques.

- **Les certificats autosignés**

Ce type de certificat est obtenu rapidement et sans la nécessité de recourir à un organisme tiers. L'avantage de recourir à ce genre de certificats est évidemment leur gratuité. L'inconvénient majeur est que ne pouvant être authentifiés par un organisme tiers, la possibilité de déployer les scripts signés avec ce type de certificats sera très réduite, car l'attitude naturelle est de ne pas faire confiance à une entité physique ou morale que l'on ne connaît pas. Cela étant, les certificats auto-signés sont très utiles dans des réseaux de petite envergure où les personnes qui écrivent du code se connaissent. Dans ce cas de figure, un certificat autosigné a pour autorité l'ordinateur dans lequel il a été créé. Et un script signé avec un tel certificat ne pourra être exécuté que sur cette même machine. Si le script devait être déployé, ne serait-ce que pour une autre machine, le certificat devra être ajouté sur cette autre machine, et ainsi de suite.

- **Les certificats créés par une autorité de certification**

Ce type de certificat est obtenu via une autorité de certification et a un coût. Il peut s'agir d'organismes comme Verisign ou Globalsign qui, en plus d'être reconnus comme étant dignes de confiance, vérifient l'identité de ceux qui souhaitent acquérir ce genre de certificat. Les certificats issus d'autorités de certification sont recommandés dans les grandes entreprises, car ces autorités étant automatiquement approuvées, le déploiement de scripts ou autres fichiers de configuration signés sera bien plus aisé.

**NOTE Sur les autorités de certification**

Une autorité de certification peut être représentée par un organisme tiers, mais aussi par une infrastructure interne à une société. Dans les deux cas, un budget non négligeable devra être prévu.

Le choix de recourir à l'une ou l'autre méthode est primordial et doit se baser sur des études extrêmement rigoureuses. Toute entreprise devrait avoir une politique de sécurité de ses données et une orientation claire en la matière, ce qui n'est évidemment pas le cas dans les faits.

## Créer un certificat autosigné

La méthode que nous privilégierons dans ce livre est celle permettant d'obtenir un certificat autosigné. Pour ce faire, nous utiliserons un outil nommé *Certificate Creation Tool* ou [MakeCert.exe](#) disponible dans le SDK (*Software Development Kit*) ou kit de développement en lien avec le .Net Framework.

**OUTIL SDK pour Windows 8**

► <http://msdn.microsoft.com/fr-fr/windows/desktop/hh852363>

**Figure 6–1**

L'outil MakeCert.exe fait partie de la liste des outils livrés avec le kit de développement Windows 8.



## Kit de développement logiciel Windows (Kit SDK Windows) pour Windows 8

Le Kit de développement logiciel Windows (Kit SDK Windows) pour Windows 8 contient des en-têtes, des bibliothèques et une sélection d'outils que vous pouvez utiliser lorsque vous créez des applications s'exécutant sous les systèmes d'exploitation Windows. Vous pouvez utiliser le Windows SDK avec l'environnement de développement de votre choix pour créer des applications du Windows Store (uniquement sur Windows 8) utilisant des technologies Web (HTML5, CSS3 et JavaScript, par exemple) ; du code natif (C++) et géré (C#, Visual Basic) ; des applications de bureau utilisant le modèle de programmation natif (Win32/COM) ou géré (NET Framework).

Le Windows SDK induit également le **Kit de certification des applications Windows** qui permet de tester le programme de certification Windows 8 et le programme de logo Windows 7 sur votre application de bureau. Si vous souhaitez également tester votre application sur Windows RT, utilisez le **Kit de certification des applications Windows pour Windows RT**.

Le Windows SDK n'est plus livré avec un environnement complet de génération par ligne de commande. Vous devez installer un compilateur et un environnement de génération séparément. Si vous avez besoin d'un environnement de développement complet comprenant des compilateurs et un environnement de génération, vous pouvez télécharger **Visual Studio 2012 Express**, qui induit les composants appropriés du Windows SDK.

Pour accéder à des ressources et à des informations supplémentaires, rendez-vous sur le [Centre de développement Windows](#).

Une fois le kit de développement téléchargé et installé, il faut :

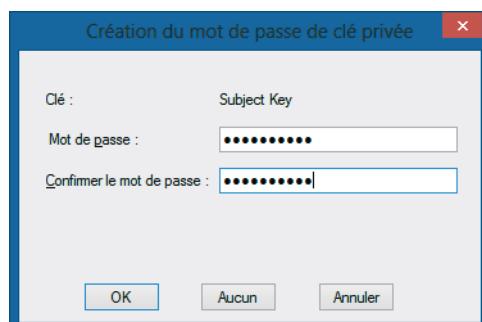
- 1** Rechercher l'emplacement où se trouve l'outil **MakeCert.exe**.
- 2** Lancer une invite de commande en mode administrateur, puis se diriger vers le répertoire contenant l'outil **MakeCert.exe** (dans Windows 8 : **C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bin**).
- 3** Créer une autorité de certification locale via la commande suivante :

```
makecert -n "CN=PowerShell Local Certificate Root" -a sha1 -eku  
1.3.6.1.5.5.7.3.3 -r -sv root.pvk root.cer -ss Root -sr localMachine
```

- 4** Après avoir appuyé sur la touche **Entrée**, une fenêtre devrait apparaître.

**Figure 6–2**

MakeCert.exe demande un mot de passe de clé privée.



Un mot de passe est demandé pour protéger la clé privée.

- 5 Maintenant que l'autorité de certification est créée, il faut fabriquer un certificat personnel provenant de cette autorité :

```
makecert -pe -n "CN=Kais Ayari" -ss MY -a sha1 -eku 1.3.6.1.5.5.7.3.3 -iv  
root.pvk -ic root.cer
```

- 6 En appuyant sur la touche *Entrée*, une boîte de dialogue apparaît.

**Figure 6–3**

Demande de mot de passe pour la création du certificat utilisateur



Via cette fenêtre, le mot de passe qui a été indiqué lors de la création de l'autorité de certification est requis pour valider la création du certificat utilisateur.

- 7 Une fois ces opérations terminées avec le statut *Succeeded* à chaque fois, il faut lancer une console PowerShell et vérifier que le certificat a bien été créé avec la commande suivante :

```
get-childitem cert:\CurrentUser\my -codesigning
```

La sortie devrait afficher une empreinte numérique (qui correspond à la valeur de la propriété *Thumbprint*) :

```
Répertoire : Microsoft.PowerShell.Security\Certificate::CurrentUser\my  
Thumbprint Subject  
-----  
BA88223AB3A0ADA10F044E48DB13B54E5A2DF657 CN=Kais Ayari
```

Le certificat a donc été correctement créé et peut être utilisé pour signer des scripts.

## Signer numériquement des scripts

Une fois le certificat créé, il ne nous reste plus qu'à signer nos scripts à partir de celui-ci. Tout d'abord, nous réalisons un script très simple qui va lister les services en cours

d'exécution. Pour cela, il faut créer un fichier à partir d'un éditeur de texte et y mettre le contenu suivant :

#### Script listant les services en cours d'exécution

```
get-service | where-object {$_._Status -eq "Running"}
```

Le fichier sera nommé `ListeSvcRun` et devra porter l'extension `.ps1`. Pour signer le script, la cmdlet `Set-AuthenticodeSignature` va nous aider :

```
PS> Set-AuthenticodeSignature -Certificate @({Get-ChildItem Cert:\CurrentUser\My-CodeSigningCert)[0]} -FilePath .\ListeSvcRun.ps1
```

Répertoire : C:\Users\Kais

SignerCertificate	Status	Path
-----	-----	-----
BA88223AB3A0ADA10F044E48DB13B54E5A2DF657	Valid	ListeSvcRun.ps1

Le paramètre `-Certificate` spécifie le certificat qui sera utilisé pour signer le script `ListeSvcRun.ps1`. Quant au paramètre `-FilePath`, sa valeur est le chemin d'accès au script. Analysons maintenant le contenu du fichier que nous venons de signer numériquement :

```
PS> Get-Content .\ListeSvcRun.ps1
```

```
get-service | where-object {$_._Status -eq "Running"}
# SIG # Begin signature block
# MIIELgYJKoZIhvCNQcCoIEHzCBBsCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQABBAfzDtgWUsITrck0sYfvNR
# AgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQvnnCwiyoarE2KbiVRoc+cuGi
# d2KgggI4MIICNDCCaAaGgAwIBAgIQCQ8j2rQHS5BBXvKwTpZ9PTAJBgUrDgMCHQUA
# MCwxKjAoBgNVBAMTIVBvd2VyU2h1bGwgTG9jYWwgQ2VydG1maWNhGUGUm9vdDAe
# FwOxMzA1MTkyMjQ5MDhaFwOz0TEyMzEyMzU5NTlambUxEzARBgNVBAMTCkthaXMg
# QX1hcmkwgZ8wDQYJKoZIhvCNQEBBQADgY0AMIGJAoBAKiwGfBv3UWmcqKd/P1
# NKHPKKYfqtxQVm1dYqkZTu6TV9vL6BbpAv2q1EehJ801jdz1ZUndomzKUFyy1+
# 0cHAJkNPEjBtEVZJWs0rdQoXHwtI4CMB3HvBI2WViQKwtyIuQ/Fs6vxoA04L9wG/
# O27Y37XthzWL9Iy+NpArHFUbAgMBAAGjdjBOMBMA1UdjQQMMAoGCCsGAQUFBwMD
# MF0GA1UdAQRWMFSAEMiKQX8wUkt+fZeG0YqJVl6hLjAsMSowKAYDVQQDEyFQb3d1
# c1NoZWxsIEvxY2FsIEN1cnRpZmljYXR1IFJvb3SCELpiKM7D1taUREkC913f/tkw
# CQYFKw4DAh0FAAOBgQCooLipmEHQiXMbZmucu2heN1wnfcf7+ZSUtw79Vdj8EEMs
# jt1ozaimKujR9d+54vmsCaFroJ9haamH21LWVTAFBca70s2GQUwc2FbDXq09bkdk
# nRrkAGDfgXshd7f1PkDTZ7QI01FDLRkCBfoSVeqvalWAH1903GRi+SNhDZcdL0DGc
# AWwgxFcAgEBMEAwLDEqMCgGA1UEAxMhUG93ZXJTaGVsbCBMb2NhbCBDZXJ0aWZp
# Y2F0ZSBsB290AhAJDyPatAdLkEFe8pZ01n09MAkGSs0AwIaBQCgeDAYBgorBgEE
# AYI3AgEMMQuwCKACgAChAoAAMBkGCSqGSIB3DQEJAzEMBgorBgEEAYI3AgEEEMBwG
```

```
# CisGAQQBgcjCAQsxDjAMBgorBgEEAYI3AgEVCMGCSqGSIB3DQEJBDEWBBQiih/4
# 5KL246nzBxclMMvr1ry7BDANBgkqhkiG9w0BAQEFAASBgEOTYQDqSLUXw+u256rz
# 04QeR74dt40Qk+z+j6q3p2HCxFaSeSvRXRVdk9joR/Fjb7IR2SYYxUd5jzq0SJ30
# nwv/S1iHwgUthCqNKikmLmSzcr/mTAwCHLEq2X3Qr0+6CSYPodNLC1jzqJP+d05o
# On+Exh0WnKaJCOMGDR1nVEpH
# SIG # End signature block
```

Nous pouvons observer l'apparition d'un bloc de caractères n'ayant a priori aucun sens. En réalité, ce bloc constitue la signature numérique et est délimité par les balises `# SIG # Begin Signature Block` et `# SIG # End Signature Block`. Le script est donc signé et sa signature peut être vérifiée à l'aide de la cmdlet `Get-AuthenticodeSignature` :

```
PS> Get-AuthenticodeSignature -FilePath .\ListeSvcRun.ps1
Répertoire : C:\Users\Kais

SignerCertificate          Status      Path
-----          -----      -----
BA88223AB3A0ADA10F044E48DB13B54E5A2DF657  Valid      ListeSvcRun.ps1
```

Ici, la signature est valide, comme l'indique la propriété `Status`. Et cette validité correspond au contenu du script, qui ne doit pas être changé. Si nous modifions ne serait-ce qu'un seul caractère, alors l'intégrité du contenu du script serait remise en cause par PowerShell, qui refuserait dès lors son exécution. L'exemple qui va suivre reprend le précédent à la différence qu'au lieu de lister les services en cours d'exécution, nous listerons toujours dans le même script ceux qui ne le sont pas (ce qui amène à une modification du script) :

```
PS> Get-AuthenticodeSignature -FilePath .\ListeSvcRun.ps1
Répertoire : C:\Users\Kais

SignerCertificate          Status      Path
-----          -----      -----
BA88223AB3A0ADA10F044E48DB13B54E5..  HashMismatch  ListeSvcRun.ps1
```

La valeur du paramètre `Status` est `HashMismatch`, ce qui veut dire que l'empreinte calculée par PowerShell avant l'exécution du script ne correspond pas à celle figurant dans la signature numérique de ce dernier. Ceci est donc une invitation à vérifier le contenu du script, car il pourrait être dangereux.

**NOTE Concernant les modifications de fichiers signés**

Tout script ou fichier de configuration signé doit être à nouveau signé par son propriétaire en cas de modifications.

Le fait de pouvoir signer des scripts avec PowerShell représente donc un niveau supplémentaire dans le processus de mise en place d'une politique de sécurité. Et cette dernière doit être bien pensée et mise en perspective à la lumière des besoins de l'entreprise.

## PARTIE 2

# **PowerShell en tant que langage**



# 7

## Les variables

---

*L'utilisation des variables est peut-être la chose la plus partagée dans le monde de la programmation. En effet, elles symbolisent une multitude de types de valeurs qui sont susceptibles de changer au cours de l'exécution du programme. Et c'est là où réside d'ailleurs tout l'intérêt des variables : à un symbole particulier peuvent correspondre des valeurs qui ne sont pas figées dans le temps. Toutefois, la manifestation et l'utilisation d'une variable changent d'un langage à un autre et PowerShell n'échappe pas à cette règle.*

*Nous tenterons donc de définir ce qu'est une variable à partir du matériau qu'est PowerShell. Sur la base de cette compréhension, nous explorerons les différents types qu'elles peuvent revêtir.*

### SOMMAIRE

- ▶ Comprendre les variables
- ▶ Les variables d'environnement
- ▶ Les variables automatiques
- ▶ Les variables de préférence

## Définir ce qu'est une variable

Une variable est un emplacement mémoire identifié par un nom et dans lequel des valeurs sont stockées, qui peuvent être des résultats d'expressions, des chaînes de caractères, des nombres, etc.

En PowerShell, on désigne les variables par des chaînes de caractères commençant par le symbole dollar \$. La définition d'une variable donne ceci :

### Création d'une variable

```
$<Nom_Variable> = <Valeur_Variable>;
```

Contrairement à certains langages de programmation comme le C, il n'est pas nécessaire de déclarer la variable avant de l'utiliser. L'initialisation se fait donc au moment de sa création :

```
PS> $VarNum = 19;
PS> $VarNum
19
PS> $VarStr = 'Blue'
PS> $VarStr
Blue
PS> $VarProcs = Get-Process | Where-Object {$__.ProcessName -eq 'svchost'}
PS> $VarProcs
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
1782	58	16104	34480	141		112	svchost
703	35	8904	19348	102		320	svchost
383	13	2912	8844	37		776	svchost
468	14	3608	6868	27		824	svchost
875	101	16528	24388	112		912	svchost
949	41	71184	75704	159		928	svchost
749	37	11316	17292	1186		1212	svchost
473	38	18172	24424	367		1520	svchost
129	10	1864	6168	36		1920	svchost
97	8	1272	4408	22		2072	svchost
357	25	3740	8916	46		2176	svchost
390	28	6672	12196	883		3004	svchost

Ces exemples illustrent la pluralité des types rencontrés. Une variable peut en effet contenir n'importe quel type de données, contrairement à des langages comme VBScript où les valeurs ne sont constituées que de chaînes. Le nom d'une variable est constitué des caractères [A-Z] et [0-9] ainsi que du symbole \_ ; s'il contient des caractères spéciaux, alors il faudra les placer entre accolades {}.

```
PS> ${Special-Shell} = 1,2,3
PS> ${Special-Shell}
1
2
3
```

Les variables peuvent être utilisées en mode console, mais aussi dans les scripts. De plus, des cmdlets sont disponibles pour gérer les variables dans PowerShell.

- `New-Variable` : crée une variable.
- `Get-Variable` : renvoie les variables de la session actuelle.
- `Set-Variable` : attribue une valeur à une variable ou modifie une valeur existante.
- `Clear-Variable` : efface la valeur d'une variable sans supprimer la variable elle-même.
- `Remove-Variable` : supprime à la fois la variable et sa valeur.

PowerShell distingue plusieurs catégories de variables en plus de celles qui sont créées par l'utilisateur : les variables d'environnement, les variables automatiques et les variables de préférence.

## Les variables d'environnement

Les variables d'environnement dépassent l'horizon uniquement lié à PowerShell, car elles contiennent des informations qui concernent l'ensemble du système d'exploitation. Elles sont donc particulièrement utiles, car le système d'exploitation et l'ensemble des programmes en ont besoin au cours de leur propre exécution. Il peut s'agir d'informations en lien avec une architecture processeur, du nombre de processeurs, des chemins d'accès, c'est-à-dire finalement des informations fréquemment utilisées. Pour accéder aux variables d'environnement, PowerShell dispose d'un fournisseur nommé `Environment`.

### À SAVOIR Qu'est-ce qu'un fournisseur ?

Un fournisseur est un programme .Net qui lit les informations issues d'un magasin de données et qui les rend accessibles via un lecteur sous forme d'une arborescence ressemblant à celle d'un système de fichiers. Pour plus de détails, entrez en mode console et tapez la ligne de commande `Get-Help Get-PSPProvider`.

Les données en provenance de ce fournisseur sont lisibles à partir du lecteur `Env:`, qui permet de les lire à la manière d'un système de fichiers (même si ce fournisseur pourrait être considéré comme un espace de noms plat) :

```
PS> cd Env:  
PS Env:\> Get-ChildItem
```

Name	Value
---	----
ALLUSERSPROFILE	C:\ProgramData
AMDAPPSDKROOT	C:\Program Files (x86)\AMD APP\
APPDATA	C:\Users\Kais\AppData\Roaming
CommonProgramFiles	C:\Program Files\Common Files
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files
CommonProgramW6432	C:\Program Files\Common Files
COMPUTERNAME	LENDESK
ComSpec	C:\WINDOWS\system32\cmd.exe
configsetroot	C:\WINDOWS\ConfigSetRoot
FP_NO_HOST_CHECK	NO
GTK_BASEPATH	C:\Program Files (x86)\GtkSharp\2.12\
HOMEDRIVE	C:
HOME PATH	\Users\Kais
LOCALAPPDATA	C:\Users\Kais\AppData\Local
LOGONSERVER	\\\MicrosoftAccount
NUMBER_OF_PROCESSORS	2
OS	Windows_NT
Path	C:\Program Files (x86)\AMD APP\bin\x86_64;C:\P .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.W AMD64
PROCESSOR_ARCHITECTURE	AMD64 Family 20 Model 2 Stepping 0, AuthenticA 20
PROCESSOR_IDENTIFIER	0200
PROCESSOR_LEVEL	C:\ProgramData
PROCESSOR_REVISION	C:\Program Files
ProgramData	C:\Program Files (x86)
ProgramFiles	C:\Program Files
ProgramFiles(x86)	C:\Users\Kais\Documents\WindowsPowerShell\Modu C:\Users\Public
ProgramW6432	Console
PSModulePath	C:
PUBLIC	C:\WINDOWS
SESSIONNAME	C:\Users\Kais\AppData\Local\Temp
SystemDrive	C:\Users\Kais\AppData\Local\Temp
SystemRoot	LENDESK
TEMP	LENDESK
TMP	Kais
USERDOMAIN	C:\Users\Kais
USERDOMAIN_ROAMINGPROFILE	C:\Program Files\Oracle\VirtualBox\
USERNAME	c:\Program Files (x86)\Microsoft Visual Studio
USERPROFILE	C:\Program Files (x86)\Microsoft Visual Studio
VBOX_INSTALL_PATH	C:\WINDOWS
VS100COMNTOOLS	
VS110COMNTOOLS	
windir	

Les variables d'environnement sont sans doute les plus utilisées dans un système d'exploitation. En outre, elles sont un formidable moyen de communiquer des informations de manière conventionnelle.

## Les variables automatiques

Les variables automatiques sont en lien avec l'état de PowerShell. Par état, nous entendons tout ce qui, en termes d'informations, a trait au déroulement de l'exécution en mémoire de PowerShell. Ceci implique que PowerShell crée ces variables tout en modifiant certaines de leurs valeurs au fur et à mesure de sa propre exécution, afin de préserver une certaine précision de l'information. Le tableau 7-1 énumère les variables automatiques les plus communes.

**Tableau 7-1** Liste des variables automatiques les plus utilisées dans PowerShell

Variable Automatique	Description
\$\$	Contient le dernier jeton de la dernière ligne reçue au cours d'une session PowerShell.
\$?	Indique l'état d'exécution de la dernière ligne de commande. La variable contient la valeur <code>true</code> si l'opération s'est bien passée et <code>false</code> dans le cas contraire.
\$^	Contient le premier jeton de la dernière ligne reçue au cours d'une session PowerShell.
\$_	Contient l'objet actif de pipeline. Cette variable est peut-être la plus utilisée.
\$Args	Contient une liste de paramètres. Ces derniers, applicables aux fonctions, scripts et blocs de script, ont la particularité de ne pas être déclarés au sens traditionnel du terme.
\$ConsoleFileName	Indique le chemin d'accès d'un fichier console (avec l'extension <code>.psc1</code> ) utilisé en dernier au cours d'une session PowerShell. Ce fichier a la particularité de sauvegarder la configuration d'une session PowerShell.
\$Error	Contient une liste des erreurs rencontrées au cours de la session.
\$ExecutionContext	Contient des informations en lien avec le contexte d'exécution de PowerShell.
\$False	Contient la valeur <code>false</code> .
\$Home	Indique le chemin d'accès au répertoire de base de l'utilisateur.
\$Host	Représente l'application hôte PowerShell, c'est-à-dire que tout ce qui concerne l'hôte PowerShell est accessible en partie via cette variable.
\$LastExitCode	Contient la valeur ou le code de sortie de la dernière ligne de commande exécutée (quelle que soit la forme qu'elle prend).

**Tableau 7–1** Liste des variables automatiques les plus utilisées dans PowerShell (suite)

Variable Automatique	Description
\$Matches	Contient toutes les chaînes indiquant les correspondances quand les opérateurs <code>-match</code> et <code>-notmatch</code> sont utilisés.
\$MyInvocation	Contient des informations à propos de la ligne de commande actuelle.
\$NULL	Contient une valeur vide.
\$PID	Indique l'identifiant du processus représentant la session PowerShell actuelle.
\$Profile	Indique le chemin d'accès au profil PowerShell de l'utilisateur et de l'application PowerShell.
\$PSBoundParameters	Contient l'ensemble des paramètres actifs d'une fonction ou script et les valeurs qui leur sont associées. Il faut cependant remarquer que cela concerne les paramètres déclarés.
\$PSCmdlet	Représente une cmdlet ou fonction avancée en cours d'exécution.
\$PsCulture	Indique le nom de la culture utilisée dans le système d'exploitation.
\$PsHome	Indique le chemin d'accès au répertoire d'installation de PowerShell.
\$PsItem	Comme pour la variable automatique <code>\$_</code> .
\$PsVersionTable	Donne des détails concernant la version de PowerShell en cours d'exécution.
\$Pwd	Indique le chemin d'accès du répertoire actif.
\$True	Contient la valeur <code>true</code> .

La caractéristique la plus forte des variables automatiques est qu'elles sont d'une certaine manière associées au fonctionnement pur de PowerShell. Leur utilisation peut aider à réduire considérablement la verbosité des scripts, fonctions, mais aussi de n'importe quelle ligne de commande.

**NOTE Sur les variables automatiques**

Les variables automatiques ne peuvent pas être modifiées par les utilisateurs.

## Les variables de préférence

Il existe une autre catégorie de variables tout aussi importantes : les variables de préférence. Elles ont la particularité d'être liées à l'utilisateur et à ses préférences, c'est-à-dire à tout ce qui affecte l'environnement de PowerShell. Donc, contrairement aux variables automatiques, les variables de préférence peuvent être modifiées par les uti-

lisateurs tout en étant créées par PowerShell. Le tableau 7-2 fournit une liste de celles qui sont le plus utilisées.

**Tableau 7-2** Liste des variables de préférence les plus utilisées pour configurer PowerShell

Variable de préférence	Description
\$ConfirmPreference	Détermine si PowerShell demande automatiquement confirmation avant exécution d'une commande.
\$DebugPreference	Spécifie comment PowerShell répond aux messages de débogage engendrés par une cmdlet, un script ou un fournisseur.
\$ErrorActionPreference	Détermine comment PowerShell répond à une erreur ne provoquant pas l'arrêt de l'exécution d'un script ou d'une ligne de commande. Concerne aussi les fournisseurs.
\$ErrorView	Spécifie le format d'affichage des messages d'erreur pouvant survenir dans l'environnement PowerShell.
\$FormatEnumerationLimit	Indique le nombre d'éléments affichés au sein d'une propriété. La valeur par défaut est 4.
\$MaximumAliasCount	Détermine le nombre d'alias autorisés au cours d'une session. La valeur par défaut est 4 096.
\$MaximumDriveCount	Indique le nombre de lecteurs autorisés au cours d'une session. La valeur par défaut est 4 096.
\$MaximumErrorCount	Spécifie le nombre d'erreurs enregistrées dans l'historique. La valeur par défaut est 256.
\$MaximumHistoryCount	Spécifie le nombre de commandes enregistrées dans l'historique au cours d'une session. La valeur par défaut est 4 096.
\$OFS	Détermine le caractère séparant les éléments d'un tableau lorsque celui-ci est transposé en chaîne.
\$OutputEncoding	Détermine la méthode d'encodage de caractères utilisée par PowerShell lorsque celui-ci envoie du texte à d'autres applications.
\$PSEmailServer	Spécifie le serveur de messagerie utilisé par défaut pour envoyer des messages électroniques.
\$PSDefaultParameterValues	Détermine les valeurs par défaut des paramètres qui concernent les cmdlets et fonctions avancées.
\$PSModuleAutoLoadingPreference	Active ou désactive l'importation automatique de modules au cours d'une session PowerShell.
\$PSSessionConfigurationName	Indique le nom (sous forme d'URL) de la configuration de session PowerShell par défaut.
\$PSSessionOption	Détermine les options par défaut en ce qui concerne les sessions PowerShell à distance.
\$VerbosePreference	Indique comment PowerShell répond aux messages en lien avec le débogage d'une commande.

**Tableau 7–2** Liste des variables de préférence les plus utilisées pour configurer PowerShell (suite)

Variable de préférence	Description
\$WarningPreference	Indique comment PowerShell répond aux messages d'avertissement engendrés par les scripts, cmdlets et fournisseurs.
\$WhatIfPreference	Indique si le paramètre <code>-WhatIf</code> est activé automatiquement. Ne concerne que les commandes qui prennent en charge ce paramètre.

Il faut aussi noter que les variables de préférence affectent l'ensemble d'une session PowerShell, ce qui inclut évidemment les commandes. Cependant, les cmdlets disposent de paramètres dont le but est de spécifier le type de comportement qu'elles adoptent, c'est-à-dire finalement de rompre avec le mécanisme d'héritage issu de l'environnement d'exécution PowerShell.

# 8

## Les opérateurs

---

*PowerShell dispose d'un grand nombre d'opérateurs, éléments de langage servant à manipuler des valeurs. Ils peuvent être utilisés dans des contextes différents, que ce soit en mode console ou en mode scripting. Ils ont également la capacité de s'adapter à de multiples types d'objets.*

*Dans ce chapitre, nous étudierons les différents types d'opérateurs pris en charge par PowerShell : d'abord les opérateurs arithmétiques pour calculer des valeurs, puis les opérateurs logiques pour complexifier les filtres d'objets. Ensuite, nous comparerons des valeurs avec les opérateurs de comparaison, affecterons des valeurs aux variables avec les opérateurs d'affectation, redirigerons d'autres valeurs vers des fichiers texte avec les opérateurs de redirection, parlerons des opérateurs `split` et `join`, identifierons les types d'objets via les opérateurs de type et, enfin, nous nous intéresserons aux opérateurs spéciaux.*

### SOMMAIRE

- ▶ Les opérateurs arithmétiques
- ▶ Les opérateurs logiques
- ▶ Les opérateurs de comparaison
- ▶ Les opérateurs d'affectation
- ▶ Les opérateurs de redirection
- ▶ Les opérateurs de fractionnement et de jointure
- ▶ Les opérateurs de type
- ▶ Les opérateurs spéciaux

## Les opérateurs arithmétiques

Les opérateurs arithmétiques sont utiles pour calculer des valeurs numériques pouvant servir à l'exécution d'une ligne de commande ou d'un script. Il est possible d'ajouter, diviser, multiplier ou soustraire des valeurs. Le tableau suivant illustre l'ensemble des opérateurs arithmétiques pris en charge par PowerShell.

**Tableau 8-1** Opérateurs arithmétiques dans PowerShell

Opérateur	Description
+	Ajoute des entiers. Peut aussi concaténer des chaînes, des tableaux et des tables de hachage.
/	Divise deux valeurs.
*	Multiplie des entiers. Peut aussi copier des chaînes et des tableaux.
-	Soustrait une valeur d'une autre valeur. Peut aussi convertir un nombre en nombre négatif.
%	Renvoie comme résultat le reste d'une division.

Voici quelques exemples d'utilisation de ces opérateurs de base :

```
PS> 2+2
4
PS> 81/9
9
PS> 7*7
49
PS> 19-10
9
PS> 10%3
1
PS> 'Windows' + 'PowerShell'
Windows PowerShell
PS> 's' * 7
sssssss
```

Classiquement, ces opérateurs arithmétiques sont soumis à un ordre de précédence. PowerShell les traite dans l'ordre qui suit :

- 1 Les expressions à l'intérieur de parenthèses () .
- 2 - pour tout ce qui concerne les nombres négatifs.
- 3 \*, / et %.
- 4 + et - en ce qui concerne respectivement les additions et les soustractions.

**À SAVOIR Concernant la précédence des opérateurs**

PowerShell traite toujours les expressions entre parenthèses () en premier, et ce, quelles que soient les modalités.

Les exemples suivants illustrent comment s'appliquent les règles de précédence :

```
PS> 10+20/4*2
20
PS> 10+20/(4*2)
12,5
PS> (5*5)/(5*1)
5
PS> -(10+9)
-19
```

Les opérations arithmétiques s'exécutent aussi sur des objets de types différents. PowerShell détermine simplement l'opération à partir du type de l'objet situé le plus à gauche dans l'opération :

```
PS> 'Windows' + 7
Windows7
PS> 'Windows' * 2
WindowsWindows
PS> $var1 = "C:\"
PS> $var2 = "Devel\Sources"
PS> $var1 + $var2
C:\Devel\Sources
```

Les opérateurs arithmétiques + et - sont aussi utilisés pour incrémenter et décrémenter des valeurs en lien avec les variables et propriétés d'objets. Dans ce cas précis, il faut plutôt parler d'opérateurs unaires (voir section sur les opérateurs d'affectation) :

```
PS> $num = 2
PS> $num++
PS> $num
3
PS> $num = 8
PS> $num--
PS> $num
7
```

Ces opérateurs sont très puissants tant les capacités de polymorphisme sont grandes.

## Les opérateurs logiques

Avec PowerShell, il est possible de connecter des instructions conditionnelles afin de créer des filtres d'objets plus complexes. La possibilité de connecter des instructions les unes avec les autres donne la capacité de structurer une expression unique à partir de plusieurs instructions. Le tableau 8-2 montre les opérateurs logiques pris en charge par PowerShell.

**Tableau 8-2** Opérateurs logiques dans PowerShell

Opérateur	Description
-and	" ET logique ". Le résultat est vrai uniquement dans le cas où les deux instructions sont vraies.
-or	" OU logique ". Le résultat est vrai lorsqu'au moins une des deux instructions est vraie.
-xor	" OU exclusif logique ". Le résultat est vrai lorsque une et une seule des deux instructions est vraie.
-not	" NON logique ". L'instruction qui suit cet opérateur est niée. Si l'instruction est vraie, alors le résultat sera faux. Si l'instruction est fausse, alors le résultat sera vrai.
!	Autre forme du " NON logique ".

Les opérateurs logiques sont souvent utilisés pour tester la validité de plusieurs expressions. Dans cette perspective, le résultat renvoyé est :

- soit **True** : le résultat d'une opération est vrai ;
- soit **False** : le résultat d'une opération est faux.

PowerShell évalue les expressions nécessaires à l'évaluation du résultat final. Pour l'opérateur **-and**, si l'opérande gauche est évalué comme étant faux ①, alors l'opérande droit ne sera pas évalué. Pour l'opérateur **-or**, si l'opérande gauche est évalué comme étant vrai ②, alors l'opérande droit ne sera pas évalué :

```
PS> (2 -eq 2) -and (3 -eq 3)
True
PS> ① (2 -eq 1) -and (3 -eq 3)
False
PS> ② (2 -eq 2) -or (3 -eq 4)
True
PS> (2 -eq 2) -xor (3 -eq 4)
True
PS> -not (3 -eq 3)
False
PS> -not (3 -eq 2)
True
```

```
PS> $var1 = 20
PS> $var2 = 30
PS> ($var1 -lt $var2) -and ($var2 -ge $var1)
True
```

Les opérateurs logiques sont un excellent moyen pour créer des expressions complexes ; les filtres d'objets seront d'autant plus puissants, tout comme la flexibilité des commandes et des scripts.

## Les opérateurs de comparaison

PowerShell dispose d'opérateurs pour comparer deux valeurs, spécifiant des conditions particulières afin d'évaluer les expressions.

Tableau 8–3 Opérateurs de comparaison disponibles dans PowerShell

Opérateur	Description
-eq	Égal à.
-ne	Différent de.
-gt	Supérieur à.
-ge	Supérieur ou égal à.
-lt	Inférieur à.
-le	Inférieur ou égal à.
-like	Établit une correspondance en utilisant le caractère générique *.
-notlike	N'établit pas de correspondance à l'aide du caractère générique *.
-match	Établit une correspondance grâce aux expressions régulières.
-notmatch	S'assure de l'absence de correspondance avec une chaîne.
-contains	Vérifie qu'une valeur est contenue dans une ou des valeur(s) de référence.
-notcontains	Vérifie qu'une valeur n'est pas contenue dans une ou des valeur(s) de référence.
-replace	Remplace une valeur par une autre.
-in	Vérifie qu'une valeur est contenue dans une ou des valeur(s) de référence. Ressemble à l'opérateur -contains.
-notin	Vérifie qu'une valeur n'est pas contenue dans une ou des valeur(s) de référence. Ressemble à l'opérateur -notcontains.

## Les opérateurs d'égalité

Les opérateurs d'égalité (`-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`) sont utilisés pour évaluer des expressions et renvoyer comme résultat `true` ou `false` :

```
PS> 2 -eq 2
True
PS> 3 -ne 5
True
PS> 7 -lt 1
False
PS> 22 -ge 22
True
```

Lorsque l'entrée de l'opérateur, qui est la valeur située à gauche de ce dernier, représente une collection de valeurs, alors PowerShell renvoie les correspondances par rapport à un modèle spécifié à droite de l'opérateur :

```
PS> 'Windows', 'PowerShell' -eq 'PowerShell'
PowerShell
PS> '123', '456', '789' -eq '789'
789
```

## Les opérateurs de relation

Les opérateurs de relation (`-contains` et `-notcontains`) fonctionnent comme les opérateurs d'égalité, à l'exception qu'ils renvoient toujours une valeur booléenne :

```
PS> 1,2,3,4,5 -contains 4
True
PS> 'Devel' -contains 'el'
False
PS> '123', '456', '789' -contains '789'
True
PS> 'abc', 'def', 'ghi' -contains 'xyz'
False
```

## Les opérateurs de correspondance

Les opérateurs de correspondance (`-match` et `-notmatch`) servent à rechercher des éléments issus d'un modèle structuré à partir d'expressions régulières. En outre, la recherche ne peut se faire que dans les chaînes :

```
PS> 'PowerShell' -match 'Power'  
True  
PS> 'Linux' -match 'Windows'  
False
```

La variable automatique `$Matches` est remplie lorsque l'entrée de l'opérateur est une valeur scalaire unique. Lorsqu'il s'agit d'une collection, les opérateurs de correspondance renvoient comme résultat des valeurs issues de la collection et la variable `$Matches` n'est pas remplie :

```
PS> 'PowerShell', 'Perl', 'Python' -match 'Python'  
Python  
PS> '12', '23', '34' -match '23'  
23
```

## L'opérateur `-replace`

Pour remplacer une valeur par une autre, on emploie l'opérateur `-replace` :

```
PS> 'Windows PowerShell' -replace 'PowerShell', 'Monad'  
Windows Monad  
PS> 'BlueCurve' -replace 'Blue', 'Red'  
RedCurve
```

On remarque qu'il a deux arguments. Voici comment se présente sa syntaxe.

### Syntaxe de l'opérateur de remplacement `-replace`

```
<entrée> <op.de remplacement> <ch. à remplacer>, <ch. de remplacement>
```

## Les opérateurs `-like` et `-notlike`

Les opérateurs `-like` et `-notlike` établissent des correspondances à l'aide du symbole `*` :

```
PS> 'PowerShell' -like '*Shell'  
True  
PS> 'PowerShell' -notlike 'Power*'  
False
```

## Les opérateurs -in et -notin

Les opérateurs `-in` et `-notin` ont été ajoutés dans PowerShell version 3. Ils ressemblent beaucoup aux opérateurs de relation en ce sens qu'ils peuvent nous dire, par exemple, si une valeur d'entrée se trouve dans une collection. La différence essentielle réside dans le fait qu'ici, la valeur de test est à gauche de l'opérateur, alors que cette même valeur se trouve à droite de l'opérateur lorsqu'il s'agit des opérateurs de relation :

```
PS> 'Perl' -in 'PowerShell', 'Perl', 'Python'
True
PS> 'Lisp' -in 'PowerShell', 'Perl', 'Python'
False
```

Par défaut, aucun opérateur de comparaison ne respecte la casse. Pour forcer un opérateur à la respecter, il faut faire précéder son nom de la lettre `c`. Au contraire, pour le forcer explicitement à ne pas la respecter, il faut faire précéder son nom de la lettre `i`.

## Les opérateurs d'affectation

PowerShell fournit des opérateurs pour affecter, modifier ou même ajouter des valeurs aux variables.

**Tableau 8–4** Liste des opérateurs d'affectation dans PowerShell

Opérateur	Description
<code>=</code>	Affecte une valeur à une variable.
<code>+=</code>	Incrémente ou ajoute une valeur spécifiée à une variable.
<code>-=</code>	Décrémente la valeur d'une variable.
<code>*=</code>	Multiplie la valeur d'une variable.
<code>/=</code>	Divise la valeur d'une variable.
<code>%=</code>	Divise la valeur d'une variable et lui affecte le reste.
<code>++</code>	Incrémente d'une unité la valeur d'une variable.
<code>--</code>	Décrémente d'une unité la valeur d'une variable.

Si nous voulons affecter une valeur à une variable, il faut utiliser l'opérateur `=` :

```
PS> $num = 7
PS> $num
7
```

Ici, affectation veut dire création, c'est-à-dire que la valeur affectée implique le processus de création de la variable, sauf si elle existe déjà. De plus, tout type d'objet .Net peut être affecté comme valeur de la variable, par exemple un tableau :

```
| PS> $var = "Perl", "Python", "Lisp"
```

ou alors un dictionnaire :

```
| PS> $var = @{1="one";2="two";3="three"}
```

Il est aussi possible d'incrémenter la valeur d'une variable à l'aide de l'opérateur `+=`, qui ajoute d'abord pour ensuite affecter :

```
| PS> $num = 2
| PS> $num += 2
| PS> $num
4
```

Si la variable contient une chaîne, la valeur spécifiée est ajoutée :

```
| PS> $var = "Windows"
| PS> $var += " Blue"
| PS> $var
Windows Blue
```

De façon similaire, on décrémente la valeur d'une variable avec l'opérateur `-=`, qui soustrait d'abord pour ensuite affecter :

```
| PS> $num = 2
| PS> $num -= 1
| PS> $num
1
```

L'opérateur `*=` multiplie la valeur d'une variable. Lorsque cette dernière est numérique, PowerShell la multiplie par la valeur spécifiée. Dans le cas où c'est une chaîne, PowerShell duplique la chaîne autant de fois que spécifié :

```
| PS> $num = 2
| PS> $num *= 2
| PS> $num
4
| PS> $var = "Psx"
| PS> $var *= 4
| PS> $var
PsxPsxPsxPsx
```

L'opérateur `/=` divise une valeur purement numérique :

```
PS> $num = 4
PS> $num /= 2
PS> $num
2
```

Pour affecter le reste d'une division (ou modulo) à une variable, on utilise l'opérateur `%=` :

```
PS> $num1 = 10
PS> $num2 = 3
PS> $num1 %= $num2
1
```

Les opérateurs `++` et `--` sont aussi très utiles si nous voulons respectivement augmenter ou diminuer d'une unité la valeur d'une variable :

```
PS> $num = 9
PS> $num++
PS> $num
10
PS> $num = 9
PS> $num--
PS> $num
8
```

Ces opérateurs disposent de versions préfixées et suffixées, c'est-à-dire qu'ils peuvent être placés avant ou après une variable. La version préfixée augmente ou diminue la valeur d'une variable *avant* que celle-ci ne soit utilisée dans une instruction :

```
PS> $num1 = 18
PS> $num2 = ++$num1
PS> $num2
19
PS> $num1
19
```

La version suffixée augmente ou diminue la valeur d'une variable *après* que celle-ci a été utilisée dans une instruction :

```
PS> $num1 = 18
PS> $num2 = $num1--
PS> $num2
18
```

```
PS> $num1  
17
```

PowerShell permet d'affecter des valeurs à plusieurs variables en même temps :

```
PS> $a, $b, $c = 10, 11, 12
```

Dans cet exemple, la valeur **10** est affectée à la variable **\$a**, la valeur **11** à **\$b** et la valeur **12** à **\$c**. Nous pouvons également affecter une seule valeur à de multiples variables en même temps :

```
PS> $a = $b = $c = 999
```

Les opérateurs d'affectation sont donc incontournables pour manipuler les variables. Les connaître est un plus non négligeable.

## Les opérateurs de redirection

Lorsqu'un utilisateur exécute une commande en console, PowerShell redirige la sortie vers cette même console. Ce type de redirection est un comportement par défaut commun à tous les interpréteurs de commandes. Il faut cependant noter qu'il est possible de recourir à d'autres types de redirections, par exemple vers un fichier texte, ou même de ne rediriger que les erreurs vers un fichier texte. Toutes ces possibilités sont symbolisées par les caractères suivants :

- **\*** : représente tout type de sortie.
- **1** : réussite.
- **2** : erreurs.
- **3** : messages d'avertissement.
- **4** : messages commentés.
- **5** : informations en lien avec le processus de débogage.

### NOTE Concernant les opérateurs de redirection

Les opérateurs **\***, **3**, **4** et **5** ont été ajoutés dans la version 3 de PowerShell.

PowerShell prend en charge un certain nombre d'opérateurs de redirection permettant de manipuler les différents flux de sortie.

**Tableau 8–5** Liste des opérateurs de redirection dans PowerShell

Opérateur	Description
>	Envoie la sortie à un fichier.
>>	Incrémente le contenu d'un fichier existant de la sortie spécifiée.
2>	Envoie la sortie d'erreurs à un fichier.
2>>	Incrémente le contenu d'un fichier existant de la sortie d'erreurs.
2>&1	Envoie la sortie d'erreurs ainsi que la sortie de réussite au flux de sortie de réussite.
3>	Envoie la sortie contenant les messages d'avertissement au fichier spécifié.
3>>	Incrémente le contenu d'un fichier existant de la sortie contenant des messages d'avertissement.
3>&1	Envoie la sortie contenant des messages d'avertissement ainsi que la sortie de réussite au flux de sortie de réussite.
4>	Envoie la sortie contenant les messages commentés au fichier spécifié.
4>>	Incrémente le contenu d'un fichier existant de la sortie contenant des messages commentés.
4>&1	Envoie la sortie contenant des messages commentés ainsi que la sortie de réussite au flux de sortie de réussite.
5>	Envoie la sortie contenant les messages en lien avec le traitement des commandes pour le débogage au fichier spécifié.
5>>	Incrémente le contenu d'un fichier existant de la sortie contenant des messages en lien avec le traitement des commandes pour le débogage.
5>&1	Envoie la sortie contenant des messages en lien avec le traitement des commandes pour le débogage ainsi que la sortie de réussite au flux de sortie de réussite.
*>	Envoie tout type de sortie à un fichier.
*>>	Incrémente le contenu d'un fichier existant de tout type de sortie.
*>&1	Envoie tout type de sortie au flux de sortie de réussite.

Tous ces opérateurs sont là pour nous aider à maîtriser les flux de sortie et peuvent répondre aux exigences de contextes variés. L'exemple suivant illustre la redirection de sortie vers un fichier texte :

```
PS> Get-Service | Select-Object -First 10 > 10Procs.txt
```

Lorsque la sortie est redirigée vers un fichier texte, rien n'est affiché à l'écran. Donc, si nous voulons analyser la sortie, il faut lire le fichier :

```
PS> type .\10Procs.txt
```

Status	Name	DisplayName
Running	AdobeARMservice	Adobe Acrobat Update Service
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service
Stopped	AeLookupSvc	Expérience d'application
Stopped	ALG	Service de la passerelle de la couc...
Stopped	AllUserInstallA...	Agent d'installation pour tous les ...
Running	AMD External Ev...	AMD External Events Utility
Running	AMD FUEL Service	AMD FUEL Service
Stopped	AppIDSvc	Identité de l'application
Stopped	Appinfo	Informations d'application
Stopped	aspnet_state	ASP.NET State Service

Le fichier `10Procs.txt` a été créé dynamiquement par PowerShell. Essayons à présent d'y ajouter des données via l'opérateur `>>` :

```
PS> Write-Output "Voici une liste de 10 services..." >> .\10Procs.txt
PS> type .\10Procs.txt
```

Status	Name	DisplayName
Running	AdobeARMservice	Adobe Acrobat Update Service
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service
Stopped	AeLookupSvc	Expérience d'application
Stopped	ALG	Service de la passerelle de la couc...
Stopped	AllUserInstallA...	Agent d'installation pour tous les ...
Running	AMD External Ev...	AMD External Events Utility
Running	AMD FUEL Service	AMD FUEL Service
Stopped	AppIDSvc	Identité de l'application
Stopped	Appinfo	Informations d'application
Stopped	aspnet_state	ASP.NET State Service

Voici une liste de 10 services...

Certains utilisateurs préfèrent rediriger uniquement la sortie d'erreurs vers un fichier texte :

```
PS> Get-Process -Name Wide 2> Errors.txt
PS> type .\Errors.txt
```

```
Get-Process : Impossible de trouver un processus nommé «Wide». Vérifiez
le nom du processus et appelez de nouveau l'applet de commande.
Au caractère Ligne:1 : 1
+ Get-Process -Name Wide 2> Errors.txt
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Wide:String)
[Get-Process], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft
.PowerShell.Commands.GetProcessCommand
```

D'autres veulent avoir à la fois les sorties d'erreurs et de réussites à l'écran :

```
PS> Get-Service -Name Wide, WinRM 2>&1

Get-Service : Impossible de trouver un service assorti du nom « Wide ».
Au caractère Ligne:1 : 1
+ Get-Service -Name Wide, WinRM 2>&1
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Wide:String)
[Get-Service], ServiceCommandException
+ FullyQualifiedErrorId : NoServiceFoundForGivenName,Microsoft
.PowerShell.Commands.GetServiceCommand

Status    Name           DisplayName
-----   --  -----
Stopped  WinRM          Gestion à distance de Windows (Gest...
```

Si, pour une raison ou pour une autre, l'utilisateur ne souhaite pas voir la sortie d'erreurs, il peut la rediriger vers la variable `$null` :

```
PS> Get-Service -Name Wide, WinRM 2>$null

Status    Name           DisplayName
-----   --  -----
Stopped  WinRM          Gestion à distance de Windows (Gest...
```

#### ATTENTION Redirection vers un fichier existant

Les opérateurs de redirection (`>` et `>>`) n'ajoutent pas de données si le fichier spécifié existe déjà, mais le remplacent sans avertissement. S'il s'agit d'un fichier masqué, système ou en lecture seule, le processus de redirection échouera.

La redirection de sortie peut aussi être faite via la cmdlet `Out-File`, qui sait entre autres forcer la redirection de contenu vers des fichiers masqués, système ou en lecture seule. En outre, contrairement aux opérateurs de redirection qui n'utilisent que l'Unicode, `Out-File` permet un mode d'encodage différent. Voici comment utiliser cette cmdlet :

```
PS> Get-Service -Name Wide, WinRM 2>&1 | Out-File -FilePath .\Sortie.txt
```

Le processus de redirection des informations est donc, comme nous venons de le voir, un processus riche et complexe à la fois. Savoir comment contrôler les flux d'objets est un élément essentiel à l'utilisation de PowerShell.

## Les opérateurs de fractionnement et de jointure

Depuis le début du chapitre, les types d'opérateurs que nous avons étudiés manipulent plusieurs types de données, qu'il s'agisse de valeurs numériques, de chaînes ou même de collections. Dans cette section, nous mettrons l'accent sur deux opérateurs dont le dessein est de manipuler uniquement des chaînes.

### L'opérateur `-split`

Il est parfois nécessaire de fractionner des chaînes en sous-chaînes. PowerShell propose pour ce faire l'opérateur `-split`. L'exemple suivant montre le fractionnement d'une chaîne au niveau de l'espace, qui est le délimiteur par défaut :

```
PS> -split "Windows Workflow Foundation"
Windows
Workflow
Foundation
```

Il est bien sûr possible de fractionner une chaîne sur la base d'un délimiteur différent :

```
PS> "Windows:Workflow:Foundation" -split ':'
Windows
Workflow
Foundation
```

Notez les différentes positions de l'opérateur `-split`. Nous avons utilisé l'opérateur `-split` unaire dans le premier exemple et l'opérateur `-split` binaire dans le second. Il est préférable d'opter pour ce dernier, car il offre plus de possibilités, par exemple le contrôle du nombre de sous-chaînes :

```
PS> '1,2,3,4,5,6,7,8,9' -split ',',4
1
2
3
4,5,6,7,8,9
```

Ici, l'opérateur `-split` a deux arguments : le délimiteur et le nombre de sous-chaînes que l'on souhaite extraire. La série de nombres est transtypée en valeur chaîne avant son traitement. Parmi les arguments possibles, on peut spécifier un bloc de script afin de définir le délimiteur à partir de bases plus complexes :

```
PS> 'Diogène,Spinoza,Schopenhauer' -split {$_ -eq 'n' -or $_ -eq 'e'}
Diogè
,Spi
oza,Schop
hau
r
```

**ALLER PLUS LOIN Options de l'opérateur `-split`**

`Get-Help about_Split`

## L'opérateur `-join`

Nous allons à présent étudier l'opération inverse consistant à combiner plusieurs sous-chaînes en une seule à l'aide de l'opérateur de jointure `-join`. Ce dernier dispose, tout comme l'opérateur `-split`, d'une forme unaire et d'une forme binaire :

```
PS> -join ('x','y','z')
xyz
PS> 'Windows','PowerShell' -join ':'
Windows:PowerShell
PS> 'Windows','PowerShell' -join '0x3'
Windows0x3PowerShell
```

Notez dans le premier exemple l'apparition de parenthèses au niveau de l'argument. Ceci est lié au fonctionnement de PowerShell. Il se trouve qu'ici une règle s'applique (qui est la même pour `-split`) : l'opérateur de jointure unaire `-join` a la priorité sur une virgule, ce qui a comme conséquence de forcer PowerShell à ignorer toutes les chaînes après la virgule ; ainsi, seule la première chaîne est soumise à l'opérateur. Dans ce cas de figure, il faut placer toutes les chaînes entre parenthèses pour que l'argument devienne une collection.

Les autres exemples utilisent la forme binaire de l'opérateur, qui est à privilégier ne serait-ce que pour la marge de manœuvre offerte.

## Les opérateurs de type

PowerShell structure chaque objet sur la base d'un type. Il est donc important pour les utilisateurs qui manipulent ces objets au quotidien de pouvoir déterminer la

nature d'un type d'objet. Le tableau 8-6 liste les opérateurs de type existant dans PowerShell.

**Tableau 8-6** Opérateurs de type pris en charge par PowerShell

Opérateur	Description
<code>-is</code>	Vérifie que le type .NET spécifié correspond à celui de l'objet d'entrée. Dans ce cas, retourne la valeur <code>true</code> .
<code>-isnot</code>	Vérifie que le type .NET spécifié ne correspond pas à celui de l'objet d'entrée. Dans ce cas, retourne la valeur <code>true</code> .
<code>-as</code>	Convertit l'objet d'entrée dans le type .NET spécifié.

La syntaxe des opérateurs de type est assez particulière.

#### Syntaxe des opérateurs de type

```
| <entrée> <opérateur de type> [type .Net]
```

ou :

```
| <entrée> <opérateur de type> "type .Net"
```

Les exemples qui suivent montrent de quelle manière les opérateurs de type `-is` et `-isnot` doivent être utilisés. Ils agissent de manière très complémentaire :

```
PS> 19 -is [int]
True
PS> 19 -is "int"
True
PS> "23/04/1998" -is [DateTime]
False
PS> "23/04/1998" -is [String]
True
PS> (Get-ChildItem)[0] -is [IO.DirectoryInfo]
True
PS> "23/04/1998" -isnot [DateTime]
True
PS> 19 -isnot [float]
True
```

L'opérateur de type `-as` se démarque quelque peu des deux autres en ce sens qu'il n'est pas de type booléen et que son but est de convertir des objets :

```
PS> "23/04/1998" -as [DateTime]
jeudi 23 avril 1998 00:00:00

PS> '999' -as [int]
999
PS> 1033 -as [System.Globalization.CultureInfo]

LCID          Name          DisplayName
----          ----          -----
1033          en-US         Anglais (États-Unis)
```

Les opérateurs de type sont très intéressants dans le cadre de l'écriture de scripts où les types d'objets passés en paramètres ont une grande importance.

## Les opérateurs spéciaux

PowerShell dispose par ailleurs d'opérateurs dits spéciaux, qui effectuent un certain nombre d'opérations que les autres opérateurs ne peuvent pas faire.

### L'opérateur d'appel

Il est possible d'exécuter une commande, un script ou un bloc de script à l'aide de l'opérateur d'appel `&`. Dans ce cas de figure, PowerShell analyse l'opérateur et utilise la valeur passée en argument de cet opérateur pour définir le nom de la commande à exécuter :

```
PS> $cmd = "Get-Date"
PS> & $cmd
vendredi 31 mai 2013 10:25:53
```

Ici, nous avons créé une variable de type chaîne et utilisé l'opérateur d'appel `&` afin d'exécuter la commande.

### L'opérateur de déréférencement

Cet opérateur (symbolisé par un signe `.`) donne accès aux propriétés ainsi qu'aux méthodes d'un objet :

```
PS> $obj = "string"
PS> $obj.length
PS> $obj.ToUpper()
```

## L'opérateur de type « dot sourcing »

Cet opérateur (symbolisé aussi par un signe `.`) permet d'exécuter un script afin que tous les éléments qu'il contient soient accessibles à la portée appelante :

```
PS> . MyScripts.ps1
```

ALLER PLUS LOIN Pour plus de détails sur les portées

Get-Help about\_scope

## L'opérateur de membre statique

L'opérateur de membre statique (symbolisé par les signes `::`) sert à invoquer les propriétés et méthodes statiques d'un objet .Net :

```
PS> [DateTime]::Today
```

## L'opérateur de plage

L'opérateur de plage (symbolisé par les signes `..`) représente un ensemble de nombres entiers séquentiels dans un tableau de nombre entiers ayant une limite inférieure et une limite supérieure :

```
PS> 1..7
1
2
3
4
5
6
7
```

## L'opérateur de mise en forme

L'opérateur de mise en forme (symbolisé par `-f`) sert à configurer l'aspect des chaînes. Les directives doivent figurer à gauche de l'opérateur et la chaîne à mettre en forme à sa droite :

```
PS> "{0,-10} {1}" -f 1,"PowerShell"
1      PowerShell
```

**ALLER PLUS LOIN** **Plus de détails concernant les directives de mise en forme**

► <http://msdn.microsoft.com/en-us/library/system.string.format.aspx>

## L'opérateur de sous-expression

L'opérateur de sous-expression (symbolisé par `$()`) retourne le résultat d'une ou de plusieurs expression(s). Cela implique que, si le résultat est unique, alors il s'agit d'une valeur scalaire. Enfin, si le résultat est composite, alors il s'agit d'un tableau de valeurs :

```
| PS> $(Get-Date)  
| PS> $(get-process)
```

## L'opérateur de sous-expression de tableau

L'opérateur de sous-expression de tableau (symbolisé par `@()`) retourne le résultat d'une ou de plusieurs expression(s) sous forme de tableau, et ce, même si le tableau ne contient qu'un seul élément :

```
| PS> @(Get-Service)
```

## L'opérateur virgule

L'opérateur virgule (symbolisé par `,`) a deux formes.

- Une forme unaire crée un tableau avec un membre.
- Une forme binaire crée un tableau avec plusieurs membres.

En voici deux exemples :

```
| PS> $Unaire = ,3  
| PS> $Binaire = 1,2,3
```

## L'opérateur de pipeline

L'opérateur de pipeline (symbolisé par le signe `|`) envoie la sortie d'une commande vers l'entrée d'une autre :

```
| PS> Get-Service -Name "WinRM" | Stop-Service
```

## L'opérateur de transtypage

L'opérateur de transtypage (symbolisé par `[]`) convertit un objet d'un type en un autre :

```
PS> [DateTime]$Date = "07/05/1995"
PS> $Date
mercredi 7 mai 1995 00:00:00
```

## L'opérateur d'index

L'opérateur d'index (symbolisé aussi par `[]`) permet de sélectionner des valeurs précises d'une collection, qu'elle soit un tableau ou un dictionnaire. Le premier objet d'un tableau a `0` comme valeur d'index :

```
PS> $Tableau = a,b,c,d
PS> $Tableau[0]
a
```

Un dictionnaire est quant à lui indexé sur la base de ses différentes clés :

```
PS> $Hash = @{Nom="PowerShell";Version="3.0"}
PS> $Hash["Nom"]
PowerShell
```



# 9

## Les tableaux et dictionnaires

---

*Manipuler des collections d'objets est récurrent lorsque l'on utilise PowerShell. Une collection, qui est par définition un groupe d'objets de même type ou non, est par essence une structure extrêmement flexible, que l'utilisateur sera amené à utiliser aussi bien en mode console qu'en mode scripting. L'objectif de ce chapitre est de se familiariser avec les différents types de collections que l'on peut trouver dans PowerShell : les tableaux et les dictionnaires.*

### SOMMAIRE

- ▶ Les tableaux
- ▶ Les dictionnaires

## Les tableaux

Un tableau est une structure de données plus ou moins complexe capable de stocker des données qui ne sont pas forcément du même type. On peut donc concevoir un tableau comme une sorte de conteneur de valeurs ou d'objets.

### Créer un tableau

Créer un tableau avec PowerShell est très simple : il suffit d'affecter plusieurs valeurs à une variable. Par exemple :

```
| PS> $Tableau = 1,2,3,4,5
```

Ici, la variable `$Tableau` contient cinq valeurs numériques. Si l'on exécute cette variable, voici le résultat produit :

```
| PS> $Tableau
1
2
3
4
5
```

Il est aussi possible d'utiliser l'opérateur de plage (...) pour initialiser un tableau :

```
| PS> $Tableau = 1..5
| PS> $Tableau
1
2
3
4
5
```

Le tableau que nous venons de créer n'a pas de type spécifique, ce que nous pouvons vérifier avec cette ligne de commande :

```
| PS> $Tableau.GetType()
IsPublic IsSerial Name                                     BaseType
-----  -----  --Object[]                                 System.Array
```

La colonne `Name` a comme valeur `Object[]`. Cela signifie que la nature des objets composant le tableau est générique et non pas spécifique. Pour que le tableau soit fortement typé, l'opérateur de transtypage (`[]`) doit précéder le nom de la variable :

```
PS> [int32[]]$Tableau = 1,2,3,4,5
PS> $Tableau.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----  -- Int32[]                                 System.Array
```

Dans ce cas, aucune donnée contenue dans le tableau ne pourra être d'un autre type. L'exemple précédent ne peut contenir que des entiers 32 bits et rien d'autre. Par conséquent, tout dépend de ce que l'on veut faire.

Un tableau peut être initialisé avec la sortie d'une commande ou instruction. L'exemple suivant montre que la sortie de la commande `Get-Service` est encapsulée dans la variable `$Services` qui de fait est un tableau de valeurs :

```
PS> $Services = Get-Service
```

L'opérateur de sous-expression de tableau peut évidemment être sollicité dans ce contexte :

```
PS> $Services = @(Get-Service)
```

Les deux possibilités que nous venons d'évoquer mènent au même résultat. La différence est que l'opérateur de sous-expression forcera PowerShell à créer un tableau, même si ce dernier ne contient qu'une seule valeur, voire aucune :

```
PS> $c = @(1)
PS> $c.Count
1
PS> $x = @()
PS> $x.Count
0
```

## Utiliser les tableaux

Un tableau est référencé par son nom de variable. Appeler le nom d'une variable contenant un tableau entraîne l'affichage de tous les éléments de ce dernier :

```
PS> $Tab = 1,2,3,4,5,6,7
PS> $Tab
1
2
3
```

```
4  
5  
6  
7
```

La référence aux éléments d'un tableau se fait selon le principe de l'indexation. Le premier élément d'un tableau est lié à la position d'index (0) :

```
PS> $Tab[0]  
1
```

Le deuxième élément d'un tableau est quant à lui lié à la position d'index (1) :

```
PS> $Tab[1]  
2
```

Pour faire référence au dernier élément d'un tableau, nous pouvons utiliser le chiffre négatif (-1) :

```
PS> $Tab[-1]  
7
```

Pour faire référence à l'avant-dernier élément d'un tableau, c'est le chiffre -2 qu'il faut indiquer :

```
PS> $Tab[-2]  
6
```

La référence à plusieurs éléments d'un tableau peut se faire via l'opérateur de plage :

```
PS> $Tab[0..4]  
1  
2  
3  
4  
5
```

Chaque élément d'un tableau est évidemment modifiable et le processus de modification se fait entre autres à l'aide de l'opérateur d'affectation (=). Par exemple, essayons de changer l'avant-dernier élément du tableau \$Tab :

```
PS> $Tab[5] = 33  
PS> $Tab[5]  
33
```

Jusqu'ici, nous n'avons manipulé le tableau que par l'intermédiaire de l'opérateur d'index (`[]`). En effet, un tableau est un type en tant que tel et un type est composé de membres. Pour les découvrir, utilisons la cmdlet `Get-Member` :

```
PS> Get-Member -InputObject $Tab
```

TypeName : System.Object[]

Name	MemberType	Definition
Count	AliasProperty	Count = Length
Add	Method	int IList.Add(System.Object value)
Address	Method	System.Object&, mscorlib, Version=4.
Clear	Method	void IList.Clear()
Clone	Method	System.Object Clone(), System.Object
CompareTo	Method	int IStructuralComparable.CompareTo(
Contains	Method	bool IList.Contains(System.Object va
CopyTo	Method	void CopyTo(array array, int index),
Equals	Method	bool Equals(System.Object obj), bool
Get	Method	System.Object Get(int )
GetEnumerator	Method	System.Collections.IEnumerator GetEnumerator()
GetHashCode	Method	int GetHashCode(), int IStructuralEq
GetLength	Method	int GetLength(int dimension)
GetLongLength	Method	long GetLongLength(int dimension)
GetLowerBound	Method	int GetLowerBound(int dimension)
GetType	Method	type GetType()
GetUpperBound	Method	int GetUpperBound(int dimension)
GetValue	Method	System.Object GetValue(Params int[])
IndexOf	Method	int IList.IndexOf(System.Object value)
Initialize	Method	void Initialize()
Insert	Method	void IList.Insert(int index, System.
Remove	Method	void IList.Remove(System.Object value)
RemoveAt	Method	void IList.RemoveAt(int index)
Set	Method	void Set(int , System.Object )
<b>SetValue</b>	Method	void SetValue(System.Object value, int index)
ToString	Method	string ToString()
Item	ParameterizedProperty	System.Object IList.Item(int index)
IsFixedSize	Property	bool IsFixedSize {get;}
IsReadOnly	Property	bool IsReadOnly {get;}
IsSynchronized	Property	bool IsSynchronized {get;}
Length	Property	int Length {get;}
LongLength	Property	long LongLength {get;}
Rank	Property	int Rank {get;}
SyncRoot	Property	System.Object SyncRoot {get;}

On peut observer la présence de membres très intéressants, comme la méthode `SetValue` qui sert à modifier une valeur. À partir de ces informations, tentons de modifier la valeur du troisième élément du tableau :

```
PS> $Tab.SetValue(99,2)
PS> $Tab[2]
99
```

On peut même connaître le nombre exact d'éléments que le tableau contient :

```
PS> $Tab.Length
7
```

PowerShell laisse une grande marge de manœuvre quant à l'utilisation des tableaux. Donc, maîtriser l'ensemble des possibilités offertes en la matière peut demander au moins plusieurs heures.

## Les dictionnaires

Un dictionnaire est semblable à un tableau, c'est-à-dire une structure de données, à la différence qu'un dictionnaire s'articule autour d'au moins une paire clé/valeur. Ce type de structure est sans nul doute la collection la plus flexible et la plus cohérente. La syntaxe d'un dictionnaire est la suivante.

### Syntaxe d'un dictionnaire

```
① @{ <clé> = <valeur> ; [ <clé> = <valeur> ]... } ②
```

Donc, un dictionnaire commence par les symboles `@{` ① et se termine par le symbole `}` ②. L'utilisation des dictionnaires est fortement recommandée lorsqu'il s'agit de définir un ensemble de données compact. De cette façon, la recherche de ces données sera plus aisée.

#### NOTE Concernant les dictionnaires

Les dictionnaires sont aussi connus sous le nom de tableaux associatifs ou tables de hachage.

## Créer un dictionnaire

Comme évoqué plus haut, un dictionnaire s'articule en paires clé/valeur de sorte qu'une valeur soit identifiée par une clé. Prenons un exemple :

```
[Clé]DC2012SRV = [Valeur]10.0.2.48
```

Ici, la valeur 10.0.2.48 est associée à la clé DC2012SRV. Si nous traduisons ceci en termes concrets, cela donne :

```
PS> $Hash = @{DC2012SRV = "10.0.2.48"}
```

En affichant notre dictionnaire, on peut distinguer deux propriétés.

- `Name` correspond à la clé.
- `Value` correspond à la valeur associée à ladite clé.

Afficher un dictionnaire dans sa totalité consiste à appeler le nom de la variable qui le représente :

```
PS> $Hash
```

Name	Value
---	-----
DC2012SRV	10.0.2.48

Le dictionnaire que nous venons de créer ne comporte qu'une seule paire clé/valeur. Pour en ajouter une autre, il faut insérer un point-virgule de séparation (;) :

```
PS> $Hash = @{DC2012SRV = "10.0.2.48" ; DC2008SRV = "10.0.2.33"}  
PS> $Hash
```

Name	Value
---	-----
DC2012SRV	10.0.2.48
DC2008SRV	10.0.2.33

## Utiliser les dictionnaires

Il y a plusieurs façons d'utiliser les dictionnaires dans PowerShell. La première est de passer par l'opérateur de déréférencement (. ) :

```
PS> $Hash.DC2012SRV  
10.0.2.48  
PS> $Hash.DC2008SRV  
10.0.2.33
```

Cette notation peut sembler très familière. En effet, un dictionnaire étant un objet comme un autre, il n'échappe pas à la règle de base de manipulation des objets dans PowerShell. Une autre manière de manipuler les dictionnaires est d'utiliser l'opérateur d'index :

```
PS> $Hash["DC2012SRV"]
10.0.2.48
PS> $Hash["DC2008SRV"]
10.0.2.33
PS> $Hash["DC2012SRV", "DC2008SRV"]
10.0.2.48
10.0.2.33
```

Cette notation est de loin la meilleure car elle offre une plus grande souplesse d'utilisation. Observons à présent les membres d'un dictionnaire :

```
PS> $Hash | Get-Member
```

TypeName : System.Collections.Hashtable

Name	MemberType	Definition
Add	Method	void Add(System.Object)
Clear	Method	void Clear(), void IDi
Clone	Method	System.Object Clone()
Contains	Method	bool Contains(System.O
ContainsKey	Method	bool ContainsKey(Syst
ContainsValue	Method	bool ContainsValue(Sys
CopyTo	Method	void CopyTo(array arra
Equals	Method	bool Equals(System.Obj
GetEnumerator	Method	System.Collections.IDi
GetHashCode	Method	int GetHashCode()
GetObjectData	Method	void GetObjectData(Sys
GetType	Method	type GetType()
OnDeserialization	Method	void OnDeserialization
Remove	Method	void Remove(System.Obj
ToString	Method	string ToString()
Item	ParameterizedProperty	System.Object Item(Sys
Count	Property	int Count {get;}
IsFixedSize	Property	bool IsFixedSize {get;
IsReadOnly	Property	bool IsReadOnly {get;}
IsSynchronized	Property	bool IsSynchronized {g
<b>Keys</b>	Property	System.Collections.ICollection
SyncRoot	Property	System.Object SyncRoot
<b>Values</b>	Property	System.Collections.ICollection

On peut déjà remarquer deux propriétés intéressantes à première vue. La première est la propriété **Keys** qui fournit la liste des propriétés d'un dictionnaire :

```
PS> $Hash.Keys
```

```
DC2012SRV
DC2008SRV
```

La seconde est la propriété `Values` listant quant à elle l'ensemble des valeurs correspondantes :

```
PS> $Hash.Values  
10.0.2.48  
10.0.2.33
```

Depuis l'arrivée de PowerShell version 3, il est possible de créer des dictionnaires de manière ordonnée.

### Syntaxe d'un dictionnaire ordonné

```
❶ [ordered]@{ <clé> = <valeur> ; [ <clé> = <valeur> ]... }
```

Le mot-clé `[ordered]` ❶ est un attribut signifiant que le dictionnaire doit être ordonné. L'exemple suivant illustre la création d'un dictionnaire à quatre clés :

```
PS> $Hash = @{ One = 1 ; Two = 2 ; Three = 3 ; Four = 4}
```

L'affichage de ce dictionnaire montre bien que les valeurs sont désordonnées :

```
PS> $Hash  


| Name  | Value |
|-------|-------|
| Four  | 4     |
| One   | 1     |
| Three | 3     |
| Two   | 2     |


```

Pour maintenir l'ordre de création du dictionnaire, il faut placer l'attribut `[ordered]` avant le symbole `@` :

```
PS> $Hash = [ordered]@{ One = 1 ; Two = 2 ; Three = 3 ; Four=4}  
PS> $Hash
```

```


| Name  | Value |
|-------|-------|
| One   | 1     |
| Two   | 2     |
| Three | 3     |
| Four  | 4     |


```

Le résultat nous montre bien que l'ordre de création a été respecté, ce qui peut être un élément essentiel dans le cadre de l'écriture de scripts.

Maîtriser l'utilisation des dictionnaires est donc un atout indispensable pour écrire du code plus complexe, mais aussi plus cohérent.

# 10

## Les boucles

---

*Il est parfois nécessaire de réaliser une opération un certain nombre de fois, même si les modalités d'action sont souvent différentes selon les objectifs recherchés. La meilleure alternative, dans ce cas précis, est d'opter pour les boucles. PowerShell offre la possibilité d'effectuer de plusieurs façons ce genre d'actions répétées.*

*Dans ce chapitre, nous étudierons toutes les formes de boucles que l'on peut trouver dans PowerShell, mais qui existent dans d'autres langages : les boucles `for`, `foreach`, `do..while`, `do..until` et `while`.*

### SOMMAIRE

- ▶ `for`
- ▶ `foreach`
- ▶ `do..while`
- ▶ `do..until`
- ▶ `while`

## La boucle for

La forme de boucle la plus classique est l'instruction `for`. Elle exécute une ou plusieurs instruction(s) tant qu'une condition particulière prend la valeur `true`, c'est-à-dire tant qu'elle est vraie. Dans l'immense majorité des cas, la boucle `for` sera utilisée pour parcourir un ensemble de valeurs, comme un tableau, dans l'objectif de traiter tout ou partie de ces valeurs.

### Syntaxe de la boucle for

```
for (① <init> ; ② <condition> ; ③ <r p tition>)
{
    <liste d'instructions> ④
}
```

Tout d'abord, on initialise une variable ① via une valeur de départ, qui servira de valeur de référence dans l'évaluation de la condition ②, qui se résout en valeur booléenne `true` ou `false`. Tant que la condition est vraie, le bloc représentant la liste d'instructions ④ s'ex cutera de nouveau. Pour d finir une limitation dans le processus d'ex cution, il est possible de modifier la valeur de r f rence   chaque r p tition ③ par un syst me d'incr mentation.

Voici des exemples simples qui aideront   mieux comprendre comment utiliser la boucle `for`.

### Exemple de boucles for utilisant des compteurs

```
for ($c=0; $c -lt 5; $c++){
    Write-Host $c
}
```

```
0
1
2
3
4
```

```
$y = 5

for (; $y -lt 10; $y++) {
    Write-Host $y
}

5
6
```

```
7  
8  
9
```

Notez que dans le second exemple, la variable `$y` a été initialisée à l'extérieur de la boucle `for`. L'exemple suivant est un peu plus complexe que les précédents.

#### Boucle for utilisant l'opérateur de sous-expression \$()

```
for ($c=0; $($x=$c*2; $c -le 10); $c++) {  
    Write-Host $x  
}  
  
0  
2  
4  
6  
8  
10  
12  
14  
16  
18  
20
```

Il met en perspective l'initialisation de deux variables ainsi que l'utilisation de l'opérateur de sous-expression pour définir l'espace réservé aux conditions. La flexibilité de cette construction est très grande comme on peut le constater à travers cet exemple.

## La boucle foreach

Un autre type de boucle très utilisé est la boucle `foreach`. En effet, elle est le type d'instruction idéal lorsqu'il s'agit d'itérer à travers une série de valeurs au sein d'une collection comme un tableau.

#### Syntaxe de la boucle foreach

```
foreach ① (<élément de la collection> ② in $<collection> ③)  
{  
    <liste d'instructions> ④  
}
```

Le fonctionnement de l'instruction `foreach` ① est finalement très linéaire : une collection ③ de valeurs est représentée par la variable `$<collection>`. Lorsque la

boucle `foreach` s'exécute, elle effectue un passage pour chaque élément de la collection, représenté par la variable `$<élément de la collection>` ②. Enfin, un bloc d'instructions ④ est exécuté sur chaque élément de la collection.

L'exemple suivant illustre la création d'un tableau et l'utilisation de la boucle `foreach` pour en afficher chaque élément.

#### Boucle foreach parcourant un tableau d'entiers

```
$TabNums = 1,2,3,4,5

foreach ($Element in $TabNums) {
    Write-Host $Element
}

1
2
3
4
5
```

La collection d'éléments peut aussi être retournée par une cmdlet.

#### Boucle foreach parcourant un tableau de processus

```
foreach ($ProcessName in Get-Process) {
    Write-Host $ProcessName.Name
}

AcroRd32
AcroRd32
armsvc
aticlxx
atiesrxx
```

Ici, la cmdlet `Get-Process` fournit une collection de valeurs qui est parcourue par l'instruction `foreach`, laquelle crée automatiquement une variable nommée `$ProcessName`. Cette variable automatique est passée à la cmdlet `Write-Host` qui en extrait uniquement la propriété `Name`. Le résultat affiché est donc une liste de noms de processus.

**NE PAS CONFONDRE `foreach` et `Foreach-Object`**

Un point important à souligner est qu'il ne faut pas confondre l'instruction `foreach` avec la cmdlet `Foreach-Object` qui dispose d'un alias nommé `foreach`. La différence essentielle est que la boucle consomme plus d'espace mémoire, mais bénéficie de certaines optimisations qui augmentent sa rapidité d'exécution. Les différences qui les distinguent sont à considérer dans le cadre de gestion de masses de données importantes où les critères de performances sont essentiels. Mis à part ce cas précis, utiliser l'une ou l'autre construction est un choix qui ne remettra pas en cause la qualité des commandes ou scripts.

## La boucle `do..while`

Lorsqu'une boucle est utilisée, il est parfois utile ou nécessaire d'exécuter un bloc d'instructions *avant* qu'une condition soit évaluée. C'est la finalité de l'instruction `do..while`. En outre, tant que la condition reste vraie, le bloc de script est répété.

### Syntaxe de la boucle `do..while`

```
do ① {  
    <liste d'instructions>  
} while ② (<condition>)
```

Le mot-clé `do` ① sert à exécuter la liste d'instructions au moins une fois et, tant que la condition ② garde la valeur `true`, la liste d'instructions s'exécute de nouveau. L'exemple suivant montre une utilisation simple de l'instruction `do..while` : nous initialisons une variable et affichons sa valeur.

### Boucle `do..while` affichant une valeur tant que celle-ci est inférieure à 5

```
do {  
    $c++;Write-Host $c  
} while($c -lt 5)  
  
1  
2  
3  
4  
5
```

On peut remarquer que la condition a la valeur `true` tant que la variable `$c` est strictement inférieure à 5. Cependant, la sortie nous montre aussi la valeur 5. En effet, PowerShell exécute le bloc d'instructions *avant* l'évaluation de la condition, et ce,

pour chaque tour effectué. Donc, la variable \$c a été tout simplement incrémentée et affichée avec un temps d'avance sur le processus d'évaluation de la condition. L'exemple suivant illustre l'utilisation d'une boucle `do..while` parcourant les éléments d'un tableau jusqu'à la valeur 5 :

```
PS> $Tab = 1,2,3,4,5
PS> do {$c++;$z++;} while($Tab[$z] -ne 5)
PS> $c
4
```

L'instruction `do..while` est idéale lorsqu'il s'agit de s'assurer qu'au sein d'une boucle, une liste d'instructions soit exécutée au moins une fois, et ce, quelle que soit la valeur résultante de l'évaluation de la condition.

## La boucle `do..until`

La boucle `do..until` est très similaire à la boucle `do..while` en ce sens que la liste d'instructions est exécutée au moins une fois, et ce, quelle que soit l'évaluation de la condition correspondante. La différence entre `do..while` et `do..until` est que cette dernière s'exécute uniquement tant que la condition correspondante a comme valeur `false`.

### Syntaxe de la boucle `do..until`

```
do ❶ {
    <liste d'instructions>
} until ❷ (<condition>)
```

Ici aussi, le mot-clé `do` ❶ sert à exécuter la liste d'instructions au moins une fois. Cependant, cette dernière s'exécutera de nouveau tant que la condition ❷ gardera la valeur `false`.

### Exemple de l'utilisation d'une boucle `do..until` avec la cmdlet `Read-Host`

```
do {
    $o = Read-Host "L'opération est risquée. Voulez-vous continuer? [O/N]"
} until($o -eq 'N')
```

```
L'opération est risquée. Voulez-vous continuer? [O/N]: o
L'opération est risquée. Voulez-vous continuer? [O/N]: o
L'opération est risquée. Voulez-vous continuer? [O/N]: n
```

Cet exemple illustre l'utilisation d'une boucle `do..until` et de la cmdlet `Read-Host`. PowerShell invite l'utilisateur à entrer une valeur. Si cette valeur est `o`, alors la boucle s'exécutera de nouveau. Si la valeur est `n` et si la condition a donc comme valeur `true`, alors PowerShell cesse de répéter l'exécution de la boucle. La boucle suivante est un cas d'utilisation plus classique avec l'initialisation d'un compteur.

#### Création d'un compteur comme base de référence à l'intérieur d'une boucle do..until

```
do {  
    $c++;write-host $c  
} until($c -eq 10)  
  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Le compteur `$c` est initialisé ; il ne cessera pas d'être incrémenté jusqu'à ce qu'il atteigne la valeur `10`. Pendant ce temps-là, la valeur du compteur `$c` est affichée à l'écran via la cmdlet `Write-Host`.

L'utilisation de la boucle `do..until` est à la lumière des exemples que nous venons d'aborder, presque la même que celle de `do..while`. Utiliser l'une ou l'autre de ces constructions est à considérer selon les objectifs à atteindre.

## La boucle while

La boucle `while` est la dernière que nous allons étudier dans cette section. Elle repose évidemment sur le même principe que les autres boucles que nous avons explorées, mais elle est plus souple dans sa construction. Par exemple, l'instruction `while` propose un type de construction plus simple que la boucle `for`.

### Syntaxe de la boucle while

```
while ① (<condition>)
{
    <liste d'instructions> ②
}
```

Le fonctionnement de la boucle `while` se présente comme suit : PowerShell évalue la condition ① associée à la boucle `while` et détermine le résultat en fonction de cette évaluation. Si le résultat est `true`, alors PowerShell exécute la liste d'instructions correspondante ②, et l'exécution du bloc de commandes se répétera tant que ladite condition restera vraie. Par exemple, la boucle `while` suivante affiche la valeur d'une variable tant que celle-ci est inférieure ou égale à 10.

#### Boucle while incrémentant un compteur jusqu'à ce que celui-ci soit inférieur ou égal à 10

```
$c = 0

while ($c -le 10) {
    Write-Host "Tour numéro -> $c";$c++
}

Tour numéro -> 0
Tour numéro -> 1
Tour numéro -> 2
Tour numéro -> 3
Tour numéro -> 4
Tour numéro -> 5
Tour numéro -> 6
Tour numéro -> 7
Tour numéro -> 8
Tour numéro -> 9
Tour numéro -> 10
```

Dans cet exemple, la condition `($c -le 10)` est vraie tant que la variable `$c` est inférieure ou égale à 10. En parallèle, PowerShell incrémente `$c` d'une unité à chaque répétition. Lorsque `$c` a comme valeur 10, le cycle de répétition de la boucle `while` est arrêté.

La boucle `while` est un excellent moyen d'exécuter du code de manière répétée, et ce, en fonction du caractère vrai d'une condition.

# 11

## Les instructions conditionnelles

---

*Les instructions conditionnelles servent à exécuter une ou plusieurs instruction(s) uniquement si une condition particulière est remplie.*

*Nous commencerons par étudier l'instruction `if..else`, qui n'exécute pas le même bloc d'instructions selon que la condition est remplie ou non. Ensuite, nous explorerons l'instruction `if..elseif..else` qui a un fonctionnement similaire, mais propose plusieurs conditions. Enfin, nous nous intéresserons à l'instruction `switch`, plus adaptée que la précédente lorsque les conditions sont nombreuses.*

### SOMMAIRE

- ▶ `if..else`
- ▶ `if..elseif..else`
- ▶ `switch`

## L'instruction conditionnelle if..else

L'instruction `if..else` est utilisée dans des logiques d'orientation du code vers des chemins différents. Le principe est le suivant : si une condition particulière ① est évaluée à la valeur `true`, alors une liste d'instructions ② sera exécutée et PowerShell quittera l'instruction. Dans le cas contraire ③, c'est un autre bloc d'instructions ④ qui sera déroulé. La condition peut être n'importe quel type d'expression renvoyant comme résultat une valeur booléenne (`true` ou `false`).

### Syntaxe de l'instruction if..else

```
if ① (<expression conditionnelle>
{
    <liste d'instructions> ②
}
[else ③
{
    <liste d'instructions> ④
}]
```

#### NOTE Sur le mot-clé else

Le mot-clé `else` n'est pas obligatoire. Dans ce cas, il n'y a pas de comportement « par défaut », mais seulement des instructions à exécuter lorsque la condition est vraie.

Voici un exemple simple de l'utilisation de l'instruction `if..else` :

### Comparaison de deux entiers

```
$x = 4
$y = 2

if ($x -gt $y)
{
    Write-Host "La variable x est plus grande que la variable y."
}
else
{
    Write-Host "La variable y est plus grande que la variable x."
}
```

Ici, deux variables sont créées : `$x` et `$y`, initialisées respectivement avec les valeurs 4 et 2. Puis une construction `if..else` est utilisée afin de tester si `$x` est supérieure à `$y`. Le résultat du test nous donne ceci :

La variable x est plus grande que la variable y.

Prenons un autre exemple s'appliquant sur des chaînes. Il s'agira de vérifier si elles sont égales.

### Comparaison de deux chaînes

```
$string_a = "Blue"  
$string_b = "Blue"  
  
if ($string_a -eq $string_b)  
{  
    Write-Host "Les valeurs sont identiques."  
}  
else  
{  
    Write-Host "Les valeurs ne sont pas identiques."  
}
```

Et voici ce que nous renvoie PowerShell :

Les valeurs sont identiques.

## L'instruction conditionnelle if..elseif..else

La construction `if..elseif..else` est une extension de la précédente qui donne la possibilité d'évaluer plusieurs conditions pour orienter son code.

### Syntaxe de l'instruction if..elseif..else

```
if ① (<expression conditionnelle>)  
{  
    <liste d'instructions_1> ②  
}  
[elseif ③ (<expression conditionnelle>)  
{  
    <liste d'instructions_2> ④  
}]  
[else ⑤  
{  
    <liste d'instructions_3> ⑥  
}]
```

PowerShell évalue tout d'abord une expression conditionnelle ①. Si elle a la valeur `true`, alors il exécute le bloc `liste d'instructions_1` ② et quitte l'instruction. Si l'expression conditionnelle a la valeur `false`, alors PowerShell évalue la condition spécifiée à l'aide du mot-clé `elseif` ③. Si sa valeur est `true`, le bloc `liste d'instructions_2` ④ est exécuté et PowerShell quitte l'instruction. Enfin, si toutes les conditions ont comme valeur `false` ⑤, alors le bloc `liste d'instructions_3` ⑥ est exécuté et PowerShell quitte l'instruction.

#### NOTE Sur le mot-clé `elseif`

Le mot-clé `elseif` peut être utilisé plusieurs fois pour produire une série de tests conditionnels.

Pour traduire ce que nous venons d'évoquer en un exemple pratique, reprenons un des exemples précédents en le complétant :

#### Évaluation de plusieurs expressions conditionnelles avec `if..elseif..else`

```
$x = 1
$y = 2

if ($x -gt $y)
{
    Write-Host "La variable x est plus grande que la variable y."
}
elseif ($x -eq $y)
{
    Write-Host "La variable x est égale à la variable y."
}
else
{
    Write-Host "La variable x n'est ni supérieure, ni égale à la variable y."
}
```

L'instruction `if..elseif..else` est une bonne façon de complexifier l'orientation du code, par exemple dans un script, et son utilisation est très récurrente en raison de la flexibilité qu'elle apporte.

## L'instruction `switch`

L'instruction `if..elseif..else` est très utile lorsqu'il s'agit d'orienter son code en fonction de plusieurs conditions. Cependant, lorsque le nombre de conditions à gérer est conséquent, le code peut devenir illisible et donc difficile à maintenir. À l'instar

d'autres langages, PowerShell dispose de l'instruction `switch` qui, dans sa structure, est une sorte de collection d'instructions `if`.

### Syntaxe de l'instruction `switch`

```
switch [-regex|-wildcard|-exact] [-casesensitive] (valeur) ①
ou
switch [-regex|-wildcard|-exact] [-casesensitive] -file nom_fichier ②

{
    chaîne|nombre|variable|{expression} {liste d'instructions}
    default ③ {liste d'instructions}
}
```

En observant cette syntaxe, on peut constater que l'instruction `switch` est complexe dans sa structure. Ici, deux possibilités de constructions sont possibles :

- effectuer des actions en fonction d'une valeur particulière (qui peut être de n'importe quel type) ① ;
- effectuer des actions basées sur le contenu d'un fichier spécifié ②.

L'instruction `switch` prend en charge un certain nombre de paramètres :

- **-Regex**

La clause de correspondance est traitée comme une chaîne `regex`. Si la valeur en question n'est pas une chaîne, PowerShell ignore ce paramètre. Ce paramètre désactive les paramètres `-Wildcard` et `-Exact`.

- **-Wildcard**

La clause de correspondance est traitée comme une chaîne contenant un caractère générique. Si la valeur en question n'est pas une chaîne, PowerShell ignore ce paramètre. Ce paramètre désactive les paramètres `-Regex` et `-Exact`.

- **-Exact**

La clause de correspondance doit correspondre exactement. Si la valeur en question n'est pas une chaîne, PowerShell ignore ce paramètre. Ce paramètre désactive les paramètres `-Regex` et `-Wildcard`.

- **-CaseSensitive**

Ce paramètre force PowerShell à respecter la casse.

- **-File**

Ce paramètre passe en paramètre un fichier plutôt qu'une valeur calculée à partir d'une instruction.

Dans le bloc de code de l'instruction `switch`, au moins une condition doit exister. De plus, une clause par défaut ③ peut être présente, mais pas plusieurs.

L'exemple suivant illustre l'utilisation de l'instruction `switch` de manière basique en créant une variable qu'on compare à une série de valeurs.

#### Évaluation de nombreuses conditions avec l'instruction `switch`

```
$c = 7

switch ($c) {
    1 {"Il s'agit de 1."}
    2 {"Il s'agit de 2."}
    3 {"Il s'agit de 3."}
    4 {"Il s'agit de 4."}
    5 {"Il s'agit de 5."}
    6 {"Il s'agit de 6."}
    7 {"Il s'agit de 7."}
}
```

Il s'agit de 7.

Orientons-nous à présent vers un exemple un peu plus complexe s'inspirant du précédent.

#### Instruction `switch` comportant le mot-clé `break`

```
$c = 7

switch ($c) {
    1 {"Il s'agit de 1."; break}
    2 {"Il s'agit de 2."; break}
    3 {"Il s'agit de 3."; break}
    4 {"Il s'agit de 4."; break}
    5 {"Il s'agit de 5."; break}
    6 {"Il s'agit de 6."; break}
    7 {"Il s'agit de 7."; break}
    8 {"Il s'agit de 8."; break}
    9 {"Il s'agit de 9."; break}
    10 {"Il s'agit de 10."; break}
}
```

Il s'agit de 7.

L'instruction `switch` comporte ici plus de conditions, mais le mot-clé `break` est associé à chacune d'elles. Dans ce contexte, ce mot-clé indique à PowerShell d'arrêter le traitement et de sortir de l'instruction `switch`. Par conséquent, au lieu d'avoir en sortie deux lignes avec la valeur `Il s'agit de 7.`, l'affichage n'en montre qu'une. Sans ce mot-clé, PowerShell continue d'évaluer toutes les conditions.

### Instruction switch sans le mot-clé break

```
$c = 7

switch ($c) {
1 {"Il s'agit de 1."}
2 {"Il s'agit de 2."}
3 {"Il s'agit de 3."}
4 {"Il s'agit de 4."}
5 {"Il s'agit de 5."}
6 {"Il s'agit de 6."}
7 {"Il s'agit de 7."}
7 {"Il s'agit de 7."}
8 {"Il s'agit de 8."}
9 {"Il s'agit de 9."}
10 {"Il s'agit de 10."}
}
```

```
Il s'agit de 7.
Il s'agit de 7.
```

Il est possible d'utiliser les expressions régulières (voir chapitre sur les expressions régulières) avec l'instruction `switch`, à l'aide du paramètre `-Regex`.

### Instruction switch avec des expressions régulières

```
switch -regex ("xyz")
{
^x {"La valeur commence par la lettre x."}
x.z {"Il y a une lettre entre les lettres x et z."}
Default {"Pas de correspondance trouvée."}
}
```

```
La valeur commence par la lettre x.
Il y a une lettre entre les lettres x et z.
```

```
switch -regex ("7Px$")
{
'^\d' {"La valeur commence par un nombre."}
'\w$' {"La valeur se termine par une lettre."}
Default {"Pas de correspondance trouvée."}
}
```

```
La valeur commence par un nombre.
La valeur se termine par une lettre.
```

L'instruction `switch` est une structure riche, mais nous n'aborderons pas tous les cas d'utilisation car cela déborde du cadre de cet ouvrage. Néanmoins, à partir de ce que nous avons étudié dans cette section, nous pouvons constater la puissance de cette structure, ainsi que la très grande flexibilité qu'elle peut apporter dans un script.

# 12

## Les fonctions

---

*Dans ce chapitre, nous allons étudier un élément fondamental de PowerShell en tant que shell, mais aussi en tant que langage : les fonctions. En effet, dans le monde de PowerShell, elles sont considérées comme des commandes à part entière, au même titre que les scripts, cmdlets et même les commandes natives Windows. De plus, les fonctions sont aussi des passerelles vers la création de modules.*

*Nous commencerons donc par définir les fonctions et leur véritable valeur ajoutée en termes de programmation, pour ensuite étudier la syntaxe et comment les documenter. Enfin, nous aborderons les fonctions avancées dont la connaissance est fondamentale, car elles permettent d'écrire des cmdlets sans avoir besoin d'un langage .NET, comme C# ou Visual Basic .NET.*

### SOMMAIRE

- ▶ Comprendre les fonctions
- ▶ La syntaxe d'une fonction dans le détail
- ▶ Documenter une fonction
- ▶ Les fonctions avancées

## Comprendre les fonctions

Une fonction est une série d'instructions identifiée par un nom, qui pourra être appelée à plusieurs reprises au cours de l'exécution d'un programme en mémoire.

En général, lorsqu'une fonction est définie, elle renvoie une valeur qui peut être affichée directement en sortie d'écran, affectée à une variable, mais aussi utilisée au sein d'autres fonctions. Avec PowerShell, les fonctions sont riches et extrêmement souples, et elles sont aussi bien écrites de manière simple avec une syntaxe aisément compréhensible, que de manière beaucoup plus complexe.

### Déclarer une fonction

Voici un exemple d'une fonction très sommaire :

#### Déclaration d'une simple fonction

```
function ① Simple-fonction ② { ③
    Write-Host "Ceci est une simple fonction."
} ③
```

Pour la déclaration, il suffit d'utiliser le mot-clé `function` ①, suivi du nom à donner à notre nouvelle fonction, `Simple-fonction` ②. Les accolades `{}` ③ servent à circonscrire le bloc de code à exécuter. Les fonctions sont construites comme les blocs de script, à la différence qu'elles ont un nom ; en réalité, ces derniers ne sont autres que des fonctions anonymes ou lambda.

Si nous exécutons cette fonction dans un shell, voici ce que nous obtenons :

```
PS> Simple-fonction
Ceci est une simple fonction.
```

Donc, en PowerShell, une fonction la plus simple possible a le format suivant.

#### Structure fondamentale d'une fonction

```
function <nom de la fonction> {instructions}
```

Il est fortement recommandé de suivre les règles de nommage qui touchent toutes les commandes PowerShell : une paire verbe-nom où :

- le verbe représente une action à mener par la fonction ;
- le nom représente l'élément visé par l'action.

Bien construit, le nom d'une fonction a la même structure que celui d'une cmdlet. L'utilisateur ne fait pas la différence entre les deux types de commandes, contrairement à PowerShell. Toutefois, nous pouvons utiliser la ligne de commande suivante pour savoir à quel type on a affaire :

```
PS> get-command Simple-fonction | Select-Object -Property CommandType, Name  
 CommandType      Name  
 -----  
 Function          Simple-fonction
```

On voit que la propriété  `CommandType` renvoie bien la valeur `Function`, ce qui indique qu'il s'agit d'une fonction et non d'une cmdlet ou même d'un alias.

#### RÉFÉRENCE Liste de verbes et de noms standardisés

► <http://go.microsoft.com/fwlink/?LinkId=160773>

Cette page fournit un certain nombre d'informations et une liste exhaustive des verbes et noms utilisés dans PowerShell.

## Les paramètres

Les fonctions acceptent des arguments de plusieurs façons. Dans un premier temps, nous aborderons uniquement la variable `$args` via laquelle PowerShell fournit les arguments à une fonction. Une autre section sera consacrée à la définition de paramètres liés aux fonctions.

### La variable `$args`

La variable `$args` propose un mécanisme automatique pour fournir des arguments aux fonctions. C'est un tableau de paramètres positionnels (voir plus loin). Prenons un exemple :

#### Fonction utilisant la variable `$args`

```
function Math-Add {  
    [int]$result = [int]$args[0] ① + [int]$args[1] ②  
    Write-Output "Le résultat de l'opération est: $result"  
}
```

Ici, la variable `$args` est invoquée deux fois : la fonction va recevoir comme premier paramètre ① le premier élément du tableau `$args` et comme second paramètre ② le second élément du tableau.

Maintenant, appelons la fonction `Math-Add` interactivement :

```
PS> Math-Add 13 22  
35
```

Les deux paramètres (13 et 22) sont précisés dans la commande et implicitement transmis par le runtime à la variable `$args`.

- `$args[0]` contient la valeur 13.
- `$args[1]` contient la valeur 22.

Il faut noter que ces deux éléments constituent des variables à part entière, même si les valeurs correspondantes font partie de la variable `$args`.

## Les paramètres nommés

La variable `$args` est limitée en termes de marge de manœuvre pour le programmeur. PowerShell propose donc d'autres façons de définir les paramètres. L'une d'elles est d'utiliser les paramètres nommés.

### Syntaxe d'une fonction avec paramètres nommés

```
function <nomp> {  
    param([type]$paramètre1,[type]$paramètre2)  
    <liste d'instructions>  
}
```

Prenons un exemple de ce type de fonction.

### Fonction utilisant les paramètres nommés

```
function Get-Identity {  
    param([String]$Nom,[String]$Prénom)  
    Write-Host "Votre nom est $Nom et votre prénom est $Prénom."  
}
```

Exécutons cette fonction en mode shell :

```
PS> Get-Identity -Nom 'Ayari' -Prénom 'Kais'  
Votre nom est Ayari et votre prénom est Kais.
```

Les paramètres nommés sont à l'évidence un moyen de mieux maîtriser l'exécution d'une commande, qu'elle soit de type fonction ou cmdlet. Pour les administrateurs ne maîtrisant pas l'utilisation du shell, et notamment les commandes, il est préférable de recourir aux paramètres nommés plutôt qu'aux paramètres positionnels que nous allons découvrir dans la prochaine section.

## Les paramètres positionnels

Contrairement aux paramètres nommés, les paramètres positionnels n'ont pas de nom et ils sont directement liés à la variable `$args`. En effet, lorsqu'une fonction est appelée avec des paramètres positionnels, ces derniers sont traités dans l'ordre et sont affectés au tableau `$args`. Prenons un exemple :

### Fonction utilisant les paramètres positionnels

```
function Add-Env {
    $Environnement = $args[0]
    Write-Host "Votre environnement est Windows $Environnement."
}
```

Exécutons notre fonction :

```
PS> Add-Env 'PowerShell'
Votre environnement est Windows PowerShell.
```

Les paramètres positionnels sont aussi utilisés dans d'autres contextes qu'avec la variable `$args`. Mais nous aborderons ce point dans la section réservée aux fonctions avancées.

## Explorer un peu plus les fonctions

Dans cette section, nous allons étudier un peu plus la structure et la syntaxe d'une fonction. Ce faisant, nous constaterons la réelle valeur ajoutée qu'apportent les fonctions dans l'écosystème PowerShell.

### La syntaxe dans le détail

La syntaxe que nous avons vue précédemment est simple, mais rudimentaire. En voici une un peu plus élaborée.

#### Fonction utilisant des méthodes avancées

```
function [<portée:①>]<nom> [([type ②]$param1,[type]$param2))]
{
    param ③ ([type]$param1, [type]$param2)

    begin ④ {
        <liste d'instructions>
    }
}
```

```
process ⑤ {  
    <liste d'instructions>  
}  
  
end ⑥ {  
    <liste d'instructions>  
}
```

On observe ici l'apparition de nouveaux mots-clés qui sont importants pour profiter au maximum des fonctions. Tout d'abord, comme dans d'autres langages, les fonctions ont une portée ①. Cela signifie que PowerShell protège l'accès aux fonctions (mais aussi aux variables, alias et PSDrives), pour éviter de modifier du code par inadvertance. Ces portées sont au nombre de quatre :

- **Global:**

C'est la portée qui prend effet lorsqu'une session PowerShell est démarrée. Tous les alias et toutes les fonctions, variables automatiques, variables préférentielles présents au démarrage de la session sont créés dans cette portée globale.

- **Local:**

C'est tout simplement la portée dans laquelle nous nous situons. Cela veut donc dire n'importe quelle portée.

- **Script:**

Cette portée est créée lorsqu'un script est exécuté. Toutes les commandes ou instructions dans le script sont exécutées dans cette portée et non pas dans la portée globale.

- **Private:**

Tous les éléments créés dans cette portée sont dits privés. Ils ne sont pas visibles par les éléments situés dans d'autres portées.

Lorsqu'une variable est définie avec PowerShell, il n'est pas nécessaire d'indiquer son type ②, contrairement à l'usage dans des langages fortement typés comme le C. Une variable peut représenter une chaîne de caractères (**string**), un entier (**integer**) ou encore une valeur booléenne (**bool**). Cependant, il est préférable, lors d'une déclaration de variable, d'indiquer son type pour des raisons de bonnes pratiques.

Le mot-clé **param** ③ est un bloc groupant les paramètres dans le corps de la fonction, même si ces derniers peuvent être définis à l'extérieur du corps de la fonction.

Les fonctions renvoient des valeurs à l'issue de leur exécution, mais ce n'est pas tout. Elles peuvent aussi accepter des données ou objets en provenance du pipeline. De plus, la manière dont ces objets sont gérés peut être différente. Lorsqu'une fonction est déclarée avec le mot-clé **begin** ④, cela signifie que les instructions correspondantes seront exécutées une seule fois à chaque appel de la fonction.

Le bloc `process` ④ est très important parce que le code correspondant s'exécutera pour chaque objet en provenance du pipeline, ou alors une fois s'il n'y a aucune entrée. Son importance est liée à sa capacité à permettre aux fonctions de communiquer d'une certaine façon les unes avec les autres.

Enfin, le bloc `end` ⑤ est le dernier à s'exécuter, une seule fois comme le bloc `begin`.

## Un exemple concret

La fonction suivante illustre l'utilisation de la syntaxe évoquée dans la section précédente.

### Fonction Get-HardDisk

```
function Script:Get-HardDisk
{
    param([int]$Drivetype)

    begin {
        Write-host "Le bloc begin s'exécute en premier.."
    }

    process {
        Get-WmiObject -class Win32_LogicalDisk -Filter "Drivetype='$Drivetype'"
    }

    end {
        Write-host "Le bloc end s'exécute en dernier.."
    }
}
```

Dans cet exemple, nous collectons des informations disque via WMI. Voici le résultat :

```
PS> Get-HardDisk -Drivetype 3

Le bloc begin s'exécute en premier..

DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 930849288192
Size          : 972008976384
VolumeName    : Windows8_OS

Le bloc end s'exécute en dernier..
```

En analysant le résultat, on observe bien que les trois blocs `begin`, `process` et `end` sont exécutés par le runtime. Le fonctionnement de chacun de ces blocs illustre parfaitement leur complémentarité.

## Documenter les fonctions

Pour afficher l'aide des commandes, l'utilisateur dispose d'une cmdlet centrale appelée `Get-Help` :

```
| PS> Get-Help Get-Process
```

La majorité de l'aide intégrée de PowerShell est contenue dans des fichiers XML. Cependant, il est aussi possible de documenter ses propres fonctions sans passer par la création de fichiers XML. Nous aborderons ici l'aide en lien avec les fonctions et non pas celle en lien avec les scripts.

### Syntaxe de l'aide

La documentation des fonctions se fait sous forme de commentaires décrivant des mots-clés.

#### Syntaxe utilisant une série de commentaires

```
| # .< mot-clé > ①  
| # .< contenu de l'aide en lien avec le mot-clé > ②
```

#### Syntaxe utilisant un bloc de commentaires

```
| <#  
| .< mot-clé > ③  
| < contenu de l'aide en lien avec le mot-clé >  
| #>
```

Il est possible de structurer l'aide d'une fonction sous la forme d'une série de commentaires précédés du symbole `#` ou alors de l'intégrer dans un bloc de commentaires délimité par les symboles `<#` et `#>` ③. L'aide basée sur les commentaires est organisée en plusieurs sections. Ces sections sont identifiables par des mots-clés ① qui peuvent, pour certains, apparaître plusieurs fois et que nous évoquerons dans la prochaine section. Le contenu ② est toujours lié à une section et suit immédiatement le mot-clé.

## Description des mots-clés

Il est nécessaire de s'intéresser à la liste de ces mots-clés, ainsi qu'à leur signification. Présentons les plus utilisés, dans l'ordre où ils apparaissent lorsqu'un utilisateur invoque `Get-Help` en mode shell.

- **.SYNOPSIS**

Décrit succinctement la fonction. Ne peut être utilisé qu'une seule fois.

- **.DESCRIPTION**

Donne une description plus détaillée de la fonction. Ne peut être utilisé qu'une seule fois.

- **.PARAMETER**

Décrit un paramètre. Peut être utilisé plusieurs fois. L'idéal est d'avoir autant de mots-clés **.PARAMETER** que de paramètres déclarés afin de documenter au mieux la fonction.

- **.EXAMPLE**

Exemples illustrant l'utilisation de la fonction, ainsi que sa sortie à l'écran. Peut être utilisé plusieurs fois.

- **.INPUTS**

Liste les objets .NET qui peuvent être redirigés en entrée vers la fonction.

- **.OUTPUTS**

Liste les objets .NET retournés par la fonction ou commande.

- **.NOTES**

Donne des informations complémentaires et souvent utiles en lien avec la fonction ou commande.

- **.LINK**

Indique une rubrique connexe, souvent un lien vers une rubrique d'aide accessible en ligne.

- **.EXTERNALHELP**

Indique un chemin vers un fichier d'aide XML.

## Exemple d'une fonction documentée

Nous reprendrons l'exemple précédent, avec cette fois-ci une documentation.

Fonction Get-HardDisk dotée d'une documentation.

```
function Get-HardDisk
{
    <#
        .SYNOPSIS
        Cette fonction permet de collecter des informations disque.

        .DESCRIPTION
        Cette fonction permet de collecter des informations disque. Les
        informations sont obtenues en interrogeant la classe WMI
        Win32_LogicalDisk.

        .PARAMETER DriveType
        Spécifie la valeur numérique correspondant au périphérique de
        stockage.

        .OUTPUTS
        La fonction Get-HardDisk renvoie des objets de type
        Win32_LogicalDisk.

        .EXAMPLE
        Get-HardDisk -Drivetype 3
    #>

    param([int]$Drivetype)

    begin{
        Write-host "Le bloc begin s'exécute en premier.."
    }

    process {
        Get-WmiObject -class Win32_LogicalDisk -Filter "Drivetype='$Drivetype'"
    }

    end {
        Write-host "Le bloc end s'exécute en dernier.."
    }
}
```

Maintenant que nous avons écrit notre documentation, voyons ce que l'on obtient avec [Get-Help](#) :

```
PS> Get-Help Get-HardDisk -full
```

**NOM**

Get-HardDisk

**RÉSUMÉ**

Cette fonction permet de collecter des informations disque.

**SYNTAXE**

```
Get-HardDisk [[-Drivetype] <Int32>] [<CommonParameters>]
```

**DESCRIPTION**

Cette fonction permet de collecter des informations disque. Les informations sont obtenues en interrogeant la classe WMI WIN32\_LogicalDisk.

**PARAMÈTRES**

-Drivetype <Int32>

Spécifie la valeur numérique correspondant au périphérique de stockage.

Obligatoire ? false

Position ? 1

Valeur par défaut

Accepter l'entrée de pipeline ? false

Accepter les caractères génériques ?

<CommonParameters>

Cette applet de commande prend en charge les paramètres courants : Verbose, Debug, ErrorAction, ErrorVariable, WarningAction, WarningVariable, OutBuffer et OutVariable. Pour plus d'informations, tapez « get-help about\_commonparameters ».

**ENTRÉES****SORTIES**

La fonction Get-HardDisk renvoie des objets de type Win32\_LogicalDisk.

----- EXEMPLE 1 -----

```
C:\PS> Get-HardDisk -Drivetype 3
```

La cmdlet `Get-Help` a bien détecté la documentation fournie dans notre fonction.

L'aide basée sur les commentaires est à l'évidence beaucoup plus facile à écrire que celle contenue dans les fichiers XML, car cette dernière est beaucoup plus stricte dans sa syntaxe.

## Les fonctions avancées

PowerShell offre plusieurs cmdlets natives, dont le nombre dépend de la version installée, mais aussi des extensions et autres modules importés. Ces cmdlets sont toutes (versions 1 et 2 de PowerShell) écrites en langage .NET, en particulier C#. Le problème est que cela ne laisse pas de place à la créativité des administrateurs qui, dans l'immense majorité des cas, doivent souvent traiter des urgences et accomplir rapidement un certain nombre de tâches administratives liées au contexte où ils se situent. Pour eux, l'équipe PowerShell a développé depuis la version 2 ce que l'on nomme les fonctions avancées.

### Principe des fonctions avancées

En ce qui concerne leur mécanisme, les fonctions avancées sont similaires aux cmdlets ; la différence est que ces dernières sont développées en langage .NET, tandis que les fonctions avancées sont écrites en langage PowerShell. Cela est très important, car les utilisateurs de PowerShell qui ne maîtrisent pas la programmation (comme les administrateurs) peuvent depuis la version 2 créer ainsi leurs propres commandes.

Il y a cependant une différence entre les fonctions dites classiques et les fonctions avancées. En effet, ces dernières utilisent l'attribut [\[CmdletBinding\]](#), qui a pour but d'identifier la fonction en tant que cmdlet. Prenons pour exemple la fonction précédente que nous transformerons en fonction avancée :

#### Fonction Get-HardDisk utilisant le principe des fonctions avancées

```
function Get-HardDisk
{
    <#
    .SYNOPSIS
    Cette fonction permet de collecter des informations disque.

    .DESCRIPTION
    Cette fonction permet de collecter des informations disque. Les
    informations sont obtenues en interrogeant la classe WMI
    WIN32_LogicalDisk.

    .PARAMETER DriveType
    Spécifie la valeur numérique correspondant au périphérique de
    stockage.

    .OUTPUTS
    La fonction Get-HardDisk renvoie des objets de type
    Win32_LogicalDisk.
```

```
.EXAMPLE
Get-HardDisk -Drivetype 3
#>

[CmdletBinding()] ①

param(
    [Parameter(Mandatory=$True)] ②
    [int]$Drivetype
)

begin{
    Write-host "Le bloc begin s'exécute en premier.."
}

process {
    Get-WmiObject -class Win32_LogicalDisk -Filter "Drivetype='$Drivetype'"
}

end {
    Write-host "Le bloc end s'exécute en dernier.."
}

}
```

Comme évoqué plus haut, l'attribut `[CmdletBinding]` ① déclare une fonction en tant que cmdlet. De cette façon, PowerShell traitera les paramètres de la fonction comme ceux d'une cmdlet. Donc, tout comme la liaison des cmdlets existe nativement, le runtime assurera la liaison des paramètres pour les fonctions avancées. L'attribut `[Parameter]` ② sert à déclarer les paramètres de la fonction.

## Les paramètres avancés

Pour que la fonction avancée ressemble au maximum aux cmdlets, l'attribut `[CmdletBinding]` est requis, mais ne suffit pas. La liaison des paramètres nécessite que ces derniers soient définis de façon plus pointue dans la fonction. Cela implique que la fonction avancée requiert un travail d'architecture ou de définition (qui dépasse le cadre de ce livre) relatif à ses finalités. Donc, à l'attribut `[CmdletBinding]`, il faut en ajouter au moins un, l'attribut `[Parameter]`, qui est obligatoire. Nous commencerons par décrire cet attribut, puis ceux les plus utilisés bien qu'ils ne soient pas obligatoires :

- L'attribut `[Parameter]`

Il est employé pour déclarer un paramètre de la fonction. Il comporte des arguments définissant ses caractéristiques. En voici les principaux :

- `Mandatory` indique si le paramètre est obligatoire. Si cet argument n'est pas spécifié, l'argument est considéré comme optionnel.

- `Position` donne la position du paramètre. Le fait que cet argument existe implique que le paramètre peut être invoqué de manière positionnelle.
- `ParameterSetName` spécifie le jeu de paramètres dans lequel se trouve le paramètre en question. Un jeu de paramètres est un ensemble de paramètres pouvant être invoqués simultanément via une ligne de commande (tous ne sont pas compatibles les uns avec les autres).
- `ValueFromPipeline` indique si le paramètre accepte des objets en provenance du pipeline. Si la fonction avancée doit communiquer avec d'autres commandes au niveau entrant, l'argument `ValueFromPipeline` est souvent nécessaire.
- `ValueFromPipelineByPropertyName` repose sur le même principe que le précédent, à la différence qu'il s'agit ici de propriétés. Par exemple, si le paramètre `name` reçoit un objet, cet objet doit contenir une propriété `name`. Dans le cas contraire, l'objet ne sera pas traité par le paramètre.
- `HelpMessage` spécifie un message décrivant le paramètre.

- L'attribut `[Alias]`

Il donne un nom supplémentaire au paramètre, permettant ainsi à ce dernier d'être appelé via plusieurs noms.

- L'attribut `[AllowNull]`

Il définit à `Null` l'argument du paramètre.

- L'attribut `[AllowEmptyString]`

Il autorise une chaîne de caractères vide en tant qu'argument.

- L'attribut `[ValidateCount]`

Il indique les nombres minimal et maximal d'arguments que le paramètre peut accepter.

- L'attribut `[ValidateLength]`

Il indique les longueurs minimale et maximale de l'argument du paramètre.

Reprendons la fonction `Get-HardDisk` que nous doterons d'attributs améliorant la maîtrise des arguments acceptés.

#### Fonction Get-HardDisk proposant un meilleur contrôle des paramètres

```
function Get-HardDisk
{
    <#
        .SYNOPSIS
        Cette fonction permet de collecter des informations disque.

        .DESCRIPTION
        Cette fonction permet de collecter des informations disque.
```

Les informations sont obtenues en interrogeant la classe WMI Win32\_LogicalDisk.

.PARAMETER DriveType

Spécifie la valeur numérique correspondant au périphérique de stockage.

.OUTPUTS

La fonction Get-HardDisk renvoie des objets de type Win32\_LogicalDisk.

.EXAMPLE

```
Get-HardDisk -Drivetype 3 -Computername 'localhost'  
#>
```

#### [CmdletBinding()]

```
param(  
    [Parameter(Mandatory=$True)]  
    [int]$Drivetype,  
    [Parameter(Mandatory=$True, ValueFromPipeline=$True,  
              ValueFromPipelineByPropertyName=$True)] ①  
    [ValidateLength(1,10)] ②  
    [Alias("cn")] ③  
    [string]$Computername  
)  
  
process {  
    Get-WmiObject -class Win32_LogicalDisk -Filter "Drivetype='$Drivetype'"  
                 -Computername $Computername  
}  
}
```

Notre fonction `Get-HardDisk` accepte des objets de type `string` en provenance du pipeline ①, mais aussi d'autres types d'objets avec une propriété `Computername` ①. De plus, l'argument du paramètre `Computername` doit comprendre entre 1 et 10 caractères ②. Enfin, le même paramètre `Computername` peut être invoqué avec un alias `cn` ③.

Lançons notre fonction, à présent complète, en provoquant volontairement des erreurs. Tout d'abord, fournissons un argument de 12 caractères :

```
PS> Get-HardDisk -Drivetype 3 -Computername 'DCServer2012'
```

```
Get-HardDisk : Cannot validate argument on parameter 'Computername'. The  
argument length of 12 is too long. Shorten the length of the argument to  
less than or equal to "10" and then try the command again.
```

Ici, le shell nous renvoie une erreur, car l'argument est trop long. Les contraintes que nous avons définies ont donc bien été prises en compte par le runtime. Relançons la fonction en sollicitant le pipeline :

```
PS> 'localhost' | Get-HardDisk -Drivetype 3  
DeviceID      : C:  
DriveType     : 3  
ProviderName  :  
FreeSpace      : 930849288192  
Size          : 972008976384  
VolumeName    : Windows8_OS
```

Là aussi, nous avons réussi à envoyer un objet à la fonction `Get-HardDisk`, objet lié au paramètre `Computername`.

Avec tout ce que nous avons étudié au cours de ce chapitre, on s'aperçoit très vite que la connaissance des fonctions est vitale pour qui veut administrer des systèmes Windows, car une fois écrites, les fonctions peuvent être réutilisées de façon illimitée. En outre, lorsqu'elles sont documentées, elles sont plus faciles à maintenir, même par d'autres personnes.

# 13

## Les modules

---

*Une question que nous n'avons pas encore posée est celle de la réutilisation du code. En effet, un code fréquemment utilisé doit être encapsulé dans un package, ou dans un module, pour éviter de le réécrire plusieurs fois. La question de la réutilisation du code pose un certain nombre de problèmes quasi épistémologiques que nous n'aborderons pas dans ce livre. Cependant, réutiliser du code en provenance d'une autre source est un phénomène très répandu, mais malheureusement mal compris.*

*Dans ce chapitre, nous définirons ce qu'est un module PowerShell. Ensuite, nous apprendrons à en construire, opération indispensable pour qui veut créer des extensions. Enfin, nous évoquerons succinctement les PSSnapins, qui représentent en quelque sorte l'ancien modèle en termes d'extensions PowerShell.*

### SOMMAIRE

- ▶ Qu'est-ce qu'un module PowerShell ?
- ▶ Créer un module
- ▶ Un mot sur les PSSnapins

## Qu'est-ce qu'un module PowerShell ?

Un module PowerShell est un package contenant un certain nombre de commandes PowerShell (cmdlets, variables, fonctions, alias ou même fournisseurs). Le but ultime d'un module est de faciliter la réutilisation du code dans des perspectives de partage avec d'autres personnes et de gain de temps non négligeable. Sur la base de cette définition, nous comprenons que, finalement, un module PowerShell n'est rien d'autre qu'un script, à la différence qu'un module porte l'extension `.psm1`, alors qu'un script porte l'extension `.ps1`.

### Se familiariser avec les modules PowerShell

PowerShell dispose d'un certain nombre de modules préchargés lors de son exécution. Ces derniers sont des extensions permettant d'administrer certaines parties du système d'exploitation. En conséquence, un administrateur doit savoir comment interagir avec ces modules afin d'être le plus efficace possible dans le cadre de ses opérations. Voyons quelles sont les cmdlets disponibles pour manipuler les modules :

```
PS> get-command -noun *module*
```

CommandType	Name	ModuleName
Cmdlet	Export-ModuleMember	Microsoft.PowerShell.Core
Cmdlet	Get-Module	Microsoft.PowerShell.Core
Cmdlet	Import-Module	Microsoft.PowerShell.Core
Cmdlet	New-Module	Microsoft.PowerShell.Core
Cmdlet	New-ModuleManifest	Microsoft.PowerShell.Core
Cmdlet	Remove-Module	Microsoft.PowerShell.Core
Cmdlet	Test-ModuleManifest	Microsoft.PowerShell.Core

L'ensemble de ces cmdlets est mis à la disposition des utilisateurs pour se familiariser avec les modules. Le tableau 13-1 décrit leur fonctionnement.

Tableau 13-1 Cmdlets utiles pour manipuler les modules PowerShell

Nom de la cmdlet	Description
Export-ModuleMember	Spécifie les membres ou parties d'un module à exporter à partir d'un fichier <code>.psm1</code> . Ces membres peuvent être des cmdlets, fonctions, variables et alias.
Get-Module	Liste l'ensemble des modules actuellement importés ou même les modules susceptibles d'être importés dans une session PowerShell.
Import-Module	Importe un ou plusieurs module(s) dans une session PowerShell.

Tableau 13-1 Cmdlets utiles pour manipuler les modules PowerShell (suite)

Nom de la cmdlet	Description
New-Module	Crée dynamiquement un module, qui n'existera qu'en mémoire et s'en effacera lorsque la session active sera fermée.
New-ModuleManifest	Crée un manifeste décrivant le contenu et les attributs d'un module.
Remove-Module	Supprime des modules d'une session PowerShell.
Test-ModuleManifest	Vérifie qu'un manifeste de module est parfaitement conforme.

À présent, essayons de lister les modules (sur une plate-forme Windows 8) actuellement chargés en mémoire :

```
PS> Get-Module
```

ModuleType	Name	ExportedCommands
Manifest	Microsoft.PowerShell.Management	{Add-Computer, Add-...
Manifest	Microsoft.PowerShell.Utility	{Add-Member, Add-Type..

La sortie n'affiche que deux modules. Au vu du nombre de cmdlets disponibles dans PowerShell, il y a sûrement d'autres modules disponibles et que nous ne voyons pas. Pour les lister, utilisons le paramètre `-ListAvailable` :

```
PS> Get-Module -ListAvailable
```

Répertoire : C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules

ModuleType	Name	ExportedCommands
Manifest	Appx	{Add-AppxPackage,
Manifest	BitLocker	{Unlock-BitLocker,
Manifest	BitsTransfer	{Add-BitsFile, Com
Manifest	CimCmdlets	{Get-CimAssociated
Manifest	DirectAccessClientComponents	{Disable-DAManualE
Script	Dism	{Add-AppxProvision
Manifest	DnsClient	{Resolve-DnsName,
Manifest	International	{Get-WinDefaultInp
Manifest	iSCSI	{Get-IscsiTargetPo
Script	ISE	{New-IseSnippet, I
Manifest	Kds	{Add-KdsRootKey, G
Manifest	Microsoft.PowerShell.Diagnostics	{Get-WinEvent, Get
Manifest	Microsoft.PowerShell.Host	{Start-Transcript,
Manifest	Microsoft.PowerShell.Management	{Add-Content, Clea
Manifest	Microsoft.PowerShell.Security	{Get-Acl, Set-Acl,
Manifest	Microsoft.PowerShell.Utility	{Format-List, Form
Manifest	Microsoft.WSMAN.Management	{Disable-WSManCred

Manifest	MMAgent	{Disable-MMAgent,
Manifest	MsDtc	{New-DtcDiagnostic
Manifest	NetAdapter	{Disable-NetAdapte
Manifest	NetConnection	{Get-NetConnection
Manifest	NetLbfo	{Add-NetLbfoTeamMe
Manifest	NetQos	{Get-NetQosPolicy,
Manifest	NetSecurity	{Get-DAPolicyChang
Manifest	NetSwitchTeam	{New-NetSwitchTeam
Manifest	NetTCPIP	{Get-NetIPAddress,
Manifest	NetworkConnectivityStatus	{Get-DAConnectionS
Manifest	NetworkTransition	{Add-NetIPHttpsCer
Manifest	PKI	{Add-CertificateEn
Manifest	PrintManagement	{Add-Printer, Add-
Script	PSDiagnostics	{Disable-PSTrace,
Binary	PSScheduledJob	{New-JobTrigger, A
Manifest	PSWorkflow	{New-PSWorkflowExe
Manifest	PSWorkflowUtility	Invoke-AsWorkflow
Manifest	ScheduledTasks	{Get-ScheduledTask
Manifest	SecureBoot	{Confirm-SecureBoo
Manifest	SmbShare	{Get-SmbShare, Rem
Manifest	SmbWitness	{Get-SmbWitnessCli
Manifest	Storage	{Add-InitiatorIdTo
Manifest	TroubleshootingPack	{Get-Troubleshooti
Manifest	TrustedPlatformModule	{Get-Tpm, Initiali
Manifest	VpnClient	{Add-VpnConnection
Manifest	Wdac	{Get-OdbcDriver, S
Manifest	WindowsDeveloperLicense	{Get-WindowsDevelo
Script	WindowsErrorReporting	{Enable-WindowsErr

Ici, la sortie fait état d'un nombre très élevé de modules disponibles dans PowerShell. La colonne [ExportedCommands](#) montre de manière très partielle les cmdlets disponibles au sein de ces modules ; comment en obtenir une liste complète ? Une réponse (mais il y en a d'autres) se trouve du côté de la cmdlet [Get-Command](#) et son paramètre [-Module](#). L'exemple qui suit liste les cmdlets du module [SmbShare](#) :

```
PS> Get-Command -Module SmbShare | Select Name, Module
```

Name	Module
---	-----
Block-SmbShareAccess	SmbShare
Close-SmbOpenFile	SmbShare
Close-SmbSession	SmbShare
Get-SmbClientConfiguration	SmbShare
Get-SmbClientNetworkInterface	SmbShare
Get-SmbConnection	SmbShare
Get-SmbMapping	SmbShare
Get-SmbMultichannelConnection	SmbShare
Get-SmbMultichannelConstraint	SmbShare

Get-SmbOpenFile	SmbShare
Get-SmbServerConfiguration	SmbShare
Get-SmbServerNetworkInterface	SmbShare
Get-SmbSession	SmbShare
Get-SmbShare	SmbShare
Get-SmbShareAccess	SmbShare
Grant-SmbShareAccess	SmbShare
New-SmbMapping	SmbShare
New-SmbMultichannelConstraint	SmbShare
New-SmbShare	SmbShare
Remove-SmbMapping	SmbShare
Remove-SmbMultichannelConstraint	SmbShare
Remove-SmbShare	SmbShare
Revoke-SmbShareAccess	SmbShare
Set-SmbClientConfiguration	SmbShare
Set-SmbServerConfiguration	SmbShare
Set-SmbShare	SmbShare
Unblock-SmbShareAccess	SmbShare
Update-SmbMultichannelConnection	SmbShare

Pour importer le module `SmbShare`, il faut utiliser la cmdlet `Import-Module` :

```
PS> Import-Module SmbShare
```

**NOTE Concernant l'ajout de modules à une session**

Avec l'arrivée de PowerShell version 3, les modules sont importés dynamiquement, c'est-à-dire qu'il n'est plus nécessaire de recourir à la cmdlet `Import-Module`. Il suffit juste de connaître et d'invoquer une cmdlet du module et PowerShell se chargera d'importer ce dernier à la place de l'utilisateur.

## Créer un module PowerShell

L'utilisation des modules livrés avec PowerShell est récurrente dans la perspective d'un administrateur ou d'un ingénieur. Néanmoins, dans certaines circonstances, il est nécessaire de créer ses propres modules pour répondre à des besoins spécifiques.

Il existe deux catégories de modules dans PowerShell :

- les modules binaires écrits en langage .Net comme C# ou VB.NET ;
- les modules écrits en langage de script PowerShell (qui est aussi un langage .NET).

Cette distinction est très importante. L'écriture de modules binaires est souvent le fait de programmeurs désirant écrire des extensions PowerShell. De plus, ces

modules sont dans la majorité des cas écrits en C#. Nous n'étudierons pas comment écrire des modules binaires, car ce livre s'adresse prioritairement aux administrateurs, moins aux programmeurs. Nous nous intéresserons uniquement à la création de modules à partir du matériau PowerShell lui-même.

## Écrire un module en langage PowerShell

Depuis la version 2 de PowerShell, il est possible d'écrire des modules non seulement binaires, mais aussi en PowerShell. Cette possibilité était une véritable révolution à l'époque, parce qu'en matière d'extensions possibles, la seule perspective était d'écrire ce que nous verrons dans la prochaine section et que l'on nomme des PSSnapins ou composants logiciels enfichables qui ne pouvaient que prendre la forme d'une bibliothèque (.dll). Dès lors, écrire ses propres extensions n'était plus réservé à certains programmeurs. La qualité était devenue en quelque sorte universelle et c'est une excellente chose, car on peut juger de la qualité d'un langage à sa capacité à ouvrir des horizons à ceux qui l'utilisent.

Un module écrit en PowerShell n'est rien d'autre qu'un script ; il porte une extension .psm1, mais est utilisé dans des contextes d'exécution tout à fait similaires. Tout ce que nous avons abordé depuis le début du livre va donc nous servir pour la création de modules. D'abord, commençons par écrire le code servant de base à notre module.

### Création d'une fonction avancée qui servira de base à la création de module

```
function Check-Target ① {  
    [CmdletBinding()] ②  
  
    Param(  
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]  
        [System.String[]]$ComputerName ③  
    )  
  
    BEGIN ④ {  
        Write-Host "Début des opérations. Veuillez patienter..." -ForegroundColor Blue  
        -BackgroundColor Yellow  
    }  
  
    PROCESS ⑤ {  
        foreach ($Computer in $ComputerName) {
```

```
$Result ⑥ = Get-WmiObject -Query "SELECT * FROM Win32_PingStatus WHERE Address
= '$Computer'"  
  
$Target ⑦ = New-Object PSCustomObject
$Target | Add-Member -NotePropertyName 'Computer'
           -NotePropertyValue ($Computer)
$Target | Add-Member -NotePropertyName 'IpAddress'
           -NotePropertyValue ($Result.ipv4address)
$Target | Add-Member -NotePropertyName 'ResponseTime'
           -NotePropertyValue ($Result.responsetime)  
  
if ($Result.statuscode -eq 0) {
    $Target | Add-Member -NotePropertyName 'Responding' -NotePropertyValue $True
} else {
    $Target | Add-Member -NotePropertyName 'Responding' -NotePropertyValue
$False
}
Write-Output $Target ⑧  
  
}  
  
}  
  
END {  
  
$Result=$Null
$Target=$Null
Write-Host "`nFin des opérations." -ForegroundColor Blue
           -BackgroundColor Yellow ⑨  
  
}  
}
```

La structure du code consiste en une fonction avancée nommée `Check-Target` ① dont le but est de tester l’accessibilité à des ordinateurs distants et d’afficher les résultats de ces tests à l’écran. La qualité de fonction avancée est donnée par l’attribut `[CmdletBinding]` ② qui confère à `Check-Target` le statut de cmdlet à part entière. Un paramètre obligatoire ③ spécifie une liste d’un ou plusieurs nom(s) d’ordinateur(s). Le bloc `BEGIN` ④ est utilisé ici pour signifier le début des opérations avant le traitement de chacune des valeurs que la commande devra traiter. Puis le bloc `PROCESS` ⑤ traite chacune des cibles et effectue une requête WMI ⑥ afin de rechercher les informations nécessaires à la structuration d’objets ⑦ qui comporteront les éléments d’information affichés en sortie à l’écran ⑧. Enfin, lorsque le processus incluant toutes ces opérations est terminé, l’utilisateur est averti par un message à l’écran ⑨.

Il faut à présent enregistrer le fichier en n’oubliant pas de lui donner l’extension `.psm1`. Ensuite, une bonne pratique consiste à déposer le module dans un dossier spé-

cial. Il n'y a pas d'obligation absolue quant à la localisation du dossier comportant le module, mais des emplacements par défaut existent :

- un emplacement pour le système : `$pshome\Modules` ;
- un autre pour l'utilisateur actif: `$home\Documents\WindowsPowerShell\Modules` ou alors `$home\Mes Documents\WindowsPowerShell\Modules`.

Placer les modules dans ces emplacements est à privilégier, car PowerShell est programmé pour les analyser et les modules y seront donc mieux détectés.

Créons un dossier nommé `CheckTarget` et plaçons-y le module que nous venons de créer. Copions le dossier dans l'emplacement de l'utilisateur actif :

```
PS> Copy-Item -path \Emplacement\du\module -destination  
$home\Documents\WindowsPowerShell\Modules
```

Évidemment, si les dossiers `WindowsPowerShell` et `Modules` n'existent pas, alors il faudra les créer. La dernière étape va consister à importer le module. Nous ajouterons le module explicitement, car la version 3 de PowerShell, qui importe implicitement, est bien moins répandue que la version 2 :

```
PS> Import-Module CheckTarget
```

Il nous reste à tester une commande `Check-Target` :

```
PS> Check-Target -ComputerName 'srv01','srv02','srv03'
```

Début des opérations. Veuillez patienter...

Computer	IpAddress	ResponseTime	Responding
srv01	10.0.2.15	0	True
srv02			False
srv03	10.0.2.17	0	True

Fin des opérations.

Nous pouvons constater que la commande fonctionne bien et produit le résultat escompté. Le module pourra donc être utilisé, voire amélioré par d'autres personnes, en créant par exemple un fichier d'aide expliquant comment utiliser le module de manière adéquate, ou alors en intégrant d'autres fonctions au module dans le but de l'enrichir.

## Un mot sur les PSSnapins

Avant la sortie de la version 2, le seul moyen de créer des extensions PowerShell était d'écrire ce que l'on appelle des PSSnapins. Un PSSnapin est un composant logiciel enfileable structuré en *assembly* .Net qui peut contenir des fournisseurs et des cmdlets. Le processus de création et d'utilisation des PSSnapins était et est toujours contraignant, en ce sens qu'il faut les inscrire dans la base de registres. Cela demandait d'accomplir certaines tâches avant de pouvoir les utiliser et, souvent, les PSSnapins étaient inclus dans des programmes visant à faciliter le processus d'installation et d'utilisation.

L'arrivée de PowerShell version 2 a changé la donne puisque Microsoft a remarquablement changé de paradigme. En effet, le concept de modules a fait son apparition et a permis à l'époque d'étendre les fonctionnalités de PowerShell, soit en utilisant des langages compilés comme C#, soit en utilisant PowerShell. En conséquence, toutes les contraintes inhérentes aux PSSnapins ont été enlevées et les perspectives en la matière considérablement élargies.

À l'heure actuelle, avec la version 3 de PowerShell, il est toujours possible de créer des PSSnapins, mais Microsoft ne le recommande pas et confirme qu'ils vont disparaître petit à petit. Microsoft montre lui-même l'exemple, car les PSSnapins de l'époque sont devenus aujourd'hui des modules, à l'exception de `Microsoft.PowerShell.Core` qui contient des cmdlets intégrées à PowerShell.

L'objectif ici est simple : promouvoir le développement de modules au détriment des PSSnapins, autrement dit flexibilité et souplesse plutôt que lourdeur et contrainte. Le changement de modèle paradigmique se fera lentement et prudemment pour des raisons liées au fonctionnement interne de PowerShell.

Du point de vue des utilisateurs, et en particulier de ceux qui créent des extensions, écrire des PSSnapins n'est plus la voie à choisir parce que des problèmes de compatibilité risquent de survenir avec les versions futures de PowerShell. Plus encore, les PSSnapins existants doivent être convertis en modules dans la perspective d'une meilleure pérennité. Donc, le changement de modèle de programmation concerne aussi les utilisateurs de PowerShell.



# 14

## PowerShell et la gestion d'erreur

---

*Il est très rare d'écrire du code fonctionnant de manière parfaite. Celui-ci réserve souvent des surprises, résultat d'erreurs logiques vis-à-vis desquelles il faut réagir de façon méthodologique et structurée. Cette étape est plus importante que l'écriture du code en tant que telle, car un code mal écrit pourrait avoir des conséquences extrêmement dévastatrices sur l'environnement dans lequel il est exécuté.*

*Le code écrit doit être soumis à des phases de tests évaluant sa qualité, mais aussi une certaine robustesse devenue aujourd'hui plus qu'indispensable. Cependant, parmi les bonnes pratiques, il est fortement recommandé de prévenir ce genre de problèmes, c'est-à-dire de donner les moyens au code écrit de faire face à de telles éventualités.*

*Analyser et identifier les erreurs susceptibles de se produire est donc un phénomène récurrent, voire inévitable. Nous commencerons par distinguer l'erreur de l'exception. En PowerShell, une erreur étant une entité à part entière, son analyse est une condition sine qua non à toute compréhension de ce qu'est une erreur et ce dont elle relève. Nous nous intéresserons aussi au paramètre `-ErrorAction` commun à l'ensemble des cmdlets, pour enfin explorer les instructions `trap` et `try..catch..finally`.*

### SOMMAIRE

- ▶ La différence entre une erreur et une exception
- ▶ Anatomie d'une erreur
- ▶ Le paramètre `-ErrorAction` et la variable `$ErrorActionPreference`
- ▶ `trap`
- ▶ `try..catch..finally`

## Différencier l'erreur de l'exception

PowerShell distingue deux types d'erreurs : celles qui laissent l'exécution se dérouler, appelées **nonterminating errors**, et celles causant l'arrêt de l'exécution du pipeline, appelées quant à elles **terminating errors** ou encore exceptions. La distinction entre ces deux types d'erreurs est importante.

- L'erreur de type **nonterminating** est signalée à l'écran, mais ne provoque pas l'arrêt de l'exécution d'une commande (cmdlet, fonction, script, etc.).
- L'erreur de type **terminating** est aussi signalée et bloque l'exécution. Contre elle, il est possible d'agir de plusieurs façons, que nous étudierons dans les sections suivantes de ce chapitre.

Pour être plus clair, voici à quoi ressemble une erreur de type **terminating** ou exception :

```
PS> 1/0
Tentative de division par zéro.
Au caractère Ligne:1 : 1
+ 1/0
+ ~~~
+ CategoryInfo          : NotSpecified: () [], RuntimeException
+ FullyQualifiedErrorId : RuntimeException
```

En voici une autre, de type **nonterminating** :

```
PS> Get-WmiObject -Class 'Win32_Bios' -ComputerName 'NotExist'
Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT :
0x800706BA)
Au caractère Ligne:1 : 1
+ Get-WmiObject -Class 'Win32_Bios' -ComputerName 'NotExist'
+ ~~~~~
+ CategoryInfo          : InvalidOperationException : () [Get-WmiObject],
COM Exception
+ FullyQualifiedErrorId : GetWMICOM Exception,Microsoft.PowerShell.Commands
.GetWmiObjectCommand
```

En analysant la sortie des deux exemples, on s'aperçoit que le terme **Exception** apparaît dans les deux cas. Alors pourquoi l'erreur provoquée dans le second exemple est-elle considérée comme **nonterminating** alors que le terme **Exception** apparaît en sortie ? La réponse est dans la grille de lecture de PowerShell : il s'agit d'une exception du point de vue du service WMI, mais PowerShell la considère comme une erreur ne nécessitant pas l'arrêt d'une exécution.

Cette dualité très particulière est à situer dans un contexte propre au fonctionnement de PowerShell, et pas dans le contexte d'un autre langage comme C# où les choses

sont différentes. Comprendre ces différences est la base de toutes démarches liées à la gestion d'erreur dans PowerShell.

## Comprendre l'anatomie d'une erreur

Rencontrer des erreurs est un phénomène récurrent, et ce, quel que soit le mode d'utilisation de PowerShell. Savoir les analyser dans leur structure est donc une qualité indispensable, car la capacité de correction n'en sera que plus efficace. En PowerShell, une erreur, quelle que soit sa provenance, est un objet à part entière. Provoquons-en volontairement une :

```
PS> Get-WmiObject -Class Win32_Bios -ComputerName 'NotExist'

Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT :
0x800706BA)
Au caractère Ligne:1 : 1
+ Get-WmiObject -Class Win32_Bios -ComputerName 'NotExist'
+ ~~~~~
+ CategoryInfo          : InvalidOperation : () [Get-WmiObject],
COMException
+ FullyQualifiedErrorId : GetWMICOMException,Microsoft.PowerShell.Commands
.GetWmiObjectCommand
```

Cet exemple illustre une requête WMI via la cmdlet `Get-WmiObject`. Son but est de collecter un certain nombre d'informations concernant le BIOS d'un serveur dont le nom est `NotExist`. En sortie, on s'aperçoit que le serveur n'est pas joignable et qu'une erreur s'affiche donc à l'écran. Les informations disponibles par défaut sont souvent suffisantes pour permettre un débogage, mais il est parfois nécessaire d'analyser davantage les erreurs qui se présentent à nous. Dans ce but, PowerShell conserve l'ensemble des erreurs dans une variable automatique : un tableau nommé `$Error` où les erreurs sont classées dans l'ordre inversement chronologique. L'erreur la plus récente, par exemple, est contenue dans le premier élément :

```
PS> $Error[0]

Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT :
0x800706BA)
Au caractère Ligne:1 : 1
+ Get-WmiObject -Class Win32_Bios -ComputerName 'NotExist'
+ ~~~~~
+ CategoryInfo          : InvalidOperation : () [Get-WmiObject],
COMException
+ FullyQualifiedErrorId : GetWMICOMException,Microsoft.PowerShell.Commands
.GetWmiObjectCommand
```

Voici les différents outils disponibles pour manipuler le tableau \$Error :

PS> get-member -InputObject \$Error		
Name	MemberType	Definition
Add	Method	int Add(System.Object value), void AddRange(System.Collections.ICollection collection)
AddRange	Method	int BinarySearch(int index, int value), void Clear(), void IList.Clear()
BinarySearch	Method	System.Object Clone(), System.Type GetType()
Clear	Method	bool Contains(System.Object item), void CopyTo(array array), void Equals(System.Object obj)
Clone	Method	System.Collections.IEnumerator GetEnumerator()
Contains	Method	int GetHashCode()
CopyTo	Method	System.Collections.ArrayList G
Equals	Method	int IndexOf(System.Object value), void Insert(int index, System.Object item), void InsertRange(int index, int count)
GetEnumerator	Method	int LastIndexOf(System.Object value), void Remove(System.Object obj), void RemoveAt(int index), void RemoveRange(int index, int count)
GetHashCode	Method	void Reverse(), void Sort()
GetRange	Method	string ToString()
GetType	Method	void TrimToSize()
IndexOf	Method	System.Object Item(int index)
Insert	Method	int Capacity {get;set;}
InsertRange	Method	int Count {get;}
LastIndexOf	Method	bool IsFixedSize {get;}
Remove	Method	bool IsReadOnly {get;}
RemoveAt	Method	bool IsSynchronized {get;}
RemoveRange	Method	System.Object SyncRoot {get;}
Reverse	Method	
SetRange	Method	
Sort	Method	
ToArray	Method	
ToString	Method	
TrimToSize	Method	
Item	ParameterizedProperty	
Capacity	Property	
Count	Property	
IsFixedSize	Property	
IsReadOnly	Property	
IsSynchronized	Property	
SyncRoot	Property	

Après nous être intéressés au contenant, intéressons-nous au contenu. Une erreur est un objet de type `ErrorRecord` dont voici les membres :

PS> \$error[0]   Get-Member	
TypeName	Definition
System.Management.Automation.ErrorRecord	

Name	MemberType	Definition
----	-----	-----
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetObjectData	Method	void GetObjectData(System.Runtime.
GetType	Method	type GetType()
Tostring	Method	string ToString()
writeErrorStream	NoteProperty	System.Boolean writeErrorStream=Tr
CategoryInfo	Property	System.Management.Automation.Error
ErrorDetails	Property	System.Management.Automation.Error
Exception	Property	System.Exception Exception {get;}
FullyQualifiedErrorId	Property	string FullyQualifiedErrorId {get;
InvocationInfo	Property	System.Management.Automation.Invoc
PipelineIterationInfo	Property	System.Collections.ObjectModel.Rea
ScriptStackTrace	Property	string ScriptStackTrace {get;}
TargetObject	Property	System.Object TargetObject {get;}
PSMessageDetails	ScriptProperty	System.Object PSMessageDetails {ge

La structure d'un objet `ErrorRecord` contient un certain nombre de propriétés qu'il est intéressant d'examiner :

- **CategoryInfo**

Classe les erreurs rencontrées en plusieurs grandes catégories.

- **ErrorDetails**

Fournit une description plus détaillée de l'erreur. Peut être nulle.

- **Exception**

Correspond à l'exception en lien avec l'erreur.

- **FullyQualifiedErrorId**

Identifie l'erreur de manière très précise.

- **InvocationInfo**

Donne des informations concernant le contexte où l'erreur est apparue, comme le nom du script, la ligne de commande, la ligne ayant provoqué l'erreur, etc.

- **TargetObject**

Décrit l'objet actif au moment où l'erreur a été provoquée. Peut être nulle.

- **PipelineIterationInfo**

Indique le statut du pipeline au moment où l'erreur a été provoquée.

Toutes ces informations sont cruciales et savoir les manipuler est une condition indispensable en matière de gestion d'erreurs et accroît incontestablement l'efficacité et la puissance des scripts que nous écrivons.

## Le paramètre `-ErrorAction` et la variable `$ErrorActionPreference`

Dans l'écosystème PowerShell, les cmdlets disposent d'un jeu de paramètres courants, ajoutés dynamiquement par le runtime dans un processus complexe que nous ne décrirons pas dans ce livre. Parmi ces paramètres courants, `-ErrorAction` détermine comment les cmdlets vont répondre prioritairement à des erreurs de type `nonterminating`. Les arguments valides de ce paramètre sont listés dans le tableau suivant.

**Tableau 14-1** Arguments possibles du paramètre `-ErrorAction`

Argument	Description
<code>SilentlyContinue</code>	Supprime le message d'erreur et continue l'exécution de la commande.
<code>Continue</code>	Affiche le message d'erreur à l'écran et continue l'exécution de la commande. C'est la valeur par défaut.
<code>Inquire</code>	Affiche le message d'erreur et invite l'utilisateur à confirmer ou non la poursuite de l'exécution de la commande.
<code>Stop</code>	Affiche le message d'erreur puis arrête automatiquement l'exécution de la commande.
<code>Ignore</code>	Produit le même comportement que la valeur <code>SilentlyContinue</code> , mais sans incrémenter la variable automatique <code>\$Error</code> .

Pour illustrer ce que nous venons d'évoquer, essayons d'obtenir des informations BIOS sur deux machines distantes :

```
PS> Get-WmiObject -Class Win32_Bios 'm10307','m10308'

Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT : 0x800706BA)
Au caractère Ligne:1 : 1
+ Get-WmiObject -Class Win32_Bios -ComputerName 'm10307','m10308'
+ ~~~~~
+ CategoryInfo          : InvalidOperation : () [Get-WmiObject],
COMException
+ FullyQualifiedErrorCode : GetWMICOMException,Microsoft.PowerShell.Commands
.GetWmiObjectCommand

SMBIOSBIOSVersion : VirtualBox
Manufacturer      : innotek GmbH
Name              : default System Bios
SerialNumber      : 0
Version           : VBOX - 1
```

La sortie nous montre que la commande n'a réussi que sur la deuxième cible. On constate que l'erreur s'affiche à l'écran, mais sans empêcher PowerShell de continuer l'exécution. En observant attentivement la ligne de commande, on remarque que le paramètre `-ErrorAction` n'a pas été invoqué. Ce qui définit ce comportement par défaut est ici une variable de préférence nommée `$ErrorActionPreference`. Cette variable a exactement le même comportement que le paramètre `-ErrorAction`, sauf que ce dernier influe sur la commande, tandis que la variable de préférence influence quant à elle l'ensemble de l'environnement. La valeur par défaut de cette variable est `Continue` :

```
PS> $ErrorActionPreference  
Continue
```

Cela explique le comportement de PowerShell dans l'exemple précédent. Pour modifier ce comportement de manière circonstanciée, la paramètre `-ErrorAction` doit être invoqué. Reprenons notre commande :

```
PS> Get-WmiObject -Class Win32_Bios 'm10307','m10308' -ErrorAction 'Stop'  
Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT :  
0x800706BA)  
Au caractère Ligne:1 : 1  
+ Get-WmiObject -Class Win32_Bios -ComputerName 'm10307','m10308'  
+ ~~~~~  
+ CategoryInfo          : InvalidOperation : () [Get-WmiObject],  
COMException  
+ FullyQualifiedErrorId : GetWMICOMException,Microsoft.PowerShell.Commands  
.GetWmiObjectCommand
```

Le paramètre `-ErrorAction` a pour valeur `Stop`, ce qui entraîne l'arrêt de l'exécution de la part de PowerShell. La valeur de la variable de préférence `$ErrorActionPreference` a été remplacée par celle du paramètre `-ErrorAction` ; toutefois, ceci n'a d'effet que pour l'opération actuelle et non pas pour les opérations ultérieures.

Nous avons vu jusqu'à présent seulement des erreurs de type `nonterminating`, qui ne provoquent pas directement l'arrêt de l'exécution du runtime. Ces erreurs ne constituent pas dans l'absolu des éléments bloquants, elles sont simplement affichées. Cependant, il est possible de les gérer, c'est-à-dire de faire quelque chose lorsqu'elles se manifestent ; et cela est évidemment indispensable en ce qui concerne les exceptions.

## L'instruction trap

Depuis le début du chapitre, nous avons évoqué la nécessité de distinguer les erreurs des exceptions. Le fait est que PowerShell sait mieux réagir aux premières qu'aux secondes. Ces exceptions peuvent cependant être gérées en utilisant des instructions spéciales. En PowerShell, il y en a deux : l'instruction `trap`, qui est l'objet de cette section et dont nous synthétiserons l'utilisation devenue obsolète, puis l'instruction `try..catch..finally` que nous étudierons dans la section suivante et qui représente quant à elle l'orientation à privilégier en matière de gestion d'exceptions.

Le principe de l'instruction `trap` est qu'une liste d'instructions est exécutée lorsqu'une exception se produit. Dès lors, PowerShell peut continuer ou arrêter l'exécution. Sa syntaxe s'articule comme suit :

```
trap [[type d'erreur]] ❶ {<liste d'instructions>} ❷
```

Lorsqu'une exception se produit, PowerShell exécute la liste d'instructions ❷. Par défaut, il s'agit de tout type d'erreurs, mais il est possible de spécifier un type d'erreur particulier ❶.

Il n'y a pas de limite quant au nombre d'instructions `trap` dans un script ou dans tout autre commande. L'exemple suivant montre une simple fonction exécutant une commande mal écrite :

```
function Get-Spec {
    trap {
        "Error found: $_"
        continue
    }
    get-mxmsdcl
}
```

La fonction `Get-Spec` contient l'instruction de gestion des exceptions `trap` et exécute une commande nommée `get-mxmsdcl` qui, comme on pourrait le prévoir, est censée causer l'exception forçant PowerShell à se rediriger vers `trap`. Notez que la variable automatique `$_` contient l'erreur produite et que l'utilisation du mot-clé `continue` laisse PowerShell continuer son exécution sans écrire l'erreur dans le flux.

```
PS> Get-Spec
```

Error found: Le terme « get-mxmsdcl » n'est pas reconnu comme nom d'applet de commande, fonction, fichier de script ou programme exécutable. Vérifiez l'orthographe du nom, ou si un chemin d'accès existe, vérifiez que le chemin d'accès est correct et réessayez.

L'instruction `trap` peut aussi cibler des types d'erreurs spécifiques :

```
trap [System.DivideByZeroException] {
    "DivideByZero exception trapped !!!"
    continue
}

trap [System.Management.Automation.CommandNotFoundException] {
    "CommandNotFound exception trapped !!!"
    continue
}

trap [System.FormatException] {
    "Format exception trapped !!!"
    continue
}

PS> 1/0
DivideByZero exception trapped !!!
```

Cet exemple met en évidence trois instructions `trap` ciblant respectivement des types d'erreurs bien spécifiques :

- `System.DivideByZeroException`  
Exceptions lancées lorsqu'on tente de diviser un nombre par zéro.
- `System.Management.Automation.CommandNotFoundException`  
Exceptions concernant les commandes inexistantes.
- `SystemFormatException`  
Exceptions liées aux formats des données.

Ensuite, on tente une division par zéro, ce qui provoque une exception, mais pas n'importe laquelle. Sur l'ensemble des instructions `trap` déclarées et implémentées, PowerShell a correctement détecté celle à exécuter. Cette dernière intercepte donc l'erreur et affiche le message attendu.

L'instruction `trap` existe depuis la version 1 de PowerShell. À l'époque, la seule façon de gérer les exceptions était justement de passer par cette construction. La version 2 a inauguré une autre alternative, héritée du langage C# et dont Microsoft recommande l'utilisation : l'instruction `try..catch..finally`.

## L'instruction try..catch..finally

La seconde possibilité de gérer les exceptions en PowerShell est l'instruction `try..catch..finally`, qui autorise une gestion d'erreurs plus fine :

```
try ①{  
    <liste d'instructions> ②  
} catch ③ [[type d'erreur]] ④ {  
    <liste d'instructions> ⑤  
} finally ⑥ {  
    <liste d'instructions> ⑦  
}
```

Le bloc `try` ① est utilisé pour définir une liste d'instructions ② dont les erreurs doivent être anticipées par PowerShell. Lorsqu'une erreur se produit, cette dernière est d'abord incrémentée dans le tableau `$Error`, puis PowerShell vérifie l'existence d'un bloc `catch` ③ dont le but est de gérer l'erreur en question. Le type d'erreur ④ à gérer, qui est une exception Microsoft .NET, peut être spécifique et signalé entre crochets. S'il n'y a pas de bloc `catch`, alors PowerShell recherche un bloc `catch` ou même une instruction `trap` dans les portées parentes. Si un bloc `catch` existe, PowerShell exécute le bloc d'instructions ⑤ correspondant. Enfin, dans tous les cas de figure, la liste d'instructions ⑦ liée au bloc `finally` ⑥ est exécutée.

### NOTE Concernant les blocs catch et finally

Une instruction `try..catch..finally` peut inclure plusieurs blocs `catch` et il n'y a pas de limite en la matière. En ce qui concerne le bloc `finally` en revanche, il ne peut y en avoir que zéro ou un.

L'exemple suivant illustre une utilisation simple de l'instruction `try..catch` :

```
try {  
    get-mxmsdc1  
} catch {  
    write-host "une erreur s'est produite !!"  
}  
  
une erreur s'est produite !!
```

Ici, PowerShell tente d'exécuter une commande, mais n'y arrive pas. Le script provoque donc une exception, amenant PowerShell à exécuter la liste d'instructions à l'intérieur du bloc `catch`.

Lorsqu'une instruction `try..catch..finally` comprend plusieurs blocs `catch`, il est recommandé de gérer des erreurs spécifiques :

```
try {
    1/0
} catch [System.DivideByZeroException] {
    "DivideByZero exception !!!"
    continue
} catch [System.Management.Automation.CommandNotFoundException] {
    "CommandNotFound exception !!!"
    continue
} catch [System.FormatException] {
    "Format exception !!!"
    continue
} finally {
    "Finished !!!"
}

DivideByZero exception !!!
Finished !!!
```

Cet exemple met en évidence plusieurs blocs `catch`. PowerShell exécute le bloc `try`. La division d'un entier par zéro n'étant pas possible, une exception est produite et PowerShell recherche l'existence d'un bloc `catch` correspondant, qu'il exécute. Enfin, le bloc `finally` est déroulé à son tour.

En matière de gestion d'erreurs, la précision est capitale, car elle nous permet d'anticiper un type d'erreur par rapport à un autre. En outre, le code écrit devient plus clair et plus facilement traçable. L'instruction `try..catch..finally` est, par rapport à son alter ego `trap`, une bien meilleure solution apportant plus de clarté.



## PARTIE 3

# **PowerShell en pratique**



# 15

## L'infrastructure WMI et CIM

---

*PowerShell fait preuve de cohérence, jusque dans les composants qu'il utilise, comme WMI (Windows Management Instrumentation). WMI est apparu à une époque où l'administration des composants reposait sur des axes multiples et où le besoin d'homogénéisation se faisait croissant. Depuis ses débuts, PowerShell fournit une abstraction importante pour utiliser WMI. Cette abstraction autorise une certaine simplicité que l'on ne retrouve évidemment pas dans d'autres langages comme C++ ou Visual Basic .NET.*

*Nous commencerons dans ce chapitre par nous familiariser avec le concept WMI et ce qu'il implique. Puis nous explorerons les possibilités offertes par PowerShell en matière de collecte et de manipulation de données via WMI, pour enfin nous tourner vers les cmdlets CIM (Common Information Model) représentant une voie vers laquelle les utilisateurs devront progressivement s'orienter.*

### SOMMAIRE

- ▶ Définir ce qu'est WMI
- ▶ Utiliser WMI
- ▶ Utiliser les cmdlets CIM

## Définir ce qu'est WMI

WMI (*Windows Management Instrumentation*) est l'implémentation Microsoft du standard CIM (*Common Information Model*). Ce standard, créé par une organisation nommée *Distributed Management Task Force* (DMTF) est en quelque sorte une modélisation universelle quant à l'administration des différents composants concernant les systèmes, réseaux, applications et services. L'objectif de ce standard est de fournir une cohérence très forte dans la façon dont les composants doivent être gérés.

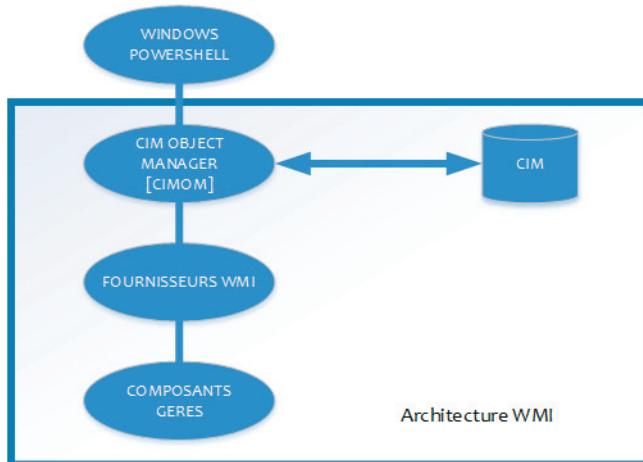
WMI s'articule essentiellement autour de classes organisées en espaces de noms.

- Une classe représente tout ou partie d'un composant gérable, qu'il soit logiciel ou matériel. Ce peut être par exemple un disque, une carte réseau ou encore un service. Par exemple, la classe [\[Win32\\_DiskDrive\]](#) représente un disque physique. La classe [\[Win32\\_Directory\]](#) représente quant à elle un répertoire.
- Un espace de noms sert à organiser ces classes de manière logique et cohérente. En effet, il en existe des centaines car, outre Windows, d'autres technologies comme Exchange Server ou SQL Server apportent chacune son lot de composants gérables à travers WMI. La multiplicité des composants administrables entraîne donc une augmentation conséquente de classes disponibles pour gérer ces composants, ce qui requiert une certaine organisation. L'espace de noms le plus utilisé dans le cadre de l'administration de système est [root\cimv2](#). Il regroupe l'ensemble des classes fréquemment utilisées et constituant le cœur du système en termes d'administration.

Le schéma suivant illustre l'articulation liant PowerShell et l'infrastructure WMI.

**Figure 15-1**

Modèle de l'infrastructure WMI



Lorsqu'une requête WMI est transmise via PowerShell, certaines étapes ont lieu dans un ordre précis :

- 1 Tout d'abord, la requête est dirigée vers CIMOM (*CIM Object Manager*), qui est un composant du service WMI.
- 2 Le composant CIMOM sait comment rediriger la requête, et surtout vers le ou les bon(s) fournisseur(s) WMI.
- 3 Les fournisseurs WMI exposent un certain nombre de classes destinées à gérer les composants matériels ou système.
- 4 La réponse à la requête passe de nouveau par CIMOM qui redirige une nouvelle fois le résultat vers le *consumer* adéquat (en l'occurrence PowerShell).

Utiliser WMI implique de bien connaître les classes que nous manipulons. La documentation fait donc partie intégrante de l'utilisation de ce service, car un objet WMI est une entité à part entière avec des propriétés et des méthodes. La commande suivante utilise la cmdlet `Get-WmiObject` (que nous étudierons dans la section suivante) afin d'instancier la classe `Win32_Processor`. L'objet créé est ensuite envoyé à la cmdlet `Get-Member` pour en lister les membres :

```
PS> Get-WmiObject -Class Win32_Processor | Get-Member
```

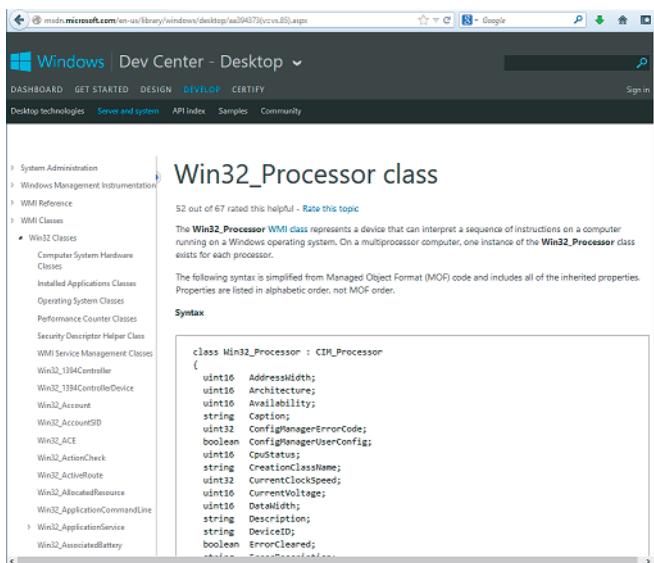
```
TypeName : System.Management.ManagementObject#root\cimv2\Win32_Proc
```

Name	MemberType	Definition
PSComputerName	AliasProperty	PSComputerName =
Reset	Method	System.Manageme
SetPowerState	Method	System.Manageme
AddressWidth	Property	uint16 AddressWi
Architecture	Property	uint16 Architect
Availability	Property	uint16 Availabil
Caption	Property	string Caption {
ConfigManagerErrorCode	Property	uint32 ConfigMan
ConfigManagerUserConfig	Property	bool ConfigManag
CpuStatus	Property	uint16 CpuStatus
CreationClassName	Property	string CreationC
CurrentClockSpeed	Property	uint32 CurrentCl
CurrentVoltage	Property	uint16 CurrentVo
DataWidth	Property	uint16 DataWidth
Description	Property	string Descripti
DeviceID	Property	string DeviceID
ErrorCleared	Property	bool ErrorCleare
ErrorDescription	Property	string ErrorDesc
ExtClock	Property	uint32 ExtClock
Family	Property	uint16 Family {g
InstallDate	Property	string InstallDa

L2CacheSize	Property	uint32	L2CacheSi
L2CacheSpeed	Property	uint32	L2CacheSp
L3CacheSize	Property	uint32	L3CacheSi
L3CacheSpeed	Property	uint32	L3CacheSp
LastErrorCode	Property	uint32	LastError
Level	Property	uint16	Level {ge
LoadPercentage	Property	uint16	LoadPerce
Manufacturer	Property	string	Manufactu
MaxClockSpeed	Property	uint32	MaxClockS
Name	Property	string	Name {get
NumberOfCores	Property	uint32	NumberOfC
NumberOfLogicalProcessors	Property	uint32	NumberOfL
OtherFamilyDescription	Property	string	OtherFami
PNPDeviceID	Property	string	PNPDevice
PowerManagementCapabilities	Property	uint16[]	PowerMa
PowerManagementSupported	Property	bool	PowerManage
ProcessorId	Property	string	Processor
ProcessorType	Property	uint16	Processor
Revision	Property	uint16	Revision
Role	Property	string	Role {get
SecondLevelAddressTranslationExtensions	Property	bool	SecondLevel
SocketDesignation	Property	string	SocketDes
Status	Property	string	Status {g
StatusInfo	Property	uint16	StatusInf
Stepping	Property	string	Stepping
SystemCreationClassName	Property	string	SystemCre
SystemName	Property	string	SystemNam
UniqueId	Property	string	UniqueId
UpgradeMethod	Property	uint16	UpgradeMe
Version	Property	string	Version {
VirtualizationFirmwareEnabled	Property	bool	Virtualizat
VMMonitorModeExtensions	Property	bool	VMMonitorMo
VoltageCaps	Property	uint32	VoltageCa
__CLASS	Property	string	__CLASS {
__DERIVATION	Property	string[]	__DERIV
__DYNASTY	Property	string	__DYNASTY
__GENUS	Property	int	__GENUS {get
__NAMESPACE	Property	string	__NAMESPA
__PATH	Property	string	__PATH {g
__PROPERTY_COUNT	Property	int	__PROPERTY_C
__RELPATH	Property	string	__RELPATH
__SERVER	Property	string	__SERVER
__SUPERCLASS	Property	string	__SUPERCL
PSConfiguration	PropertySet	PSConfiguration	
PSStatus	PropertySet	PSStatus	{Availa
ConvertFromDateTime	ScriptMethod	System.Object	Co
Convert.ToDateTime	ScriptMethod	System.Object	Co

Comme nous pouvons le constater, la liste des membres est importante et certains d'entre eux sont quasi mystiques. Microsoft fournit une documentation en ligne importante, répertoriant l'ensemble des classes WMI. Le site MSDN (*Microsoft Developer Network*) peut notamment nous aider dans l'exploration de notre fameuse classe [\[Win32\\_Processor\]](#).

**Figure 15–2**  
Documentation WMI  
accessible en ligne



L'étape de la documentation est très importante, car on n'imagine pas le nombre de dysfonctionnements pouvant survenir à la suite de mauvaises manipulations. L'infrastructure WMI requiert, tout comme .Net, un certain temps d'adaptation avant de parvenir à une bonne maîtrise.

## Utiliser WMI

Vous l'aurez compris, WMI est une base de données extrêmement vaste. L'utiliser fournit des informations précieuses comme :

- la configuration réseau ;
- des informations processeur ;
- la version du système d'exploitation ;
- la version du Service Pack ;
- les services actuellement en cours d'exécution ;

- la liste des logiciels installés ;
- le type de carte graphique ;
- etc.

## Les cmdlets WMI

PowerShell permet de collecter et de modifier ces informations via un ensemble de cmdlets qui sont décrites dans le tableau suivant.

**Tableau 15-1** Liste des cmdlets interagissant avec le service WMI

Cmdlet	Description
Get-WmiObject	Obtient des objets actifs WMI ou des informations sur les classes WMI disponibles dans un espace de noms donné.
Invoke-WmiMethod	Instancie les méthodes d'objets WMI.
Register-WmiEvent	Abonne à un événement WMI.
Remove-WmiObject	Supprime un objet WMI existant.
Set-WmiInstance	Crée ou modifie une instance d'une classe WMI.

## WMI en pratique

La cmdlet sans doute la plus utilisée parmi celles que nous venons de mentionner est `Get-WmiObject`. En effet, elle fournit des instances de classes non seulement au niveau de l'ordinateur local, mais aussi sur des machines distantes :

```
PS> Get-WmiObject -Class Win32_LogicalDisk -ComputerName 'localhost'
```

```
DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 773249114112
Size          : 972008976384
VolumeName    : Windows8_OS
```

```
PS> Get-WmiObject -Class Win32_LogicalDisk -ComputerName 'SRV2008ISS'
```

```
DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 5035732992
Size          : 15999168512
VolumeName    :
```

Le paramètre `-ComputerName`, qui autorise les requêtes à distance, n'utilise pas la communication à distance PowerShell. Ceci implique qu'il n'est pas nécessaire que les machines distantes répondent aux conditions exigées par la communication à distance PowerShell. De plus, le nom de la machine distante peut être soit un nom NETBIOS, soit un nom FQDN (*Fully Qualified Domain Name*) ou même une adresse IP :

```
PS> Get-WmiObject -Class Win32_LogicalDisk -ComputerName 10.0.4.12
```

La cmdlet `Get-WmiObject` dispose d'un paramètre nommé `-NameSpace` spécifiant l'espace de noms à partir duquel la requête doit être effectuée :

```
PS> Get-WmiObject -NameSpace 'Root\Cimv2' -Class Win32_LogicalDisk  
-ComputerName 'localhost'
```

Le paramètre `-Filter` spécifie une clause qui sera utilisée pour filtrer le résultat d'une requête :

```
PS> Get-WmiObject -NameSpace 'Root\Cimv2' -Class Win32_LogicalDisk  
-ComputerName 'localhost' -Filter "Drivetype='3'"
```

Lorsque WMI est utilisé, il n'est pas rare de rencontrer des erreurs liées à l'authentification. Cela vient du fait que l'utilisateur n'a pas les permissions appropriées. Dans ce cas de figure, la cmdlet `Get-WmiObject` fournit un paramètre (`-Credential`) spécifiant un compte d'utilisateur ayant l'autorisation d'exécuter l'action voulue :

```
PS> Get-WmiObject -NameSpace 'Root\Cimv2' -Class Win32_LogicalDisk  
-ComputerName 'localhost' -Filter "Drivetype='3'" -Credential  
Domaine\Nom_Utilisateur
```

PowerShell invitera aussitôt l'utilisateur à entrer un mot de passe pour poursuivre l'opération. Si les informations passées au paramètre `-Credential` sont fausses, alors PowerShell le signalera à l'utilisateur.

Le nombre de classes WMI qui existent est très important. Pour connaître les classes disponibles dans un espace de noms, la cmdlet `Get-WmiObject` fournit le paramètre `-List`. Essayons de lister les classes dans l'espace de noms `Root\Microsoft` :

```
PS> Get-WmiObject -Namespace 'Root\Microsoft' -List | select Name
```

Name
---
__SystemClass

```
__thisNAMESPACE
__ProviderRegistration
__EventProviderRegistration
__ObjectProviderRegistration
__ClassProviderRegistration
__InstanceProviderRegistration
__MethodProviderRegistration
__PropertyProviderRegistration
__EventConsumerProviderRegistration
__NAMESPACE
__IndicationRelated
__FilterToConsumerBinding
__EventConsumer
__AggregateEvent
__TimerNextFiring
__EventFilter
__Event
__NamespaceOperationEvent
__NamespaceModificationEvent
__NamespaceDeletionEvent
__NamespaceCreationEvent
__ClassOperationEvent
__ClassDeletionEvent
__ClassModificationEvent
__ClassCreationEvent
__InstanceOperationEvent
__InstanceCreationEvent
__MethodInvocationEvent
__InstanceModificationEvent
__InstanceDeletionEvent
__TimerEvent
__ExtrinsicEvent
...
...
```

Comme nous pouvons le voir, la liste est impressionnante. La cmdlet `Get-WmiObject` sera donc un outil très précieux dans le processus de découverte des classes disponibles.

Parfois, il est nécessaire de modifier les objets eux-mêmes, en plus de les lister, ce qui se fait à l'aide de la cmdlet `Invoke-WmiMethod`. Par exemple, la commande suivante lance la calculatrice Windows :

```
PS> Invoke-WmiMethod -path Win32_Process -name Create -argumentList Calc.exe
```

```
__GENUS      : 2
__CLASS      : __PARAMETERS
__SUPERCLASS : :
```

```
__DYNASTY      : __PARAMETERS
__RELPATH      :
__PROPERTY_COUNT : 2
__DERIVATION    : {}
__SERVER        :
__NAMESPACE     :
__PATH          :
ProcessId      : 2992
ReturnValue     : 0
PSComputerName :
```

Ici, nous avons sollicité la méthode statique (c'est-à-dire une méthode disponible sans que la classe en question ne soit instanciée) `Create()` pour créer un processus `Calc.exe`. Le paramètre `-name` spécifie le nom de la méthode à appeler et le paramètre `-argumentlist` indique quant à lui les paramètres à passer à la méthode appelée.

Les objets WMI créés peuvent être supprimés, à l'aide de `Remove-WmiObject` :

```
PS> $calc = get-wmiobject -query "select * from win32_process where
  name='calc.exe'"
PS> $calc | remove-wmiobject
```

Cet exemple met en évidence la récupération des instances existantes à l'aide du paramètre `-query` autorisant l'utilisation du langage de requête WMI (WQL). Le résultat de la requête est stocké dans une variable nommée `$calc`. Enfin, la variable est passée à la cmdlet `Remove-WmiObject` qui supprime toutes les instances du processus `Calc.exe`.

L'infrastructure WMI demande un certain temps d'adaptation. Les classes issues des différents espaces de noms sont utilisées de manière différente. Il n'y a pas d'approche homogène en la matière. Donc, la recherche et la documentation seront des conditions sine qua non à la réussite des commandes utilisant WMI. Le paradoxe de WMI réside dans sa simplicité d'utilisation et dans la difficulté d'une documentation éparsse.

## Utiliser les cmdlets CIM

Microsoft, parallèlement à l'arrivée de PowerShell version 3, a défini une nouvelle interface donnant accès à WMI (qui pour rappel repose sur le modèle CIM). Elle modifie de manière radicale le mode d'accès aux données. Pour le moment néanmoins, elle ne remplace pas celle que nous connaissons déjà et qui est basée sur la technologie COM (*Component Object Model*). Donc, les cmdlets WMI sont évidem-

ment encore d'actualité, mais un nouvel ensemble de cmdlets a été développé et est conforme à ce nouveau standard.

## Les cmdlets CIM

Les cmdlets CIM symbolisent cette nouvelle génération de cmdlets en lien avec le modèle CIM/WMI. Elles ne remplacent pas les cmdlets WMI, mais leur utilisation est à privilégier de manière progressive.

**Tableau 15–2** Liste non exhaustive des cmdlets CIM

Cmdlet	Description
Get-CimClass	Fournit une liste de classes CIM à partir d'un espace de noms donné.
Get-CimInstance	Fournit les instances de classe CIM de manière locale ou distante.
Get-CimSession	Liste les sessions CIM créées dans la session en cours.
Get-CimAssociatedInstance	Obtient les instances CIM associées à une instance CIM spécifique.
Invoke-CimMethod	Instancie une méthode de classe CIM.
New-CimInstance	Instancie une classe CIM.
New-CimSession	Crée une session CIM.
New-CimSessionOption	Spécifie des options s'appliquant à une session CIM.
Remove-CimSession	Supprime une session CIM existante.

## CIM en action

La symétrie entre les cmdlets CIM et WMI est très frappante. Il est même possible d'utiliser des classes WMI avec des cmdlets CIM comme l'illustre bien la cmdlet `Get-CimClass` :

```
PS> Get-CimClass -ClassName Win32_LogicalDisk
NameSpace : ROOT/cimv2
CimClassName      CimClassMethods      CimClassProperties
-----          -----
Win32_LogicalDisk {SetPowerState, R... {Caption, Description,...}
```

La cmdlet `Get-CimInstance` retourne les instances existantes d'une classe CIM ou WMI :

```
PS> Get-CimInstance -ClassName Win32_Process
```

ProcessId	Name	HandleCount	WorkingSetSize	VirtualSize
0	System Idle Process	0	20480	65536
4	System	986	13238272	17281024
296	smss.exe	37	1007616	4460544
560	csrss.exe	253	4218880	50212864
664	wininit.exe	73	3629056	42418176
672	csrss.exe	366	7745536	97574912
712	services.exe	269	8232960	44306432
736	winlogon.exe	143	7921664	53428224
744	lsass.exe	1001	11304960	40128512
852	svchost.exe	372	8773632	41041920

Pour invoquer une méthode liée à une classe [WIN32], nous pouvons utiliser la cmdlet `Invoke-CimMethod` :

```
PS> Invoke-CimMethod -ClassName Win32_Process -MethodName "Create" -Arguments @{
    commandline = "calc.exe" }
```

ProcessId	ReturnValue	PSComputerName
3596	0	

La méthode statique `Create()` de la classe `Win32_Process` est passée au paramètre `-MethodName` et l'argument de cette même méthode au paramètre `-Arguments` sous forme de dictionnaire.

Il est aussi possible de créer des sessions de type CIM (qui sont différentes des sessions PowerShell classiques) via la cmdlet `New-CimSession`. Commençons par initialiser notre session :

```
PS> $CimSession = New-CimSession -ComputerName 'SRV2012DC'
```

Maintenant, essayons d'obtenir la liste des services actuellement en cours d'exécution sur la machine distante, mais en nous basant sur la session CIM créée :

```
PS> Get-CimInstance -CimSession $CimSession -ClassName Win32_Service | Select
    processid,name,status,exitcode
```

processid	name	status	exitcode
1716	AdobeARMservice	OK	0
0	AdobeFlashPlayerUpdateSvc	OK	1077

```
0 AeLookupSvc          OK      1077
0 ALC                 OK      1077
0 AllUserInstallAgent OK      1077
944 AMD External Events Utility OK      0
1736 AMD FUEL Service   OK      0
0 AppIDSvc            OK      1077
1012 Appinfo           OK      0
0 aspnet_state         OK      1077
1064 AudioEndpointBuilder OK      0
988 Audiosrv          OK      0
0 AxInstSV             OK      1077
0 BDESVC               OK      1077
1524 BFE                OK      0
...
...
```

La sortie nous montre que la commande a été exécutée avec succès. Si nous n'avons plus besoin de notre session CIM, nous la supprimons avec la cmdlet `Remove-CimSession` :

```
PS> Remove-CimSession -CimSession $Cimsession
```

Les cmdlets CIM, dans leur structure, produisent des effets similaires aux cmdlets WMI. Cependant, la stratégie à terme est, en plus de s'adresser au monde Windows, de pouvoir administrer d'autres infrastructures reposant sur des systèmes différents.

# 16

## Gérer les services

---

*Dans un système d'exploitation, la gestion des services est centrale. En effet, conjugués aux processus, les services maintiennent dans l'absolu le bon déroulement de l'exécution des applications. Par conséquent, savoir les reconnaître, pour mieux les gérer, est une base fondamentale de l'administration d'un système d'exploitation.*

*PowerShell fournit un ensemble de commandes pour gérer facilement les services. Dans ce chapitre, nous apprendrons d'abord comment les lister. Un service, étant un composant administrable, peut avoir différents états vis-à-vis desquels nous nous focaliserons dans un second temps. Nous verrons aussi comment gérer le mode de démarrage d'un service, étape importante de sa configuration. Enfin, nous terminerons en apprenant comment modifier le compte de connexion d'un service.*

### SOMMAIRE

- ▶ Lister les services
- ▶ Démarrer, arrêter, redémarrer et interrompre l'exécution d'un service
- ▶ Configurer le mode de démarrage d'un service
- ▶ Modifier le compte de connexion d'un service

## Lister les services

Lister les services est une pratique extrêmement répandue dans le monde de l'administration système, même si les méthodes changent d'un système à un autre. Pour obtenir une liste des services présents sur une machine locale ou distante, il faut utiliser la cmdlet `Get-Service`, que nous avons employée de nombreuses fois dans cet ouvrage :

```
PS> Get-Service -ComputerName 'MLWIN8'

Status      Name                DisplayName
----      ----
Running    AdobeARMservice    Adobe Acrobat Update Service
Stopped   AdobeFlashPlaye...  Adobe Flash Player Update Service
Stopped   AeLookupSvc        Expérience d'application
Stopped   ALG                 Service de la passerelle de la couc...
Stopped   AllUserInstallA...  Agent d'installation pour tous les ...
Running   AMD External Ev...  AMD External Events Utility
Running   AMD FUEL Service   AMD FUEL Service
Stopped   AppIDSvc           Identité de l'application
Stopped   Appinfo             Informations d'application
Stopped   aspnet_state        ASP.NET State Service
Running   AudioEndpointBu...  Générateur de points de terminaison...
Running   Audiosrv           Audio Windows
Stopped   AxInstSV           Programme d'installation ActiveX (A...
Stopped   BDESVC              Service de chiffrement de lecteur B...
Running   BFE                 Moteur de filtrage de base
Running   BITS                Service de transfert intelligent en...
```

Le paramètre `-ComputerName` peut accepter plusieurs noms de machines et la valeur par défaut est l'ordinateur local. La requête peut évidemment être filtrée, par exemple en articulant `Get-Service` avec la cmdlet `Where-Object` :

```
PS> Get-Service -ComputerName 'MLWIN8' | Where-Object { $_.Status -eq
'Running' }

Status      Name                DisplayName
----      ----
Running    AdobeARMservice    Adobe Acrobat Update Service
Running   AMD External Ev...  AMD External Events Utility
Running   AMD FUEL Service   AMD FUEL Service
Running   AudioEndpointBu...  Générateur de points de terminaison...
Running   Audiosrv           Audio Windows
Running   BFE                 Moteur de filtrage de base
Running   BITS                Service de transfert intelligent en...
```

Running	BrokerInfrastru...	Service d'infrastructure des tâches...
Running	CertPropSvc	Propagation du certificat
Running	CryptSvc	Services de chiffrement
Running	DcomLaunch	Lanceur de processus serveur DCOM
Running	DeviceAssociati...	Service d'association de périphérique
Running	Dhcp	Client DHCP
Running	Dnscache	Client DNS
Running	DPS	Service de stratégie de diagnostic

Un service a souvent des dépendances, au sens actif et au sens passif. Pour lister les services qu'un service particulier requiert, il faut utiliser le paramètre `-RequiredServices` :

```
PS> Get-Service -Name WinRM -RequiredServices
```

Status	Name	DisplayName
-----	-----	-----
Running	RPCSS	Appel de procédure distante (RPC)
Running	HTTP	HTTP

Par ailleurs, il est aussi possible de lister les services qui dépendent d'un service particulier :

```
PS> Get-Service -Name Winmgmt -DependentServices
```

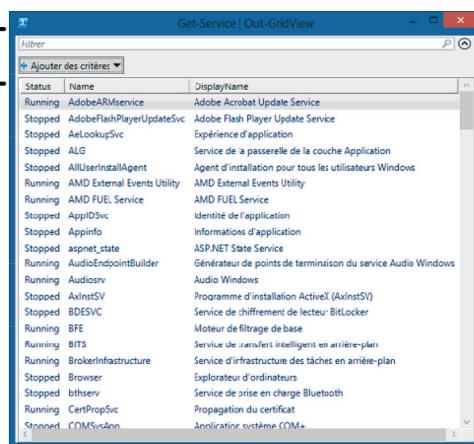
Status	Name	DisplayName
-----	-----	-----
Running	wscsvc	Centre de sécurité
Stopped	SharedAccess	Partage de connexion Internet (ICS)
Stopped	NcaSvc	Assistant Connectivité réseau
Running	iphlpsvc	Assistance IP

Nous pouvons observer que les résultats affichés à l'écran sont ordonnés sous forme de tableaux. Une analyse fine de l'état des services demanderait que ces tableaux soient interactifs. Malheureusement, la cmdlet `Get-Service` ne propose pas cette option, mais cela est possible à l'aide de la cmdlet `Out-GridView`.

Comme nous pouvons le constater, la sortie ne se fait pas en console, mais dans une autre fenêtre, ce qui permet d'agir interactivement dessus. Le mode d'action est basé sur le filtrage de l'affichage, et non pas sur les objets eux-mêmes comme redémarrer un service à distance. Faire la distinction évite ici la confusion.

**Figure 16–1**  
Analyse interactive  
des services

La cmdlet `Out-GridView` utilisée avec `Get-Service` permet une plus grande souplesse dans l'analyse des résultats.



Status	Name	DisplayName
Running	AdobeARMservice	Adobe Acrobat Update Service
Stopped	AdobeFlashPlayerUpdateSvc	Adobe Flash Player Update Service
Stopped	AeLookupSvc	Expérience d'application
Stopped	ALG	Service de passerelle de la couche Application
Stopped	AllUserInstallAgent	Agent d'installation pour tous les utilisateurs Windows
Running	AMD External Events Utility	AMD External Events Utility
Running	AMD FUEL Service	AMD FUEL Service
Stopped	AppIDSvc	Identité de l'application
Stopped	AppInfo	Informations d'application
Stopped	aspnet_state	ASP .NET State Service
Running	AudioEndpointBuilder	Générateur de points de terminaison du service Audio Windows
Running	Audiosrv	Audio Windows
Stopped	AxInstSV	Programme d'installation ActiveX (AxInstSV)
Stopped	BDESVC	Service de chiffrement de lecteur BitLocker
Running	BFE	Moteur de filtrage de base
Running	BITSS	Service de transfert intelligent en arrière-plan
Running	BrokerInfrastructure	Service d'infrastructure des tâches en arrière-plan
Stopped	Browser	Explorateur d'ordinateurs
Stopped	bthserv	Service de prise en charge Bluetooth
Running	CerPropSvc	Propagation du certificat
Stopped	COMSysAnn	Annulation système COM+

## Démarrer, arrêter, redémarrer et interrompre l'exécution d'un service

La section précédente concernait la collecte d'informations d'état au sujet des services. Nous allons maintenant présenter comment agir sur les services eux-mêmes.

### Démarrer un service

Pour démarrer un service, PowerShell fournit la cmdlet `Start-Service` :

```
PS> Start-Service -Name eventlog
PS>
```

`Start-Service` n'affiche aucun élément en sortie. Utilisée avec le paramètre `-PassThru`, la commande retourne l'objet représentant le service :

```
PS> Start-Service -Name eventlog -PassThru
```

Status	Name	DisplayName
-----	-----	-----
Running	eventlog	Journal d'événements Windows

Le paramètre `-InputObject` sert à passer en paramètre un objet représentant un service :

```
PS> $eventlogsvc = Get-Service -Name EventLog
PS> Start-Service -InputObject $eventlogsvc -PassThru

Status      Name            DisplayName
-----      --          -----
Running    EventLog        Journal d'événements Windows
```

## Arrêter un service

La cmdlet pour arrêter un service est `Stop-Service` :

```
PS> $eventlogsvc = Get-Service -Name EventLog
PS> Stop-Service -InputObject $eventlogsvc -PassThru

Status      Name            DisplayName
-----      --          -----
Stopped   EventLog        Journal d'événements Windows
```

Comme nous pouvons le constater, `Stop-Service` utilise les mêmes paramètres que `Start-Service`.

## Redémarrer un service

Redémarrer un service peut consister en la combinaison des deux commandes précédentes, mais cette méthode est peu efficiente. La meilleure approche est d'utiliser la commande `Restart-Service` :

```
PS> Restart-Service -Name EventLog -PassThru

Status      Name            DisplayName
-----      --          -----
Running   EventLog        Journal d'événements Windows
```

Dans son mode de fonctionnement, la commande `Restart-Service` arrête, puis démarre un service spécifié. Si le service en question est déjà arrêté, alors il sera démarré sans notification d'erreur.

## Interrompre un service en cours d'exécution

Un service, au cours de son exécution, a la possibilité d'être interrompu et de reprendre son action :

```
PS> Suspend-Service -Name LanmanServer
PS>
```

La commande `Suspend-Service` envoie un message d'interruption et le service, tout en étant interrompu dans son action, continue son exécution. Pour que le service reprenne son action, il faut utiliser la cmdlet `Resume-Service` :

```
PS> Resume-Service -Name LanmanServer -PassThru
```

Status	Name	DisplayName
-----	-----	-----
Running	LanmanServer	Serveur

## Configurer le mode de démarrage d'un service

Le mode de démarrage d'un service s'articule autour des valeurs suivantes.

- **Automatique**

Le service démarre parallèlement au démarrage du système.

- **Manuel**

Le service est démarré par le contrôleur de services lorsqu'un autre processus le requiert.

- **Désactivé**

Désactive le service à partir du prochain démarrage du système. Si le service est en cours d'exécution, son action continuera jusqu'à l'arrêt du système.

La cmdlet `Set-Service` modifie le mode de démarrage d'un service de la façon suivante :

```
PS> Set-Service -Name LanmanServer -StartupType Automatic -PassThru
```

Status	Name	DisplayName
-----	-----	-----
Running	LanmanServer	Serveur

Le paramètre `-Name` précise le nom du service et le paramètre `-StartupType` le mode de démarrage du service spécifié. Si l'opération doit être effectuée sur plusieurs machines, ces dernières doivent être listées, séparées par des virgules :

```
PS> Set-Service -Name LanmanServer -StartupType Automatic -PassThru  
-ComputerName 'ML2008R2', 'ML2012PS', 'MLDCX95'
```

En cas de désactivation d'un service spécifique, celui-ci ne sera réellement désactivé qu'au prochain démarrage du système. Pour que l'action prenne effet immédiatement, il faut invoquer le paramètre `-Status` :

```
PS> Set-Service -Name service -StartupType disabled -Status stopped
```

## Modifier le compte de connexion d'un service

Dans des contextes particuliers, il peut être nécessaire de modifier le compte de connexion d'un service. Pour accomplir cette tâche, nous pouvons utiliser l'utilitaire *Services Configuration Utility* qui est riche de fonctionnalités. Néanmoins, dans la perspective de cet ouvrage, nous privilierons la méthode sollicitant WMI. Commençons par créer un objet :

```
PS> [wmi]$Svc=Get-wmiobject -query "Select * from win32_service where name='LanmanServer'"
```

L'objet WMI créé représente le service `LanmanServer`. À présent, nous devons identifier la propriété qui nous renseignera sur le compte de connexion du service `LanmanServer` :

```
PS> $Svc | Get-Member
```

Name	MemberType
---	-----
PSCoputerName	AliasProperty
Change	Method
ChangeStartMode	Method
Delete	Method
GetSecurityDescriptor	Method
InterrogateService	Method
PauseService	Method
ResumeService	Method
SetSecurityDescriptor	Method
StartService	Method
StopService	Method
UserControlService	Method
AcceptPause	Property
AcceptStop	Property
Caption	Property
CheckPoint	Property
CreationClassName	Property
Description	Property

DesktopInteract	Property
DisplayName	Property
ErrorControl	Property
ExitCode	Property
InstallDate	Property
Name	Property
PathName	Property
ProcessId	Property
ServiceSpecificExitCode	Property
ServiceType	Property
Started	Property
StartMode	Property
<b>StartName</b>	Property
State	Property
Status	Property
SystemCreationClassName	Property
SystemName	Property
TagId	Property
WaitHint	Property
__CLASS	Property
__DERIVATION	Property
__DYNASTY	Property
__GENUS	Property
__NAMESPACE	Property
__PATH	Property
__PROPERTY_COUNT	Property
__RELPATH	Property
__SERVER	Property
__SUPERCLASS	Property
PSConfiguration	PropertySet
PSStatus	PropertySet
ConvertFromDateTime	ScriptMethod
Convert.ToDateTime	ScriptMethod

Une étude attentive des propriétés de l'objet `$Svc` nous montre la propriété `StartName`, qui contient l'information que nous cherchons :

```
PS> $Svc.StartName  
LocalSystem
```

La méthode `Change()` va nous aider à modifier ce compte de connexion. En lisant la documentation sur le site MSDN, on s'aperçoit qu'elle prend plusieurs paramètres.

**Figure 16–2**

Paramètres de la méthode Change() de la classe WMI [Win32\_Service]

## Change method of the Win32\_Service class

3 out of 3 rated this helpful - Rate this topic

The **Change** WMI class method modifies a **Win32\_Service**. The **Win32\_LoadOrderGroup** parameter represents a group of system services that define execution dependencies. The services must be initiated in the order specified by the Load Order Group because the services depend on each other. These dependent services require the presence of the antecedent services to function correctly.

This topic uses Managed Object Format (MOF) syntax. For more information about using this method, see [Calling a Method](#).

### Syntax

```
mof
uint32 Change(
    [in] string DisplayName,
    [in] string PathName,
    [in] uint32 ServiceType,
    [in] uint32 ErrorControl,
    [in] string StartMode,
    [in] boolean DesktopInteract,
    [in] string StartName,
    [in] string StartPassword,
    [in] string LoadOrderGroup,
    [in] string LoadOrderGroupDependencies,
    [in] string ServiceDependencies
);
```

Même si seuls deux paramètres nous intéressent dans cet ensemble, il faudra tout renseigner en utilisant la variable \$Null.

```
PS> $Svc.Change($null,$null,$null,$null,$null,$null,"Domaine\Compte_de_connexion","P@ssw0rd/999",$null,$null,$null)
```

__GENUS	:	2
__CLASS	:	__PARAMETERS
__SUPERCLASS	:	
__DYNASTY	:	__PARAMETERS
__RELPATH	:	
__PROPERTY_COUNT	:	1
__DERIVATION	:	{}
__SERVER	:	
__NAMESPACE	:	
__PATH	:	
<b>ReturnValue</b>	:	0
PSComputerName	:	

### NOTE Concernant l'appel de la méthode Change()

Nous avons indiqué des valeurs nulles pour les six premiers paramètres. Puis les valeurs en lien avec le compte et le mot de passe ont été fournies. Il n'est pas obligatoire de spécifier des valeurs nulles pour les paramètres suivants, car PowerShell les assume en tant que valeurs non modifiées.

La sortie nous montre bien que l'opération s'est déroulée correctement. En effet, la propriété `ReturnValue` a pour valeur `0`.

Changer le compte de connexion d'un service est, il faut tout de même le reconnaître, une opération peu fréquente, mais savoir comment effectuer ce genre d'opération via PowerShell est un plus non négligeable apportant une certaine efficacité.

# 17

## Gérer les logs

---

*Les logs, ou journaux d'événements, constituent une part importante de l'administration d'un système Windows. En effet, ils contiennent des événements plus ou moins importants dans le but d'informer les utilisateurs sur le déroulement de l'exécution du système d'exploitation. Ces informations aident souvent à poser des diagnostics, à établir des rapports ou même résoudre des problèmes bien précis. Ceci implique que ces journaux d'événements doivent être régulièrement lus et analysés afin de tracer les événements importants.*

*PowerShell offre un certain nombre de commandes pour gérer les logs. Évidemment, ces derniers peuvent avoir plusieurs sources et prendre plusieurs formes. Dans la perspective d'une gestion efficace, savoir les manipuler est donc primordial. Dans ce chapitre, nous apprendrons à lire, configurer, sauvegarder et supprimer les logs de manière adéquate. Ces opérations sont fondamentales dans le cadre d'une administration quotidienne.*

### SOMMAIRE

- ▶ Lister les logs
- ▶ Configurer et sauvegarder les logs
- ▶ Supprimer les logs

## Lister les logs

Les journaux d'événements varient d'un système d'exploitation à l'autre. Il est donc fortement recommandé d'essayer de découvrir quels types de journaux sont disponibles. Pour ce faire, nous utiliserons la cmdlet `Get-EventLog` :

```
PS> Get-EventLog -List
```

Max(K)	Retain	OverflowAction	Entries	Log
20	480	0 OverwriteAsNeeded	14	Application
20	480	0 OverwriteAsNeeded	0	HardwareEvents
	512	7 OverwriteOlder	0	Internet Explorer
20	480	0 OverwriteAsNeeded	0	Key Management Service
	128	0 OverwriteAsNeeded	9	OAlerts
				Security
20	480	0 OverwriteAsNeeded	57	System
15	360	0 OverwriteAsNeeded	1	Windows PowerShell

La commande `Get-EventLog` collecte les journaux d'événements présents sur une machine locale ou distante. Elle indique le nombre d'entrées par journal sans rentrer dans les détails. Pour explorer le contenu d'un journal d'événements, il faut utiliser le paramètre `-LogName` :

```
PS> Get-EventLog -LogName 'Windows PowerShell'
```

Index	Time	EntryType	Source	InstanceID	Message
1992	juin 24 14:09	Information	PowerShell	400	L'état du
1991	juin 24 14:09	Information	PowerShell	600	Le fournis
1990	juin 24 14:09	Information	PowerShell	600	Le fournis
1989	juin 24 14:09	Information	PowerShell	600	Le fournis
1988	juin 24 14:09	Information	PowerShell	600	Le fournis
1987	juin 24 14:09	Information	PowerShell	600	Le fournis
1986	juin 24 14:09	Information	PowerShell	600	Le fournis
1985	juin 23 17:08	Information	PowerShell	403	L'état du
1984	juin 23 17:05	Information	PowerShell	400	L'état du
1983	juin 23 17:05	Information	PowerShell	600	Le fournis
1982	juin 23 17:05	Information	PowerShell	600	Le fournis
1981	juin 23 17:05	Information	PowerShell	600	Le fournis
1980	juin 23 17:05	Information	PowerShell	600	Le fournis
1979	juin 23 17:05	Information	PowerShell	600	Le fournis
1978	juin 23 17:05	Information	PowerShell	600	Le fournis
1977	juin 23 17:00	Information	PowerShell	403	L'état du
1976	juin 23 14:50	Information	PowerShell	400	L'état du
1975	juin 23 14:50	Information	PowerShell	600	Le fournis

```
1974 juin 23 14:50 Information PowerShell      600 Le fournisseur
1973 juin 23 14:50 Information PowerShell      600 Le fournisseur
...

```

Ici, la sortie a été volontairement tronquée, mais comme les listes issues de ces différents journaux sont généralement très longues, la cmdlet `Get-EventLog` dispose d'un paramètre nommé `-Newest`, qui limite l'affichage aux entrées les plus récentes :

```
PS> Get-EventLog -LogName 'Windows PowerShell' -Newest 10
```

Index	Time	EntryType	Source	InstanceID	Message
1992	juin 24 14:09	Information	PowerShell	400	L'état du moteur
1991	juin 24 14:09	Information	PowerShell	600	Le fournisseur «
1990	juin 24 14:09	Information	PowerShell	600	Le fournisseur «
1989	juin 24 14:09	Information	PowerShell	600	Le fournisseur «
1988	juin 24 14:09	Information	PowerShell	600	Le fournisseur «
1987	juin 24 14:09	Information	PowerShell	600	Le fournisseur «
1986	juin 24 14:09	Information	PowerShell	600	Le fournisseur «
1985	juin 23 17:08	Information	PowerShell	403	L'état du moteur
1984	juin 23 17:05	Information	PowerShell	400	L'état du moteur
1983	juin 23 17:05	Information	PowerShell	600	Le fournisseur «
...					

Cet exemple met en évidence les dix entrées les plus récentes. Là encore, l'affichage se fait sous forme de tableau ; la colonne `Message` est affichée de manière partielle, ce qui empêche une lecture tout à fait claire des entrées. Dans ce cas de figure, il est préférable d'utiliser une autre vue, par exemple une liste :

```
PS> Get-EventLog -LogName 'Windows PowerShell'-Newest 1 | Format-List
```

```
Index          : 1992
EntryType      : Information
InstanceId     : 400
Message        : L'état du moteur passe de None à Available.
```

```
Détails :
  NewEngineState=Available
  PreviousEngineState=None
```

```
SequenceNumber=13
```

```
HostName=ConsoleHost
HostVersion=3.0
HostId=7e89413c-7869-4463-9ad8-2f949888cf31
```

```

EngineVersion=3.0
RunspaceId=88de02d9-467e-4ac9-940a-beb509f6dc94
PipelineId=
CommandName=
 CommandType=
 ScriptName=
 CommandPath=
 CommandLine=
Category : Cycle de vie du moteur
CategoryNumber : 4
ReplacementStrings : {Available, None,      NewEngineState=Available
                     PreviousEngineState=None
SequenceNumber=13

HostName=ConsoleHost
HostVersion=3.0
HostId=7e89413c-7869-4463-9ad8-2f949888cf31
EngineVersion=3.0
RunspaceId=88de02d9-467e-4ac9-940a-beb509f6dc94
PipelineId=
CommandName=
 CommandType=
 ScriptName=
 CommandPath=
 CommandLine=}
Source : PowerShell
TimeGenerated : 24/06/2013 14:09:19
TimeWritten : 24/06/2013 14:09:19
UserName :

```

La cmdlet `Format-List` a présenté sous forme de liste toutes les propriétés des objets représentant les entrées de journaux d'événements. L'analyse des informations devient donc au fur et à mesure plus aisée.

Les possibilités qu'offre la commande `Get-EventLog` en matière de filtrage des informations sont encore plus vastes. Par exemple, essayons d'obtenir les erreurs enregistrées dans le journal des événements système :

```
PS> Get-EventLog -LogName 'System' -EntryType 'Error' -Newest 15 | Select
EntryType,Source,Message
```

EntryType	Source	Message
Error	Microsoft-Windows-WLAN-AutoConfig	Le module d'extensibilité
Error	Microsoft-Windows-WLAN-AutoConfig	Le module d'extensibilité
Error	Service Control Manager	Le service Explorateur d'
Error	BROWSER	L'explorateur n'a pas pu

Error Service Control Manager	Le service Explorateur d'
Error BROWSER	L'explorateur n'a pas pu
Error Microsoft-Windows-WLAN-AutoConfig	Le module d'extensibilité
...	

Pour spécifier uniquement les événements d'erreurs, nous avons utilisé le paramètre `-EntryType`. L'exemple suivant met en articulation `Get-EventLog` avec la cmdlet `Where-Object`, qui apporte plus de puissance et de précision en matière de filtrage :

```
PS> Get-EventLog -LogName 'Windows PowerShell' | Where-Object {$_._InstanceId -eq 600}
```

Index	Time	EntryType	Source	InstanceId	Message
1991	juin 24 14:09	Information	PowerShell	600	Le fournisseur
1990	juin 24 14:09	Information	PowerShell	600	Le fournisseur
1989	juin 24 14:09	Information	PowerShell	600	Le fournisseur
1988	juin 24 14:09	Information	PowerShell	600	Le fournisseur
1987	juin 24 14:09	Information	PowerShell	600	Le fournisseur
1986	juin 24 14:09	Information	PowerShell	600	Le fournisseur
1983	juin 23 17:05	Information	PowerShell	600	Le fournisseur
1982	juin 23 17:05	Information	PowerShell	600	Le fournisseur
1981	juin 23 17:05	Information	PowerShell	600	Le fournisseur
1980	juin 23 17:05	Information	PowerShell	600	Le fournisseur
1979	juin 23 17:05	Information	PowerShell	600	Le fournisseur
1978	juin 23 17:05	Information	PowerShell	600	Le fournisseur
1975	juin 23 14:50	Information	PowerShell	600	Le fournisseur
1974	juin 23 14:50	Information	PowerShell	600	Le fournisseur
1973	juin 23 14:50	Information	PowerShell	600	Le fournisseur
1972	juin 23 14:50	Information	PowerShell	600	Le fournisseur
1971	juin 23 14:50	Information	PowerShell	600	Le fournisseur
1970	juin 23 14:50	Information	PowerShell	600	Le fournisseur
1967	juin 23 14:23	Information	PowerShell	600	Le fournisseur
1966	juin 23 14:23	Information	PowerShell	600	Le fournisseur
1965	juin 23 14:23	Information	PowerShell	600	Le fournisseur
1964	juin 23 14:23	Information	PowerShell	600	Le fournisseur
...					

Ici, les entrées du journal d'événements 'Windows PowerShell' sont envoyées à la cmdlet `Where-Object` pour que cette dernière les filtre sur leur propriété `InstanceId`.

Le filtrage par date ou période est également fréquemment utilisé. L'exemple suivant liste les événements d'erreurs du journal '[Application](#)' qui se sont produits durant les sept derniers jours :

EntryType	Source	InstanceId	Message
Error	Application Error	1000	Nom de l'appli
Error	Perflib	3221226480	La description
Error	Perflib	3221226489	La description
Error	Perflib	3221226494	La description
Error	Perflib	3221226495	La description
Error	Perflib	3221226489	La description
Error	Perflib	3221226494	La description
Error	Application Hang	1002	Le programme E
Error	Application Error	1000	Nom de l'appli
Error	Application Hang	1002	Le programme W
Error	Microsoft-Windows-Immersive-Shell	2484	Le package win
Error	SideBySide	3238068233	La création du
Error	Application Error	1000	Nom de l'appli
Error	Application Error	1000	Nom de l'appli
Error	SideBySide	3238068233	La création du
...			

Toutes ces opérations de gestion des journaux d'événements sont tout à fait fondamentales pour un administrateur en termes d'analyse des données existantes. Elles représentent une partie non négligeable de leur quotidien. Effectuer ces opérations avec PowerShell fait sans conteste gagner du temps.

## Configurer et sauvegarder les logs

### Configurer les logs

Configurer les logs aide, par exemple, à contrôler la taille des journaux d'événements ou même combien de temps un événement pourra être enregistré. C'est la cmdlet [Limit-EventLog](#) qui se charge de la configuration. L'exemple suivant limite le journal d'événements '[Windows PowerShell](#)' à une taille maximale de 7 Mo :

```
PS> Limit-EventLog -LogName 'Windows PowerShell' -MaximumSize 7MB
```

Si le journal d'événements en question excède le maximum, alors il est possible d'agir de trois façons différentes à l'aide du paramètre `-OverflowAction`.

- `DoNotOverwrite`

PowerShell signale une erreur indiquant que le maximum a été atteint.

- `OverwriteAsNeeded`

Chaque nouvelle entrée remplace la dernière.

- `OverwriteOlder`

Les nouvelles entrées ne remplacent que celles dont la durée de rétention dépasse le nombre de jours indiqué par la propriété `-RetentionDays`.

À partir de l'exemple précédent, nous pouvons définir une taille maximale de 7 Mo et spécifier une action particulière, comme remplacer les entrées dont la durée de rétention dépasse un certain nombre de jours (par exemple, quinze jours) :

```
PS> Limit-EventLog -LogName 'Windows PowerShell' -MaximumSize 7MB  
-OverflowAction OverwriteOlder -RetentionDays 15
```

#### NOTE Concernant la commande `Limit-EventLog`

Utilisez le paramètre `-ComputerName` pour configurer les logs sur des machines distantes.

## Sauvegarder les logs

Parmi les cmdlets disponibles pour manipuler les journaux d'événements, il n'y en a toujours aucun dont le but soit de sauvegarder les logs. On peut néanmoins recourir à d'autres moyens, comme WMI :

```
PS> [wmi]$Pslog = Get-WmiObject -query "Select * from Win32_NTEventLogFile where  
LogFile=Application'" -EnableAllPrivileges
```

La cmdlet `Get-WmiObject` obtient ici une instance de la classe `[Win32_NTEventLogFile]`. De plus, un filtre est appliqué sur la propriété `LogFile` afin d'indiquer que la requête porte uniquement sur le journal d'événements qui concerne les applications.

Maintenant que nous avons établi la base, il faut définir un nom pour notre sauvegarde. Il sera composé de la date de sauvegarde, du nom du type de journal ainsi que de l'extension `.evt` :

```
PS> $BackupName = $((Get-Date -Format yyyyMMdd)+"_") + $Pslog.LogfileName +  
.evt"
```

Enfin, nous utilisons la méthode `BackupEventLog()` pour effectuer la sauvegarde :

```
PS> $PsLog.BackupEventLog("Z:\EventLogBackups\$BackupName")  
  
__GENUS : 2  
__CLASS : __PARAMETERS  
__SUPERCLASS :  
__DYNASTY : __PARAMETERS  
__RELPATH :  
__PROPERTY_COUNT : 1  
__DERIVATION : {}  
__SERVER :  
__NAMESPACE :  
__PATH :  
ReturnValue : 0
```

La valeur de retour est `0`, ce qui signifie que la sauvegarde a bien eu lieu. Évidemment, un script de sauvegarde peut être développé à partir de ces lignes de commandes.

## Supprimer les logs

PowerShell offre la possibilité d'effacer le contenu des journaux d'événements à l'aide de la cmdlet `Clear-EventLog`. Cette dernière efface toutes les entrées d'un journal spécifique, mais pas le journal lui-même :

```
PS> Clear-EventLog -LogName 'Windows PowerShell'
```

Pour effectuer cette même opération sur une machine distante, il faut utiliser le paramètre `-ComputerName` :

```
PS> Clear-EventLog -LogName 'Windows PowerShell' -ComputerName MLSRV19
```

Si nous voulons effacer le journal lui-même, une possibilité parmi d'autres est d'utiliser la commande `Remove-EventLog` :

```
PS> Remove-EventLog -LogName 'Spec_Logs'
```

### À SAVOIR La particularité des cmdlets EventLog

Les cmdlets `EventLog` fonctionnent uniquement avec les journaux des événements classiques.

# 18

## Gérer la base de registres

---

*La base de registres est au système d'exploitation Windows ce qu'est le système nerveux au corps humain. Il s'agit d'une base de données stockant la configuration du système. Elle est soumise à des modifications constantes dues à des manipulations diverses, comme l'installation de nouveaux logiciels ou même des modifications de configuration liées à ces logiciels.*

*PowerShell agit sur la base de registres de manière très originale. Nous commencerons par apprendre comment naviguer dans la base en utilisant un fournisseur spécialement dédié. Puis nous verrons comment chercher, ajouter et effacer un élément de registre à l'aide de PowerShell. Nous terminerons ce chapitre en utilisant PowerShell pour gérer la base de registres d'une machine distante.*

### SOMMAIRE

- ▶ Naviguer dans la base de registres
- ▶ Chercher, ajouter et effacer un élément de registre
- ▶ La base de registres à distance

## Naviguer dans la base de registres

Concernant certains éléments du système, PowerShell a des propositions très particulières et originales. C'est le cas pour la base de registres de Windows. À travers un fournisseur dédié, il est possible de parcourir la base comme nous le ferions pour un système de fichiers classique. Tout d'abord, commençons par obtenir la liste des fournisseurs disponibles par défaut :

```
PS> Get-PSPProvider
```

Name	Capabilities	Drives
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess, Credentials	{C, A, D}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transactions	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}

La sortie nous montre qu'il y a bien un fournisseur nommé `Registry`. A priori et sur la base de cette information, il y a effectivement des lecteurs permettant de naviguer dans la base de registres. Mais lesquels ? La cmdlet `Get-PSDrive` donne la réponse à cette question :

```
PS> Get-PSDrive
```

Name	Used (GB)	Free (GB)	Provider	Root
A			FileSystem	A:\
Alias			Alias	
C	12,21	27,79	FileSystem	C:\
Cert			Certificate	\
D	,01		FileSystem	D:\
Env			Environment	
Function			Function	
HKCU			Registry	HKEY_CURRENT_USER
HKLM			Registry	HKEY_LOCAL_MACHINE
Variable			Variable	
WSMan			WSMan	

Le résultat est un peu trop verbeux. Essayons de filtrer notre requête :

```
PS> Get-PSDrive -PSProvider Registry
```

Name	Used (GB)	Free (GB)	Provider	Root
---	-----	-----	-----	---
HKCU			Registry	HKEY_CURRENT_USER
HKLM			Registry	HKEY_LOCAL_MACHINE

Il y a exactement deux lecteurs capables de naviguer dans la base de registres. Maintenant, pour naviguer par exemple dans le lecteur HKCU: (HKEY\_CURRENT\_USER) comme dans un système de fichiers classique, nous utilisons la commande `Set-Location` ou son alias `cd` :

```
PS C:\Users\Administrator> cd hkcu:  
PS HKCU:>
```

Le prompt a changé ; nous ne sommes plus dans le système de fichiers C:\ mais dans la base de registres :

```
PS HKCU:> cd .\Software  
PS HKCU:\Software> ls  
  
Hive: HKEY_CURRENT_USER\Software  
  
Name                                 Property  
----  
AppDataLow                        ---  
Microsoft                        ---  
Mine                              (default) : {}  
Policies                        ---  
ThinPrint                        ---  
VMware, Inc.                   ---  
Wow6432Node                    ---  
Classes                          ---
```

Les valeurs des clés peuvent être énumérées à l'aide de la cmdlet `Get-ItemProperty` :

```
PS HKCU:\Software> cd .\Microsoft\Windows\CurrentVersion\Explorer  
PS HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer> Get-ItemProperty .  
  
ExplorerStartupTraceRecorded : 1  
ShellState                    : {36, 0, 0, 0...}  
SIDUpdatedOnLibraries      : 1  
LastClockSize               : {39, 0, 0, 0...}  
PSPath                       : Microsoft.PowerShell.Core\Registry::HKEY_CUR...  
PSParentPath               : Microsoft.PowerShell.Core\Registry::HKEY_CUR...  
PSChildName               : Explorer
```

PSDrive	:	HKCU
PSProvider	:	Microsoft.PowerShell.Core\Registry

Les sous-clés sont énumérées quant à elles via la cmdlet `Get-ChildItem` :

```
PS HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer> Get-ChildItem |  
ForEach-Object { ($_. -split "\\")[-1] }
```

```
Advanced  
AutoplayHandlers  
BitBucket  
CabinetState  
CD Burning  
CLSID  
ControlPanel  
Discardable  
FileExts  
LowRegistry  
MenuOrder  
Modules  
MountPoints2  
NewShortcutHandlers  
Package Installation Version  
RecentDocs  
RestartCommands  
Ribbon  
RunMRU  
SearchPlatform  
Shell Folders  
Shutdown  
StartPage  
Streams  
StuckRects2  
Taskband  
TypedPaths  
User Shell Folders  
UserAssist  
VisualEffects  
Wallpapers  
SessionInfo
```

Si nous sommes intéressés uniquement par une valeur de clé de registre, alors le paramètre `-Name` de la commande `Get-ItemProperty` peut être invoqué :

```
PS HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer> Gp -Name  
ShellState -Path .  
  
ShellState : {36, 0, 0, 0...}
```

```
PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_
PSChildName  : Explorer
PSDrive     : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
```

On constate que naviguer dans la base de registres en utilisant PowerShell est relativement simple et ne pose pas de problèmes particuliers. Toutefois, simplicité d'utilisation ne veut pas dire aisance dans la pratique, car celle-ci requiert du temps, voire une certaine patience.

## Chercher, ajouter et effacer un élément de registre

La base de registres étant une véritable base de données accessible en lecture et en écriture, il est évidemment possible de la lire, de l'analyser, voire de la modifier.

### Chercher des informations dans la base de registres

Par exemple, essayons de rechercher des occurrences de clés dont le nom inclut le terme « Windows » :

```
PS HKCU:\Software\Microsoft> Get-ChildItem -Path . -Include *Windows* -Recurse
Hive: HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer
```

Name	Property
Document Windows	Maximized : no height : {0, 0, 0, 0} width : {0, 0, 0, 128} x : {0, 0, 0, 128} y : {0, 0, 0, 0}

```
Hive: HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main
```

Name	Property
WindowsSearch	Version : WS not installed

```
Hive: HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer
```

```

Name          Property
----          -----
New Windows   PopupMgr : yes
               PlaySound : 1
               UseSecBand : 1

Hive: HKEY_CURRENT_USER\Software\Microsoft

```

```

Name          Property
----          -----
Windows
(...)
```

Dans cet exemple, nous avons démarré la recherche à partir du chemin `HKCU:\Software\Microsoft`. Puis le paramètre `-Include` a spécifié des occurrences comportant le nom « Windows ». Enfin, le paramètre `-Recurse` indique à PowerShell que la recherche doit se faire aussi dans les répertoires enfants. Pour constituer une liste de clés, nous ne retiendrons que la propriété `PSChildName` :

```
PS HKCU:\Software\Microsoft> Get-ChildItem -Path . -Include *Windows* -Recurse | Select PSChildName
```

```

PSChildName
-----
Document Windows
WindowsSearch
New Windows
Windows
DragFullWindows
Windows Error Reporting
Windows NT
Windows
```

Le nom de la colonne peut paraître quelque peu mystérieux. Modifions-le pour le rendre plus compréhensible :

```
PS HKCU:\Software\Microsoft> Get-ChildItem -Path . -Include *Windows* -Recurse | Select @{Name="Clés de registres";Expression={$_.PSChildName}}
```

```

Clés de registres
-----
Document Windows
WindowsSearch
New Windows
Windows
```

```
DragFullWindows
Windows Error Reporting
Windows NT
Windows
```

## Ajouter un élément dans la base de registres

Ajouter des éléments dans la base de registres est le plus souvent une opération réalisée par des programmes, et dans une certaine transparence. Le fait est qu'un utilisateur ne devra modifier la base de registres que dans des cas exceptionnels, c'est-à-dire dans des contextes particuliers où telle opération nécessite par exemple de créer telle clé. Si cela doit vous arriver, sachez que PowerShell vous facilite l'opération avec la cmdlet `New-Item`. La ligne de commande suivante crée une clé nommée `Booster Engine` dans le chemin `HKCU:\Software` :

```
PS HKCU:\Software> New-Item -Path . -Name 'Booster Engine'

Hive: HKEY_CURRENT_USER\Software

Name          Property
----          -----
Booster Engine
```

Maintenant que la clé est créée, nous pouvons lui attribuer des propriétés via la cmdlet `New-ItemProperty` :

```
PS HKCU:\Software> cd '.\Booster Engine'
PS HKCU:\Software\Booster Engine> New-ItemProperty -Path . -Name 'GrammarLevel'
-Value 3

GrammarLevel : 3
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT..
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT..
PSChildName  : Booster Engine
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry

PS HKCU:\Software\Booster Engine>New-ItemProperty -Path . -Name 'ParserLevel'
-Value 2

ParserLevel  : 2
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT..
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT..
PSChildName  : Booster Engine
PSDrive      : HKCU
```

```
PSProvider    : Microsoft.PowerShell.Core\Registry  
PS HKCU:\Software\Booster Engine> New-ItemProperty -Path . -Name  
'IntrinsicLevel' -Value 1  
  
IntrinsicLevel : 1  
PSPath         : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT..  
PSParentPath   : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT..  
PSChildName   : Booster Engine  
PSDrive        : HKCU  
PSProvider     : Microsoft.PowerShell.Core\Registry
```

La sortie indique que les propriétés ont bien été créées. Ces dernières peuvent être listées de la manière suivante :

```
PS HKCU:\Software\Booster Engine> Get-ItemProperty -Path .  
  
GrammarLevel   : 3  
ParserLevel    : 2  
IntrinsicLevel : 1  
PSPath         : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_  
PSParentPath   : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_  
PSChildName   : Booster Engine  
PSDrive        : HKCU  
PSProvider     : Microsoft.PowerShell.Core\Registry
```

## Effacer un élément de la base de registres

Les informations que nous avons créées dans la base de registres sont évidemment effaçables ou même modifiables. Parmi les trois valeurs que nous avons créées dans la section précédente, supprimons de la clé `Booster Engine` la propriété `IntrinsicLevel` :

```
PS HKCU:\Software\Booster Engine> Remove-ItemProperty -Path . -Name  
'IntrinsicLevel'  
PS HKCU:\Software\Booster Engine>
```

La sortie est vide ; cela prouve que l'opération s'est bien déroulée :

```
PS HKCU:\Software\Booster Engine> Get-ItemProperty -Path .  
  
GrammarLevel   : 3  
ParserLevel    : 2  
PSPath         : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_  
PSParentPath   : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_
```

```
PSChildName      : Booster Engine
PSDrive          : HKCU
PSProvider        : Microsoft.PowerShell.Core\Registry
```

Si une valeur créée doit être modifiée et non pas effacée, la cmdlet `Set-ItemProperty` devra être invoquée :

```
PS HKCU:\Software\Booster Engine> Set-ItemProperty -Path . -Name 'ParserLevel' -Value 5
PS HKCU:\Software\Booster Engine> Get-ItemProperty -Path .
GrammarLevel     : 3
ParserLevel      : 5
PSPath           : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_
PSParentPath     : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_
PSChildName      : Booster Engine
PSDrive          : HKCU
PSProvider        : Microsoft.PowerShell.Core\Registry
```

## La base de registres à distance

Depuis le début du chapitre, nous avons administré la base de registres sur une machine locale. Pour une gestion à distance, plusieurs possibilités s'offrent à nous. La plus simple et la plus claire est de passer par la communication à distance PowerShell. Cependant, comme nous n'avons pas encore étudié cet aspect de PowerShell, nous opterons pour une autre option, qui est celle basée sur le framework .NET.

Dans cette section, nous allons, à partir d'un client donné, modifier la valeur d'une clé de registre sur une machine distante. Dans la section précédente, nous avons créé une clé de registre ainsi que des valeurs qui lui sont associées. L'opération va donc consister à modifier de manière distante une des valeurs associée à cette clé de registre.

Commençons par une requête distante consistant à se connecter à une instance de la clé de registre `HKCU` :

```
PS> $rootkey=[Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey("CurrentUser", "DC2K12ML")
```

La classe utilisée se nomme `[RegistryKey]`. Via cette classe, nous avons appelé la méthode `OpenRemoteBaseKey()` afin d'encapsuler dans une variable la clé `HKCU`. Cette dernière prend deux paramètres.

- Le premier paramètre consiste en la spécification d'une clé racine.
- Le second paramètre spécifie quant à lui le nom de la machine distante.

À présent, essayons d'interroger les membres de cette clé :

```
PS> $rootkey | gm
```

TypeName: Microsoft.Win32.RegistryKey

Name	MemberType	Definition
Close	Method	void Close()
CreateObjRef	Method	System.Runtime.Remoting.ObjR
CreateSubKey	Method	Microsoft.Win32.RegistryKey
DeleteSubKey	Method	void DeleteSubKey(string sub
DeleteSubKeyTree	Method	void DeleteSubKeyTree(string
DeleteValue	Method	void DeleteValue(string name
Dispose	Method	void Dispose(), void IDispos
Equals	Method	bool Equals(System.Object ob
Flush	Method	void Flush()
GetAccessControl	Method	System.Security.AccessContro
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeSer
GetSubKeyNames	Method	string[] GetSubKeyNames()
GetType	Method	type GetType()
GetValue	Method	System.Object GetValue(strin
GetValueKind	Method	Microsoft.Win32.RegistryValu
GetValueNames	Method	string[] GetValueNames()
InitializeLifetimeService	Method	System.Object InitializeLife
<b>OpenSubKey</b>	Method	Microsoft.Win32.RegistryKey
SetAccessControl	Method	void SetAccessControl(System
SetValue	Method	void SetValue(string name, S
ToString	Method	string ToString()
Handle	Property	Microsoft.Win32.SafeHandles.
Name	Property	string Name {get;}
SubKeyCount	Property	int SubKeyCount {get;}
ValueCount	Property	int ValueCount {get;}
View	Property	Microsoft.Win32.RegistryView

En lisant attentivement la sortie, nous remarquons la présence de la méthode `OpenSubKey()`. Utilisons donc cette méthode pour instancier la sous-clé de registre `Booster Engine` :

```
PS> $Booster = $rootkey.OpenSubKey("Software\Booster Engine",$True)
```

Voici la liste des propriétés de cette clé de registre :

```
PS> $Booster.GetValueNames()
```

```
GrammarLevel
```

```
ParserLevel
```

Voici les valeurs correspondantes :

```
PS> $Booster.GetValue("GrammarLevel")
3
PS> $Booster.GetValue("ParserLevel")
5
```

La valeur de la propriété `ParserLevel` est, au moment de la sortie, de `5`. Modifions-la pour qu'elle passe de `5` à `7` :

```
PS> $Booster.SetValue("ParserLevel",7)
```

Comme nous pouvons le voir, la méthode adéquate pour modifier la valeur qui nous intéresse est `SetValue()`. Dans ce contexte, il faut absolument vérifier que la modification a eu lieu :

```
PS> $Booster.GetValue("ParserLevel")
7
```

Il ne nous reste plus qu'à libérer les ressources mobilisées par cette opération, ce qui implique que les clés seront à nouveau utilisables par d'autres programmes :

```
PS> $Booster.Dispose()
PS> $Rootkey.Dispose()
```

Tout au long de ce chapitre, nous avons pu constater à quel point travailler avec la base de registres demandait de la prudence. Une bonne pratique est d'effectuer les manipulations liées à la base de registres dans un environnement de tests avant de passer à un environnement de production.



# 19

## PowerShell et COM

---

*COM (Component Object Model) est le nom d'une technologie que les anciens connaissent bien. Il s'agit en fait d'une technologie antérieure au framework .NET permettant notamment de gérer des ressources liées au système d'exploitation ou aux applications tierces. Des langages comme VBScript ou Jscript concentraient toute leur puissance sur ce matériau de programmation et cette voie d'accès aux ressources était le principal chemin à suivre en matière de scripting. WMI (Windows Management Instrumentation), ADSI (Active Directory Services Interfaces) ou même des produits liés à la suite Office sont encore aujourd'hui gérables via la technologie COM, qui reste encore d'actualité pour bien des années.*

*Dans ce chapitre, nous nous familiariserons avec les objets COM afin de les situer dans la perspective de PowerShell. Ensuite, nous apprendrons à utiliser une interface COM à travers plusieurs exemples comme l'automatisation d'Internet Explorer, le mappage d'un lecteur réseau et la manipulation du système de fichiers.*

### SOMMAIRE

- ▶ Manipuler un objet COM
- ▶ Automatiser Internet Explorer
- ▶ Mapper un lecteur réseau
- ▶ Utiliser l'objet FileSystem

## Manipuler un objet COM

Le premier élément d'information à connaître, qui est aussi un élément fondamental pour la suite du code, est ce que l'on appelle le [ProgID](#). Il s'agit de l'identifiant d'un objet. C'est via cet identifiant unique que l'objet COM sera instancié, avec la cmdlet [New-Object](#) :

```
PS> $ComObject = New-Object -ComObject 'WScript.Shell'
```

Le paramètre [-ComObject](#) spécifie l'identifiant de l'objet COM. Dans l'exemple, l'objet en question est `Wscript.Shell` ; il est très utile pour exécuter des programmes, manipuler la base de registres ou accéder au système de fichiers. Listons ses membres afin de mieux le connaître :

```
PS> $ComObject | Get-Member
```

```
TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd35090}
```

Name	MemberType	Definition
pstypenames	CodeProperty	System.Collections.
psadapted	MemberSet	psadapted {SpecialF
psbase	MemberSet	psbase {GetLifetime
psextended	MemberSet	psextended {}
psobject	MemberSet	psobject {BaseObjec
AppActivate	Method	bool AppActivate (V
CreateShortcut	Method	IDispatch CreateSho
Exec	Method	IWshExec Exec (stri
ExpandEnvironmentStrings	Method	string ExpandEnviro
LogEvent	Method	bool LogEvent (Vari
Popup	Method	int Popup (string,
RegDelete	Method	void RegDelete (str
RegRead	Method	Variant RegRead (st
RegWrite	Method	void RegWrite (stri
Run	Method	int Run (string, Va
SendKeys	Method	void SendKeys (stri
Environment	ParameterizedProperty	IWshEnvironment Env
CurrentDirectory	Property	string CurrentDirec
SpecialFolders	Property	IWshCollection Spec

Comme nous apprenons à manipuler un objet COM via PowerShell, testons (toujours dans un environnement de test) certaines propriétés et méthodes. Par exemple, la propriété `CurrentDirectory` indique le répertoire actuel du processus actif :

```
PS> $ComObject.CurrentDirectory  
C:\Users\Administrator
```

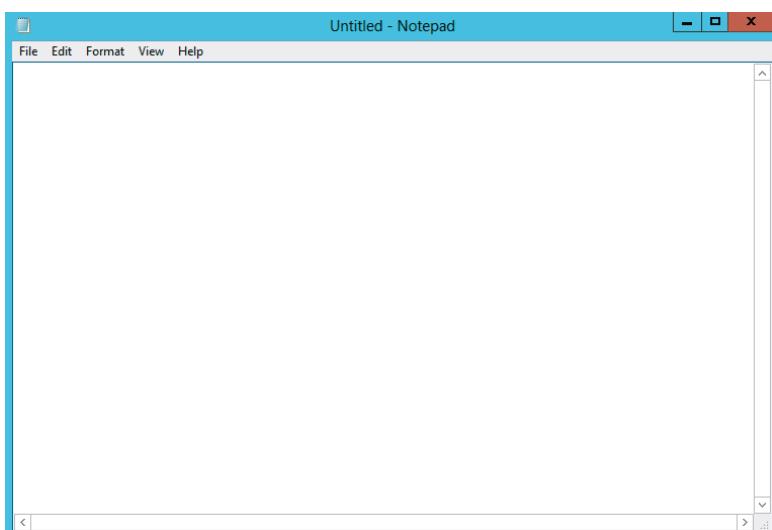
La méthode `Exec()` permet quant à elle de lancer un programme (Notepad en l'occurrence) :

```
PS> $ComObject.Exec("notepad")
```

```
Status      : 0
StdIn       : System.__ComObject
StdOut      : System.__ComObject
StdErr      : System.__ComObject
ProcessID   : 2616
ExitCode    : 0
```

**Figure 19–1**

Programme Notepad lancé à l'aide de l'objet COM (`Wscript.Shell`) instancié avec PowerShell



L'exécution de cette méthode affiche une sortie indiquant une valeur de retour de `0`. La ligne de commande a donc été exécutée avec succès et le programme Notepad est affiché à l'écran comme attendu.

La manipulation d'un objet COM est finalement similaire à celle d'un objet .Net, même si les mécanismes respectifs de ces technologies diffèrent dans leur essence. La documentation est bien plus fournie pour .NET que pour COM ; la conséquence de cela est que maîtriser un objet COM que l'on ne connaît pas a priori relève parfois d'une certaine difficulté, proportionnelle à la complexité de l'objet COM lui-même. Passons à l'étude d'exemples concrets pour mieux prendre en compte la réalité de la technologie COM.

## Automatiser Internet Explorer

Dans cette section, nous allons apprendre à automatiser Internet Explorer à travers deux exemples. Le premier exemple consistera à instancier le programme Internet Explorer en lui indiquant une URL. Le second exemple consistera quant à lui en une connexion automatique au site Gmail.

### Accéder automatiquement à une page web

Accéder automatiquement à une page web est très simple, et ce, quel que soit le langage utilisé. L'objet que nous allons utiliser se nomme `InternetExplorer.Application`. Voici sa structure :

```
PS> $ie = new-object -com "InternetExplorer.Application"
PS> $ie | gm

TypeName: System.__ComObject#{d30c1661-cdaf-11d0-8a3e-00c04fc9e26e}

Name          MemberType  Definition
----          -----
ClientToWindow Method     void ClientToWindow (int, int)
ExecWB        Method     void ExecWB (OLECMDID, OLECMDEXECOPT)
GetProperty    Method     Variant GetProperty (string)
GoBack        Method     void GoBack ()
GoForward     Method     void GoForward ()
GoHome        Method     void GoHome ()
GoSearch       Method     void GoSearch ()
Navigate    Method     void Navigate (string, Variant, Vari
Navigate2      Method     void Navigate2 (Variant, Variant, Va
PutProperty    Method     void PutProperty (string, Variant)
QueryStatusWB Method     OLECMDF QueryStatusWB (OLECMDID)
Quit          Method     void Quit ()
Refresh        Method     void Refresh ()
Refresh2       Method     void Refresh2 (Variant)
ShowBrowserBar Method     void ShowBrowserBar (Variant, Varian
Stop           Method     void Stop ()
AddressBar     Property   bool AddressBar () {get} {set}
Application    Property   IDispatch Application () {get}
Busy           Property   bool Busy () {get}
Container      Property   IDispatch Container () {get}
Document       Property   IDispatch Document () {get}
FullName       Property   string FullName () {get}
FullScreen     Property   bool FullScreen () {get} {set}
Height          Property  int Height () {get} {set}
HWND            Property  int HWND () {get}
Left             Property  int Left () {get} {set}
LocationName   Property  string LocationName () {get}
```

LocationURL	Property	string LocationURL () {get}
MenuBar	Property	bool MenuBar () {get} {set}
Name	Property	string Name () {get}
Offline	Property	bool Offline () {get} {set}
Parent	Property	IDispatch Parent () {get}
Path	Property	string Path () {get}
ReadyState	Property	tagREADYSTATE ReadyState () {get}
RegisterAsBrowser	Property	bool RegisterAsBrowser () {get} {set}
RegisterAsDropTarget	Property	bool RegisterAsDropTarget () {get} {set}
Resizable	Property	bool Resizable () {get} {set}
Silent	Property	bool Silent () {get} {set}
StatusBar	Property	bool StatusBar () {get} {set}
StatusText	Property	string StatusText () {get} {set}
TheaterMode	Property	bool TheaterMode () {get} {set}
ToolBar	Property	intToolBar () {get} {set}
(...)		

Le nombre de membres ici est important, mais une méthode retient l'attention, `Navigate()`. En se documentant un peu, on découvre que cette méthode donne effectivement un accès automatique à une page web :

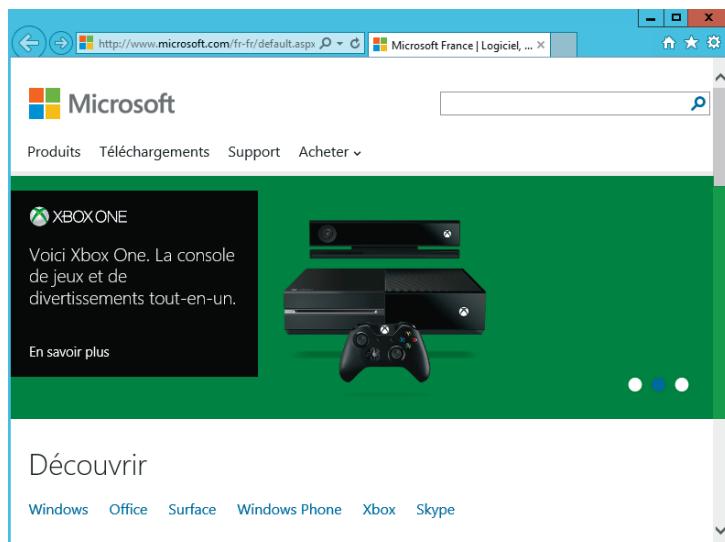
```
PS> $ie.navigate("www.microsoft.com")
```

L'appel de cette méthode ne produit aucune sortie. Ce comportement est en réalité tout à fait normal. Le programme Internet Explorer est instancié, mais non visible. Pour le rendre visible, il faut attribuer la valeur `True` à l'objet créé :

```
PS> $ie.Visible = $true
```

**Figure 19–2**

Instance du navigateur Internet Explorer lancé via PowerShell



Une page web s'est affichée automatiquement à l'adresse que nous avons spécifiée ([www.microsoft.com](http://www.microsoft.com)). Ce petit exercice est un bref aperçu de ce qu'il est possible de réaliser avec l'objet COM `InternetExplorer.Application`.

## S'authentifier automatiquement sur un site web

Il n'est pas rare dans certaines entreprises de pouvoir accéder à des ressources internes accessibles, par exemple, via un intranet. En général, cela passe par une authentification plus ou moins sécurisée, qui s'articule souvent autour d'un compte et d'un mot de passe en lien avec le compte. Si un utilisateur doit accéder à plusieurs ressources dans une même journée et si ces dernières nécessitent une authentification, alors l'utilisateur perdra à long terme un temps considérable, qu'il aurait pu utiliser pour accomplir d'autres tâches bien plus intéressantes. Automatiser l'authentification que requièrent certaines opérations devient donc par extension un tremplin vers l'automatisation de ces opérations.

Pour illustrer notre propos, nous prendrons comme exemple l'authentification sur le site Gmail qui sert à consulter ses messages électroniques. L'objet que nous utiliserons pour cette démonstration est le même que précédemment. Commençons par définir des informations dont l'objet `InternetExplorer.Application` aura besoin, comme l'URL, le nom d'utilisateur, ainsi que le mot de passe :

```
PS> $url = "http://gmail.com"
PS> $username = "effective.ps@gmail.com"
PS> $password = "dfg_kx12"
```

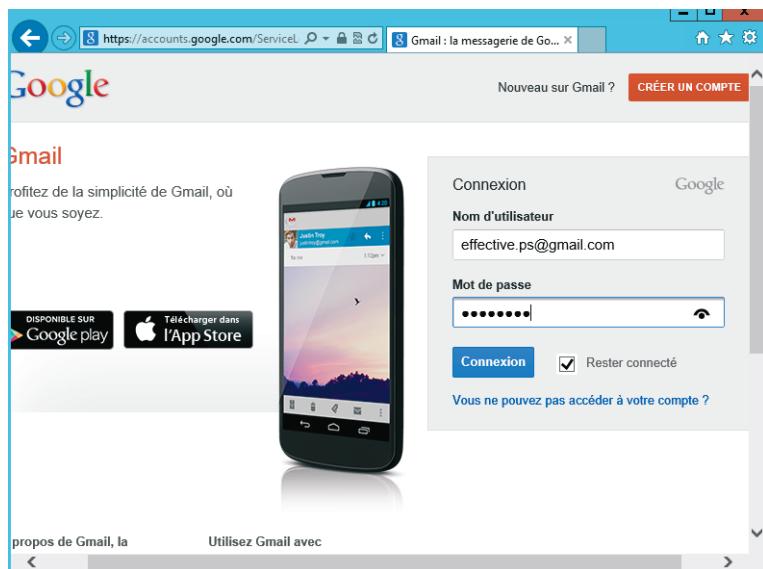
Ces informations étant fixées, passons-les à notre fameux objet :

```
PS> $ie = new-object -com internetexplorer.application
PS> $ie.navigate($url)
PS> $ie.visible = $true
PS> $ie.Document.getElementById("email").value = $username
PS> $ie.Document.getElementById("Passwd").value = $password
```

Pour des raisons pédagogiques, ces étapes sont effectuées en utilisant la console, mais pour véritablement produire un effet d'automatisation, il vaut mieux les écrire dans un script et faire en sorte que le mot de passe n'y soit pas stocké physiquement.

**Figure 19–3**

Connexion automatique au site Gmail



La page a été chargée automatiquement, les informations d’authentification y ont été correctement indiquées, mais la connexion ne s’est pas faite automatiquement, car il manque une ligne de commande pour créer cet événement :

```
PS> $ie.Document.getElementById("signin").Click()
```

Après avoir appuyé sur la touche *Entrée*, nous pouvons observer qu’une nouvelle page s’affiche, qui permet normalement d’accéder à ses messages électroniques. L’exercice s’est donc terminé avec succès.

Automatiser Internet Explorer demanderait un chapitre à lui tout seul tellement les possibilités offertes sont grandes. Les deux cas que nous avons étudiés illustrent cependant les articulations possibles en matière d’automatisation avec PowerShell.

## Mapper un lecteur réseau

Pour diverses raisons, il est fréquent qu’un administrateur ait à mapper un lecteur réseau pour accéder à certaines ressources. Il y a évidemment plusieurs façons de le faire, mais dans le contexte induit par ce chapitre, nous nous intéresserons uniquement à la méthode en lien avec la technologie COM. L’objet qui nous intéresse ici est `Wscript.Network` :

```
PS> $netmap = new-object -comobject "Wscript.Network"
PS> $netmap | gm

TypeName: System.__ComObject#{24be5a31-edfe-11d2-b933-00104b365c9f}

Name           MemberType      Definition
----           -----          -----
pstypenames   CodeProperty   System.Collections.ObjectModel
psadapted     MemberSet       psadapted {UserDomain, UserNa
psbase         MemberSet       psbase {GetLifetimeService, I
psextended    MemberSet       psextended {}
psobject       MemberSet       psobject {BaseObject, Members
AddPrinterConnection Method        void AddPrinterConnection (st
AddWindowsPrinterConnection Method   void AddWindowsPrinterConnect
EnumNetworkDrives    Method    IWshCollection EnumNetworkDri
EnumPrinterConnections Method   IWshCollection EnumPrinterCon
MapNetworkDrive  Method        void MapNetworkDrive (string,
RemoveNetworkDrive  Method    void RemoveNetworkDrive (stri
RemovePrinterConnection Method   void RemovePrinterConnection
SetDefaultPrinter   Method    void SetDefaultPrinter (strin
ComputerName     Property     string ComputerName () {get}
Organization     Property     string Organization () {get}
Site            Property     string Site () {get}
UserDomain       Property     string UserDomain () {get}
UserName         Property     string UserName () {get}
UserProfile      Property     string UserProfile () {get}
```

Pour mapper un lecteur réseau, la méthode qui doit être utilisée est `MapNetworkDrive()` :

```
PS> $netmap.MapNetworkDrive('Z:', '\DC2K12ML\Share')
```

En affichant l'explorateur Windows, on s'aperçoit qu'un nouveau lecteur a fait son apparition. La ligne de commande suivante permet de vérifier simplement que le lecteur Z: a bien été créé :

```
PS> Get-WmiObject -Class Win32_LogicalDisk
```

```
DeviceID      : A:
DriveType     : 2
ProviderName  :
FreeSpace     :
Size          :
VolumeName    :

DeviceID      : C:
DriveType     : 3
```

```
ProviderName :  
FreeSpace    : 29478100992  
Size         : 42946523136  
VolumeName   :  
  
DeviceID     : D:  
DriveType    : 5  
ProviderName :  
FreeSpace    : 0  
Size         : 7448576  
VolumeName   : WD Unlocker  
  
DeviceID     : Z:  
DriveType    : 4  
ProviderName : \\DC2K12ML\Share  
FreeSpace    : 29478100992  
Size         : 42946523136  
VolumeName   :  

```

Quand elles ne nous sont plus utiles, il est possible de libérer les ressources utilisées de cette manière :

```
PS> $netmap.RemoveNetworkDrive('Z:')
```

**NOTE Concernant le mappage de lecteurs réseau**

Mapper un lecteur réseau de manière statique est maintenant possible depuis l'arrivée de PowerShell version 3. La cmdlet à utiliser est `New-PSDrive` avec son paramètre `-Persist`.

## Utiliser l'objet FileSystem

Nous terminons ce chapitre par l'étude de l'objet COM `FileSystem`, qui est sans doute l'objet COM le plus connu et le plus utilisé en ce sens qu'il donne accès au système de fichiers :

```
PS> $Fso = New-Object -ComObject "Scripting.FileSystemObject"  
PS> $Fso | gm
```

```
TypeName: System.__ComObject#{2a0b9d10-4b87-11d3-a97a-00104b365c9f}
```

Name	MemberType	Definition
BuildPath	Method	string BuildPath (string, string)
CopyFile	Method	void CopyFile (string, string, bool)

CopyFolder	Method	void CopyFolder (string, string, bool)
CreateFolder	Method	IFolder CreateFolder (string)
CreateTextFile	Method	ITextStream CreateTextFile (string, bool)
DeleteFile	Method	void DeleteFile (string, bool)
DeleteFolder	Method	void DeleteFolder (string, bool)
DriveExists	Method	bool DriveExists (string)
FileExists	Method	bool FileExists (string)
FolderExists	Method	bool FolderExists (string)
GetAbsolutePathName	Method	string GetAbsolutePathName (string)
GetBaseName	Method	string GetBaseName (string)
GetDrive	Method	IDrive GetDrive (string)
GetDriveName	Method	string GetDriveName (string)
GetExtensionName	Method	string GetExtensionName (string)
GetFile	Method	IFile GetFile (string)
GetFileName	Method	string GetFileName (string)
GetFileVersion	Method	string GetFileVersion (string)
GetFolder	Method	IFolder GetFolder (string)
GetParentFolderName	Method	string GetParentFolderName (string)
GetSpecialFolder	Method	IFolder GetSpecialFolder (SpecialFolder)
GetStandardStream	Method	ITextStream GetStandardStream (Standard
GetTempName	Method	string GetTempName ()
MoveFile	Method	void MoveFile (string, string)
MoveFolder	Method	void MoveFolder (string, string)
OpenTextFile	Method	ITextStream OpenTextFile (string, IOMod
Drives	Property	IDriveCollection Drives () {get}

Les membres de cet objet sont surtout des méthodes, qui réalisent des opérations essentielles comme créer un fichier, ouvrir un fichier ou même obtenir des informations concernant les disques.

## Créer un fichier texte

Pour créer un fichier texte, il faut utiliser la méthode `CreateTextFile()` :

```
PS> $Fso = New-Object -ComObject "Scripting.FileSystemObject"
PS> $Wfso = $Fso.CreateTextFile("devesp.txt")
```

Ici, le fichier `devesp.txt` a été créé. Nous pouvons le vérifier avec la commande `Get-Item` :

```
PS> Get-Item .\devesp.txt

Directory: C:\Users\Administrator

Mode          LastWriteTime    Length Name
----          -----        ---- 
-a-- 02/07/2013      08:19          0 devesp.txt
```

Le fichier est donc créé, mais il ne contient encore rien. Essayons d'y écrire quelque chose :

```
PS> $Wfso.WriteLine("This is PowerShell")
```

## Ouvrir et lire un fichier texte

Avant de lire le contenu d'un fichier, il faut d'abord ouvrir ce dernier :

```
PS> $GetFile = $Fso.OpenTextFile("devesp.txt")
```

Ensuite, il peut être lu :

```
PS> $GetFile.ReadAll()
This is PowerShell
```

Lorsqu'il n'y a plus besoin de l'utiliser, il est recommandé de libérer l'espace mémoire contenant le fichier :

```
PS> $GetFile.Close()
```

Il faut reconnaître qu'avec toutes ces étapes, manipuler un fichier en passant par COM est plus contraignant qu'en passant par PowerShell et sa cmdlet [Get-Content](#). Toutefois, se pencher sur l'approche que nous venons d'étudier est utile, car de nombreux scripts l'utilisent encore.



# 20

## PowerShell et XML

---

*XML est sans doute un des langages les plus utilisés aujourd'hui. Cela s'explique par le caractère universel du langage dans sa structure. XML devient de plus en plus populaire, car il peut être sollicité dans de très nombreux contextes comme la constitution de bases de données, de fichiers de configuration ou même de journaux d'événements. PowerShell lui-même s'articule aussi autour du matériau XML, que ce soit dans la structure des types ou dans les fichiers liés au format des données.*

*La structure d'un document XML est riche et complexe, et son analyse requiert une exigence en termes de programmation. Heureusement, PowerShell offre des perspectives en la matière très intuitives, voire très puissantes. Nous commencerons donc dans ce chapitre par apprendre à manipuler un fichier XML avec PowerShell. Puis nous nous intéresserons aux cmdlets `Export-Clixml` et `Import-Clixml` et mettrons en évidence leur complémentarité. Enfin, nous verrons comment la cmdlet `Select-Xml` peut nous aider dans l'analyse et la recherche d'informations dans un fichier XML.*

### SOMMAIRE

- ▶ Manipuler un fichier XML
- ▶ Les cmdlets Export-Clixml et Import-Clixml
- ▶ Rechercher des informations à l'aide de la cmdlet Select-Xml

## Manipuler un fichier XML

Manipuler un fichier XML à l'aide de PowerShell n'échappe pas au paradigme objet. En effet, un document XML n'est rien d'autre qu'un objet à la structure complexe. Ceci implique donc que ses éléments sont analogues aux propriétés d'un objet. Pour illustrer ces informations, considérons le fichier XML suivant.

### Fichier de configuration au format XML

```
<?xml version="1.0"?>
<Parameters>
<ServerList>
    <Server>NTDEVFL</Server>
    <Server>NTDEVZT</Server>
</ServerList>
<SiteName>WebPsx</SiteName>
<InstallLocation>C:\WebIIS</InstallLocation>
<HostHeader>MLX-SDK</HostHeader>
<Port>80</Port>
<LogPath>C:\LogDirectory</LogPath>
<LogLink>C:\Directlink</LogLink>
<WebAppPath>C:\WebIIS\wpp</WebAppPath>
<ApplicationPool>DefaultAppPool</ApplicationPool>
<AppName>DevAppSoccer</AppName>
</Parameters>
```

Cet exemple, que nous nommerons `Config_Parameters.xml`, est un fichier de configuration aidant au déploiement de sites web. Il est constitué de plusieurs éléments qui représentent chacun un élément de configuration. Pour le manipuler en tant qu'objet XML, il faut opérer un transtypage :

```
PS> [xml]$Configfile = Get-Content .\Config_Parameters.xml
```

Cette opération consiste à traduire le document XML en objet XML.

```
PS> $Configfile | gm
```

TypeName: System.Xml.XmlDocument

Name	MemberType	Definition
----	-----	-----
ToString	CodeMethod	static string XmlNode
AppendChild	Method	System.Xml.XmlNode
Clone	Method	System.Xml.XmlNode
CloneNode	Method	System.Xml.XmlNode
CreateAttribute	Method	System.Xml.XmlAttr

CreateCDataSection	Method	System.Xml.XmlCData
CreateComment	Method	System.Xml.XmlComme
CreateDocumentFragment	Method	System.Xml.XmlDocum
CreateDocumentType	Method	System.Xml.XmlDocum
CreateElement	Method	System.Xml.XmlEleme
CreateEntityReference	Method	System.Xml.XmlEntit
CreateNavigator	Method	System.Xml.XPath.XP
CreateNode	Method	System.Xml.XmlNode
CreateProcessingInstruction	Method	System.Xml.XmlProce
CreateSignificantWhitespace	Method	System.Xml.XmlSigni
CreateTextNode	Method	System.Xml.XmlText
CreateWhitespace	Method	System.Xml.XmlWhite
CreateXmlDeclaration	Method	System.Xml.XmlDecla
Equals	Method	bool Equals(System.
GetElementById	Method	System.Xml.XmlEleme
GetElementsByTagName	Method	System.Xml.XmlNodeL
GetEnumerator	Method	System.Collections.
GetHashCode	Method	int GetHashCode()
GetNamespaceOfPrefix	Method	string GetNamespace
GetPrefixOfNamespace	Method	string GetPrefixOfN
GetType	Method	type GetType()
ImportNode	Method	System.Xml.XmlNode
InsertAfter	Method	System.Xml.XmlNode
InsertBefore	Method	System.Xml.XmlNode
Load	Method	void Load(string fi
LoadXml	Method	void LoadXml(string
Normalize	Method	void Normalize()
PrependChild	Method	System.Xml.XmlNode
ReadNode	Method	System.Xml.XmlNode
RemoveAll	Method	void RemoveAll()

Nous observons que l'objet est du type `System.Xml.XmlDocument`. Les membres de cet objet sont nombreux et constitués essentiellement de méthodes. Tout d'abord, affichons l'objet créé :

```
PS> $Configfile
xml
---
version="1.0"
Parameters
-----
Parameters
```

La sortie met en évidence l'élément racine du document XML. Sur cette base, essayons d'afficher le contenu de l'élément racine :

```
PS> $Configfile.Parameters
ServerList      : ServerList
```

```
SiteName      : WebPsx
InstallLocation : C:\WebIIS
HostHeader    : MLX-SDK
Port          : 80
LogPath       : C:\LogDirectory
LogLink       : C:\Directlink
WebAppPath    : C:\WebIIS\wpp
ApplicationPool : DefaultAppPool
AppName       : DevAppSoccer
```

Le nombre de propriétés est exactement de dix et l'affichage est total. Afficher le contenu d'une propriété se fait de la manière suivante :

```
PS> $Configfile.Parameters.HostHeader
MLX-SDK
```

En observant attentivement le fichier XML, on s'aperçoit que le nœud `ServerList` contient deux noeuds et que la sortie n'en montre pas la trace. La hiérarchie doit donc être parcourue jusqu'au bout :

```
PS> $Configfile.Parameters.ServerList.Server
NTDEVFL
NTDEVZT
```

Une première impression concernant la manipulation de fichiers XML avec PowerShell est qu'elle est quasi identique à la manipulation d'un autre type d'objet .Net. Utilisons maintenant la méthode `CreateElement()` afin d'ajouter un élément au document XML :

```
PS> $Misc = $Configfile.CreateElement("Misc")
```

```
Name      : Misc
LocalName : Misc
NamespaceURI :
Prefix    :
NodeType  : Element
ParentNode:
OwnerDocument : #document
IsEmpty   : True
Attributes: {}
HasAttributes: False
SchemaInfo: System.Xml.XmlName
InnerXml  :
InnerText :
NextSibling:
PreviousSibling:
```

```
Value      : 
ChildNodes : {}
FirstChild :
LastChild  :
HasChildNodes : False
IsReadOnly   : False
OuterXml    : <Misc />
BaseURI     :
```

L'élément que nous venons d'ajouter se nomme `Misc`, mais il n'a pas de contenu. Nous lui ajouterons le contenu suivant :

```
PS> $Misc.InnerText = "Web Site Configuration"
PS> $Configfile.Parameters.AppendChild($Misc)

#text
-----
Web Site Configuration
```

Vérifions à présent que les changements ont bien été appliqués :

```
PS> $Configfile.Parameters

ServerList      : ServerList
SiteName        : WebPsx
InstallLocation : C:\WebIIS
HostHeader      : MLX-SDK
Port            : 80
LogPath         : C:\LogDirectory
LogLink         : C:\Directlink
WebAppPath     : C:\WebIIS\wpp
ApplicationPool : DefaultAppPool
AppName         : DevAppSoccer
Misc            : Web Site Configuration
```

Le dernier élément correspond bien à celui que nous venons de créer. Il faut à présent sauvegarder ce changement dans ce même fichier XML :

```
PS> $Configfile.Save("C:\Web\Config_Parameters.xml")
```

Le fichier est sauvegardé et toutes ces manipulations ont été effectuées avec succès. Il faut cependant reconnaître qu'elles ne sont pas faciles à maîtriser. Un laps de temps est donc nécessaire avant de maîtriser la manipulation de fichiers XML avec PowerShell.

## Les cmdlets Export-Clixml et Import-Clixml

Dans des contextes particuliers, il est souvent utile ou nécessaire de conserver un certain nombre d'informations dans des fichiers XML, qui peuvent ensuite être réutilisées par exemple par d'autres outils de traitement de fichiers XML. Ce processus s'appelle sérialisation. Dans le contexte lié à PowerShell, il consiste à convertir des objets dans une structure XML, dans le but de les conserver pour les exploiter à un autre moment (nous parlons à ce stade précis de désérialisation). En ce qui concerne la sérialisation XML (car il existe d'autres types de sérialisation), il existe deux cmdlets que nous étudierons dans cette section : [Export-Clixml](#) et [Import-Clixml](#).

La cmdlet [Export-Clixml](#) est très utile lorsqu'il s'agit de donner une représentation XML d'objets. À la suite du processus de conversion des objets, ces derniers sont stockés dans un fichier. Par exemple, essayons d'obtenir une instance de la classe WMI [\[Win32\\_Processor\]](#) et sauvegardons le résultat dans un fichier XML :

```
PS> Get-WmiObject -Class Win32_Processor | Export-Clixml ProcessorInfos.xml
```

Nous pouvons observer l'absence de sortie à l'écran. Ce comportement est normal parce que l'instance a été convertie dans un format XML. Les informations sont donc conservées dans le fichier [ProcessorInfos.xml](#) et leur exploitation se fait via la cmdlet [Import-Clixml](#) :

```
PS> Import-Clixml .\ProcessorInfos.xml
```

PSComputerName	:	DC2K12ML
__GENUS	:	2
__CLASS	:	Win32_Processor
__SUPERCLASS	:	CIM_Processor
__DYNASTY	:	CIM_ManagedSystemElement
__RELPATH	:	Win32_Processor.DeviceID="CPU0"
__PROPERTY_COUNT	:	51
__DERIVATION	:	{CIM_Processor, CIM_LogicalDevice, CIM_Logic}
__SERVER	:	CIM_ManagedSystemElement}
__NAMESPACE	:	DC2K12ML
__PATH	:	root\cimv2
\\DC2K12ML\root\cimv2:Win32_Processor.Device	:	
AddressWidth	:	64
Architecture	:	9
Availability	:	3
Caption	:	AMD64 Family 20 Model 2 Stepping 0
ConfigManagerErrorCode	:	
ConfigManagerUserConfig	:	

```
CpuStatus : 1
CreationClassName : Win32_Processor
CurrentClockSpeed : 1397
CurrentVoltage : 33
DataWidth : 64
Description : AMD64 Family 20 Model 2 Stepping 0
DeviceID : CPU0
ErrorCleared :
ErrorDescription :
ExtClock :
Family : 2
InstallDate :
L2CacheSize : 512
L2CacheSpeed :
L3CacheSize : 0
L3CacheSpeed : 0
LastErrorCode :
Level : 20
LoadPercentage : 2
Manufacturer : AuthenticAMD
MaxClockSpeed : 1397
Name : AMD E1-1200 APU with Radeon(tm) HD
Graphics :
NumberOfCores : 1
NumberOfLogicalProcessors : 1
OtherFamilyDescription :
PNPDeviceID :
PowerManagementCapabilities :
PowerManagementSupported : False
ProcessorId : 078BFBFF00500F20
ProcessorType : 3
Revision : 512
Role : CPU
SecondLevelAddressTranslationExtensions : False
SocketDesignation : CPU socket #0
Status : OK
StatusInfo : 3
Stepping : 0
SystemCreationClassName : Win32_ComputerSystem
SystemName : DC2K12ML
UniqueId :
UpgradeMethod : 4
Version : Model 2, Stepping 0
VirtualizationFirmwareEnabled : True
VMMonitorModeExtensions : False
VoltageCaps : 2
...
...
```

Un point important à mettre en évidence est que les objets importés à l'aide de cette méthode ne sont pas des objets que l'on pourrait qualifier d'actifs. En effet, ils sont inertes et constitués essentiellement de propriétés :

```
PS> Import-Clixml .\ProcessorInfos.xml | gm

TypeName: Deserialized.System.Management.ManagementObject#root\

Name                           MemberType  Definition
----                           -----      -----
ToString                        Method     string ToString
PSComputerName                   NoteProperty System.String
AddressWidth                     Property   System.UInt16
Architecture                     Property   System.UInt16
Availability                     Property   System.UInt16
Caption                         Property   System.String
ConfigManagerErrorCode           Property   {get;set;}
ConfigManagerUserConfig          Property   {get;set;}
CpuStatus                        Property   System.UInt16
CreationClassName                 Property   System.String
CurrentClockSpeed                Property   System.UInt32
CurrentVoltage                   Property   System.UInt16
DataWidth                        Property   System.UInt16
Description                      Property   System.String
DeviceID                         Property   System.String
ErrorCleared                     Property   {get;set;}
ErrorException                   Property   {get;set;}
ExtClock                         Property   {get;set;}
Family                           Property   System.UInt16
InstallDate                      Property   {get;set;}
L2CacheSize                      Property   System.UInt32
L2CacheSpeed                     Property   {get;set;}
L3CacheSize                      Property   System.UInt32
L3CacheSpeed                     Property   System.UInt32
LastErrorCode                    Property   {get;set;}
Level                            Property   System.UInt16
...
...
```

Cela implique qu'il n'est pas possible d'agir concrètement sur les objets importés. En réalité, l'articulation des cmdlets `Export-Clixml` et `Import-Clixml` n'est pas faite pour l'action, mais pour la lecture et l'exploitation des objets dans leur version inerte. Les objectifs sont dès lors différents. Il faut aussi ajouter que la structure XML produite n'est pas manipulable à la manière d'un objet XML.

## Rechercher des informations à l'aide de la cmdlet Select-Xml

Une des perspectives proposées pour extraire des informations de fichier XML est celle du langage XPath (*XML Path Language*). Il permet de rechercher des informations à partir d'un fichier XML et de manière très précise, au sens où il dispose de sa propre syntaxe. Nous n'étudierons évidemment pas le langage XPath dans cet ouvrage, mais une compréhension de ses fondamentaux est nécessaire pour utiliser la cmdlet `Select-Xml`.

La commande `Select-Xml`, présente depuis la version 2 de PowerShell, offre la possibilité de construire des requêtes XPath sans passer par des méthodes plus classiques. De plus, ces requêtes sont optimisées pour la rapidité d'exécution.

Pour illustrer l'utilisation de la cmdlet `Select-Xml`, nous nous appuierons sur le fichier de configuration précédemment créé dans ce chapitre. Commençons par extraire, à partir de ce fichier de configuration, une liste de serveurs :

```
PS> Select-Xml -Path .\Config_Parameters.xml -XPath Parameters/ServerList/Server | ForEach-Object {$_.Node}

#text
-----
NTDEVFL
NTDEVZT
```

On peut aussi chercher le nom de l'application qui sera déployée :

```
PS> (Select-Xml -Path .\Config_Parameters.xml -XPath Parameters/AppName).Node

#text
-----
DevAppSoccer
```

La syntaxe que peut prendre le paramètre XPath peut paraître déroutante à première vue. Donc, un minimum de documentation est nécessaire. Les exemples que nous avons pris dans cette section sont simples, mais démontrent la puissance de la technologie XPath.



## PARTIE 4

# **Aller plus loin avec PowerShell**



# 21

## L'administration à distance

---

*Historiquement, la possibilité d'administrer des machines distantes date de l'arrivée de PowerShell version 2, ouvrant à l'époque de très grandes perspectives. Les scripts ou lignes de commandes pouvaient dès lors cibler plusieurs machines simultanément, et le besoin de copier les scripts n'était plus une étape préalable à leur exécution. En outre, les modalités de l'administration à distance proposées étaient différentes. Il pouvait s'agir de connexions temporaires ou de connexions persistantes. Aujourd'hui, ces axes ont évidemment été maintenus et améliorés.*

*Dans ce chapitre, nous apprendrons d'abord comment configurer l'accès à distance. Les connexions temporaires et persistantes étant deux choses différentes, nous essaierons de les différencier pour savoir quel type privilégié selon le contexte. Enfin, PowerShell version 3 offre plus de souplesse et de flexibilité dans la façon de gérer les sessions PowerShell, ce que nous verrons dans la dernière section.*

### SOMMAIRE

- ▶ Configurer l'accès à distance
- ▶ Les connexions temporaires
- ▶ Les connexions persistantes
- ▶ Quel type de connexion privilégié ?
- ▶ Se déconnecter d'une session pour se reconnecter plus tard

## Configurer l'accès à distance

L'accès à distance repose dans l'absolu sur le principe de la machine recevant la connexion, et non pas l'inverse. La communication à distance PowerShell s'établit à l'aide de la cmdlet `Enable-PSRemoting`. Cette cmdlet, qui doit être lancée avec les priviléges liés à un administrateur, effectue plusieurs actions dans l'ordre suivant.

- 1 Démarrer le service WinRM.
- 2 Attribue la valeur automatique au mode de démarrage du service WinRM.
- 3 Autorise le service WinRM à répondre aux requêtes venant de n'importe quelle adresse.
- 4 Crée une règle de pare-feu pour autoriser la communication basée sur le protocole WSMan.
- 5 Définit des modes de session particuliers (par exemple, `Microsoft.PowerShell` ou `Microsoft.PowerShell.Workflow`) s'ils n'existent pas déjà.
- 6 Active ces modes de session.
- 7 Modifie les règles de sécurité de ces modes de session de façon à autoriser l'accès à distance.
- 8 Redémarre le service WinRM de façon à ce que les changements précédents puissent prendre effet.

Nous constatons le pouvoir de la commande `Enable-PSRemoting` que lui confère la multiplicité de ses actions. Après avoir lancé une console en mode administrateur, il suffit d'écrire cette cmdlet sans aucune autre dépendance :

```
PS> Enable-PSRemoting

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable remote management of this
computer by using the Windows Remote
Management (WinRM) service.
This includes:
  1. Starting or restarting (if already started) the WinRM service
  2. Setting the WinRM service startup type to Automatic
  3. Creating a listener to accept requests on any IP address
  4. Enabling Windows Firewall inbound rule exceptions for WS-Management
traffic (for http only).

Do you want to continue?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is
"Y"): Y
WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.
```

Configured LocalAccountTokenFilterPolicy to grant administrative rights remotely to local users.

WinRM has been updated for remote management.

Created a WinRM listener on HTTP:///\* to accept WS-Man requests to any IP on this machine.

WinRM firewall exception enabled.

Confirm

Are you sure you want to perform this action?

Performing operation "Set-PSSessionConfiguration" on Target "Name:

microsoft.powershell SDDL:

O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will allow selected users to remotely run Windows PowerShell commands on this computer".

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): Y

Confirm

Are you sure you want to perform this action?

Performing operation "Set-PSSessionConfiguration" on Target "Name:

microsoft.powershell.workflow SDDL:

O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will allow selected users to remotely run Windows PowerShell commands on this computer".

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): Y

Confirm

Are you sure you want to perform this action?

Performing operation "Set-PSSessionConfiguration" on Target "Name:

microsoft.powershell132 SDDL:

O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will allow selected users to remotely run Windows PowerShell commands on this computer".

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): Y

Confirm

Are you sure you want to perform this action?

Performing operation "Set-PSSessionConfiguration" on Target "Name:

microsoft.windows.servermanagerworkflows SDDL:

O:NSG:BAD:P(A;;GA;;;IU)(A;;GA;;;BA)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will allow selected users to remotely run Windows PowerShell commands on this computer".

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): Y

L'utilisateur est invité à confirmer un certain nombre d'étapes qu'il lui appartient d'étudier dans le contexte qui est le sien. Ces étapes de configuration sont suffisantes dans le contexte d'un domaine. Dans le contexte d'un environnement autre qu'un domaine, des étapes supplémentaires seront alors nécessaires pour que la communication à distance puisse s'opérer.

## Les connexions temporaires

Dans le monde de PowerShell, il existe essentiellement deux catégories de connexions : les connexions temporaires, que nous évoquerons dans cette section, et les connexions persistantes.

Le principe des connexions temporaires est que les commandes exécutées à distance le sont sur la base d'une connexion établie juste pour le temps d'exécution des commandes. Une fois l'exécution terminée, PowerShell ferme la connexion et toutes les informations (variables, fonctions, etc.) relatives à cette connexion sont effacées.

PowerShell permet l'exécution temporaire à distance via la cmdlet [Invoke-Command](#). Le principe de cette cmdlet est d'invoquer un bloc de script ou plusieurs sur une machine distante et, une fois les opérations terminées, de fermer les connexions établies. La syntaxe de [Invoke-Command](#) est extrêmement riche :

```
PS> help Invoke-Command

NAME
    Invoke-Command

SYNOPSIS
    Runs commands on local and remote computers.

SYNTAX
    Invoke-Command [-ScriptBlock] <ScriptBlock> [-ArgumentList <Object[]>]
    [-InputObject <PSObject>] [-NoNewScope [<SwitchParameter>]]
    [<CommonParameters>]

    Invoke-Command [[-ConnectionUri] <Uri[]>] [-ScriptBlock] <ScriptBlock>
    [-AllowRedirection [<SwitchParameter>]] [-ArgumentList <Object[]>]
    [-AsJob [<SwitchParameter>]] [-Authentication <AuthenticationMechanism>]
    [-CertificateThumbprint <String>] [-ConfigurationName <String>]
    [-Credential <PSCredential>] [-EnableNetworkAccess [<SwitchParameter>]]
    [-HideComputerName [<SwitchParameter>]] [-InDisconnectedSession
    [<SwitchParameter>]] [-InputObject <PSObject>] [-JobName <String>]
    [-SessionOption <PSSessionOption>] [-ThrottleLimit <Int32>]
    [<CommonParameters>]
```

```
Invoke-Command [[-ConnectionUri] <Uri[]>] [-FilePath] <String>
[-AllowRedirection [<SwitchParameter>]] [-ArgumentList <Object[]>]
[-AsJob [<SwitchParameter>]] [-Authentication <AuthenticationMechanism>]
[-ConfigurationName <String>] [-Credential <PSCredential>]
[-EnableNetworkAccess [<SwitchParameter>]] [-HideComputerName
[<SwitchParameter>]] [-InDisconnectedSession [<SwitchParameter>]]
[-InputObject <PSObject>] [-JobName <String>] [-SessionOption
<PSSessionOption>] [-ThrottleLimit <Int32>] [<CommonParameters>]

Invoke-Command [[-ComputerName] <String[]>] [-FilePath] <String>
[-ApplicationName <String>] [-ArgumentList <Object[]>] [-AsJob
[<SwitchParameter>]] [-Authentication <AuthenticationMechanism>]
[-ConfigurationName <String>] [-Credential <PSCredential>]
[-EnableNetworkAccess [<SwitchParameter>]] [-HideComputerName
[<SwitchParameter>]] [-InDisconnectedSession [<SwitchParameter>]]
[-InputObject <PSObject>] [-JobName <String>] [-Port <Int32>]
[-SessionName <String[]>] [-SessionOption <PSSessionOption>]
[-ThrottleLimit <Int32>] [-UseSSL [<SwitchParameter>]] [<CommonParameters>]

Invoke-Command [[-ComputerName] <String[]>] [-ScriptBlock] <ScriptBlock>
[-ApplicationName <String>] [-ArgumentList <Object[]>] [-AsJob
[<SwitchParameter>]] [-Authentication <AuthenticationMechanism>]
[-CertificateThumbprint <String>] [-ConfigurationName <String>] [-Credential
<PSCredential>] [-EnableNetworkAccess [<SwitchParameter>]]
[-HideComputerName [<SwitchParameter>]] [-InDisconnectedSession
[<SwitchParameter>]] [-InputObject <PSObject>] [-JobName <String>] [-Port
<Int32>] [-SessionName <String[]>] [-SessionOption <PSSessionOption>]
[-ThrottleLimit <Int32>] [-UseSSL [<SwitchParameter>]] [<CommonParameters>]

Invoke-Command [[-Session] <PSSession[]>] [-FilePath] <String>
[-ArgumentList <Object[]>] [-AsJob [<SwitchParameter>]] [-HideComputerName
[<SwitchParameter>]] [-InputObject <PSObject>] [-JobName <String>]
[-ThrottleLimit <Int32>] [<CommonParameters>]

Invoke-Command [[-Session] <PSSession[]>] [-ScriptBlock] <ScriptBlock>
[-ArgumentList <Object[]>] [-AsJob [<SwitchParameter>]] [-HideComputerName
[<SwitchParameter>]] [-InputObject <PSObject>] [-JobName <String>]
[-ThrottleLimit <Int32>] [<CommonParameters>]
(...)
```

Nous ne rentrerons évidemment pas dans le détail de ces différents paramètres, mais nous évoquerons ceux qui nous permettront d'utiliser cette cmdlet de manière adéquate. Par exemple, essayons sur un serveur distant d'obtenir les dix processus qui consomment le plus de mémoire :

```
PS> C:\Users\Administrator> Invoke-Command -ComputerName DC2K12ML -ScriptBlock
{get-process | sort ws -Descending | select -first 10 -Property
ProcessName,Id,WS,CPU,PSComputerName}
```

ProcessName	Id	WS	CPU	PSComputerName
powershell	1564	147652608	18,9697216	DC2K12ML
wsmprovhost	3068	61259776	1,8564119	DC2K12ML
dns	1452	48586752	6,8952442	DC2K12ML
ServerManager	832	47198208	13,4316861	DC2K12ML
explorer	1912	42360832	4,9608318	DC2K12ML
Microsoft.Active..	1356	37842944	5,9436381	DC2K12ML
lsass	544	36184064	10,7952692	DC2K12ML
dwm	892	32288768	7,2072462	DC2K12ML
svchost	928	28323840	15,1008968	DC2K12ML
WmiPrvSE	1836	19701760	1,8564119	DC2K12ML

L'opération s'est déroulée en quelques secondes. Une connexion a été initiée de la part de PowerShell et clôturée aussitôt. Pour spécifier à `Invoke-Command` le nom de la machine distante, nous avons utilisé le paramètre `-ComputerName`. Puis les commandes exécutées l'ont été à l'aide du paramètre `-ScriptBlock`, qui peut en contenir plusieurs. La propriété `PSComputerName` apparaît quant à elle dans un contexte particulier, qui est justement celui de la communication à distance.

L'exemple suivant collecte les événements du journal PowerShell et les filtre. Comme il s'agit d'une machine distante qui requiert un autre niveau de privilège, le paramètre `-Credential` doit être appelé pour indiquer un autre compte :

```
PS> Invoke-Command -ComputerName DC2K12KX -ScriptBlock {Get-EventLog -log
"Windows PowerShell" | where {$_.Message -like "*filesystem*"} } -Credential
NTDEV\Administrator
```

PowerShell nous invite alors à indiquer le mot de passe en lien avec le compte spécifié.

**Figure 21–1**

PowerShell invite l'utilisateur à entrer un mot de passe.



Une fois le mot de passe validé, la commande est exécutée à distance et nous donne les résultats suivants :

EntryType	Source	InstanceId	Message
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
Information	PowerShell	600	Provider "FileSystem" is Started. ...
(...)			

La cmdlet `Invoke-Command` représente le cœur de la communication à distance de PowerShell. Les exemples que nous avons pris illustrent comment administrer une machine distante pour des opérations ponctuelles, mais il est parfois nécessaire que la connexion soit maintenue.

## **Les connexions persistantes**

Lorsque PowerShell initie une connexion temporaire, celle-ci a un coût en termes de ressources. Et lorsque ce même type de connexion doit être initialisé sur plusieurs machines distantes, alors ce coût devient de plus en plus lourd à gérer pour PowerShell, qui devient de moins en moins performant, surtout s'il y a répétition.

Pour remédier à ce problème, il est possible de créer des connexions persistantes permettant de créer des sessions en quelque sorte statiques, au sens de permanence de la connexion. De nombreuses cmdlets servent à gérer les sessions PowerShell (que l'on nomme PSSessions).

Tableau 21–1 Liste des cmdlets en lien avec les sessions PowerShell

Cmdlet	Description
New-PSSession	Crée une connexion persistante sur une machine locale ou distante.
Get-PSSession	Obtient les sessions PowerShell créées sur une machine locale ou distante.
Remove-PSSession	Ferme une ou plusieurs session(s) créée(s).
Enter-PSSession	Démarre une session interactive avec une machine distante.
Exit-PSSession	Ferme une session interactive avec une machine distante.
Disconnect-PSSession	Déconnecte d'une session PowerShell.
Connect-PSSession	Reconnecte avec la session spécifiée.
Receive-PSSession	Collecte les résultats des commandes exécutées lors d'une session pendant sa déconnexion.
Import-PSSession	Importe les commandes (cmdlets, fonctions et alias) à partir d'une session spécifique vers la session actuelle.
Export-PSSession	Exporte les commandes d'une session PowerShell pour constituer un module.

Ces cmdlets s'articulent très fortement entre elles. L'exemple suivant met en évidence la création d'une session PowerShell et son utilisation pour entrer en mode interactif avec la machine distante :

```
PS> $Session = New-PSSession -ComputerName DC2K12ML
PS> Enter-PSSession -Session $Session
[DC2K12ML]: PS>
```

L'entrée en mode interactif signifie que l'invite de commande change de nature. Cette dernière en vient donc à être constituée à la fois de l'invite de commande classique, mais aussi du nom de la machine à partir de laquelle une session PowerShell a été créée. À présent que nous sommes en mode interactif, toutes les commandes que nous allons taper prendront effet sur la machine distante, et non la machine locale :

```
[DC2K12ML]: PS> get-netadapter | fl
```

Name	:	Ethernet
InterfaceDescription	:	Intel(R) 82574L Gigabit Network Connection
InterfaceIndex	:	12
MacAddress	:	00-0C-29-F7-4E-3C
MediaType	:	802.3
PhysicalMediaType	:	802.3
InterfaceOperationalStatus	:	Up
AdminStatus	:	Up
LinkSpeed(Gbps)	:	1
MediaConnectionState	:	Connected

```
ConnectorPresent      : True
DriverInformation     : Driver Date 2012-02-29 Version 12.0.150.0 NDIS 6.30

[DC2K12ML]: PS> Get-WmiObject -Class win32_bios

SMBIOSBIOSVersion   : 6.00
Manufacturer        : Phoenix Technologies LTD
Name                : PhoenixBIOS 4.0 Release 6.0
SerialNumber        : VMware-56 4d 30 22 5a 67 12 53-33 eb 1c df 96 f7 4e 3c
Version             : INTEL - 6040000

[DC2K12ML]: PS> $env:COMPUTERNAME
DC2K12ML
```

Pour fermer la session interactive, il faut taper en console la commande suivante :

[DC2K12ML]: PS> **Exit-PSSession**  
PS>

Le prompt a changé, ce qui prouve que la session a bien été fermée. Attention cependant, *fermer* ne veut pas dire *supprimer*, car la session que nous avons créée existe toujours :

```
PS> Get-PSSession
```

<b>Id</b>	<b>Name</b>	<b>ComputerName</b>	<b>State</b>	<b>ConfigurationName</b>	<b>Availability</b>
7	Session7	DC2K12ML	Opened	Microsoft.PowerShell	Available

De plus, son statut a comme valeur `Opened`, ce qui veut dire qu'elle est toujours disponible, donc toujours exploitable :

```
PS> Invoke-Command -Session $Session -ScriptBlock {Get-EventLog -log "Windows PowerShell" | where {$_.Message -like "*filesystem*"} | Select EntryType, Source, InstanceId, Message}
```

```
Information PowerShell  
Information PowerShell  
(...)
```

```
600 Provider "FileSystem" is Started. ...  
600 Provider "FileSystem" is Started. ...
```

Cette ligne de commande sollicite la cmdlet `Invoke-Command` que nous avons utilisée dans la section précédente, à la différence qu'au lieu d'invoquer le paramètre `-ComputerName`, nous avons appelé `-Session`, qui spécifie la session que nous avons créée.

Pour définitivement détruire notre session, puisque nous n'en avons plus besoin, nous utiliserons la commande `Remove-PSSession` :

```
PS> Remove-PSSession -Session $Session -Verbose  
VERBOSE: Performing operation "Remove" on Target "DC2K12ML".
```

Nous n'avons pas encore évoqué les autres cmdlets en lien avec les sessions PowerShell. Cela sera fait dans la dernière section de ce chapitre.

## Quel type de connexion privilégier ?

Depuis le début du chapitre, nous avons finalement scindé l'administration à distance en deux modalités. La première repose sur le principe de la connexion temporaire et la seconde sur celui de la connexion permanente. Le fait de choisir l'un ou l'autre de ces principes est la responsabilité de l'utilisateur. En effet, celui-ci effectue ses opérations dans des contextes particuliers et relatifs à des données bien précises à partir desquelles son choix doit s'établir.

Par exemple, un utilisateur peut démarrer une opération avant d'aller à une réunion. Si cette opération est longue, alors il pourra opter pour une connexion permanente, puis se déconnecter de cette session pendant que l'opération se déroule et donc assister à sa réunion. Lorsqu'elle sera terminée, l'utilisateur pourra, parmi plusieurs choix possibles, surveiller le déroulement de l'opération et analyser les résultats attendus. Au contraire, si l'utilisateur, avant d'assister à une réunion, doit effectuer une opération dont le déroulement est très court, alors son choix devra se porter sur le principe de la connexion temporaire, évitant ainsi la sollicitation de ressources de manière inutile.

Ces cas sont très généraux, mais ils aident à mieux comprendre l'orientation à suivre en fonction de situations dont, encore une fois, les configurations sont multiples. Il faut remarquer avec force que ce n'est pas l'une ou l'autre des propositions qu'il faut suivre dans l'absolu, mais parfois la conjonction des deux principes. Finalement, ce qu'il faut retenir est qu'il y a une intelligence à appliquer en fonction des différents contextes, car une même opération peut s'inscrire dans des contextes variés.

## Se déconnecter d'une session pour se reconnecter plus tard

PowerShell version 3 rend plus flexible la gestion des PSSessions ; à présent, il est possible d'initialiser une session, de s'en déconnecter pour se reconnecter plus tard. Et le processus de reconnexion peut se faire soit à partir de la machine où la session a été créée, soit à partir d'une autre machine. Cependant, la contrainte est que PowerShell version 3 soit installé sur les deux machines.

Pour illustrer ce que nous venons d'évoquer, commençons par créer une session sur un serveur distant :

```
PS> New-PSSession -ComputerName DC2K12ML
```

Id	Name	ComputerName	State	ConfigurationName	Availability
1	Session1	DC2K12ML	Opened	Microsoft.PowerShell	Available

La session a été créée et a comme état la valeur `Opened`. À présent que cette étape est franchie, déconnectons-nous de cette session :

```
PS> Disconnect-PSSession -Name Session1
```

Id	Name	ComputerName	State	ConfigurationName	Availability
1	Session1	DC2K12ML	Disconnected	Microsoft.PowerShell	None

Supposons maintenant que nous devions redémarrer notre machine pour telle ou telle raison. Après le redémarrage, lançons une session PowerShell et tapons en console la commande suivante :

```
PS> Get-PSSession -ComputerName DC2K12ML
```

Id	Name	ComputerName	State	ConfigurationName	Availability
1	Session1	DC2K12ML	Disconnected	Microsoft.PowerShell	None

Nous pouvons constater que la session a été maintenue même après le redémarrage de la machine. Une reconnexion est donc possible :

```
PS> Connect-PSSession -Id 1
```

Id	Name	ComputerName	State	ConfigurationName	Availability
1	Session1	DC2K12ML	Opened	Microsoft.PowerShell	Available

Nous pouvons maintenant exécuter les commandes que nous voulons dans un contexte tout à fait habituel :

```
PS> Invoke-Command -Session (Get-PSSession) -ScriptBlock {Get-Service | where status -eq 'running' | select name,status}

Name          Status
----          -----
ADWS          Running
BFE           Running
BrokerInfrastructure Running
COMSysApp     Running
CryptSvc      Running
DcomLaunch    Running
Dfs            Running
DFSR           Running
Dhcp           Running
DNS            Running
Dnscache       Running
DPS            Running
DsmSvc         Running
EventLog       Running
```

Cette flexibilité touche aussi la cmdlet `Invoke-Command`. En effet, cette dernière offre maintenant la perspective de créer une session, d'exécuter la ou les commande(s) correspondante(s) tout en opérant une déconnexion automatique, et ce, avant même qu'une sortie soit produite. Le paramètre à appeler est `-InDisconnectedSession` :

```
PS> Invoke-Command -ComputerName DC2K12ML -InDisconnectedSession -ScriptBlock {Get-WinEvent -LogName "Windows PowerShell"}
```

Id	Name	ComputerName	State	ConfigurationName	Availability
3	Session2	DC2K12ML	Disconnected	Microsoft.PowerShell	None

Cette ligne de commande collecte les événements du journal PowerShell sur un serveur distant. La sortie affichée est un objet PSSession nous indiquant que la session est déconnectée. La commande `Get-WinEvent` continue d'être exécutée dans une session en cours d'exécution, mais qui est déconnectée. La question est maintenant de savoir comment récupérer le résultat de cette commande. La réponse se trouve du côté de la cmdlet `Receive-PSSession` :

```
PS> Receive-PSSession -Id 3

Message          :
PSCoputerName   : DC2K12ML
RunspaceId       : 63e8a762-f704-4d7e-8e41-5569312cb1cf
Id              : 400
Version          :
Qualifiers       : 0
Level            : 4
Task             : 4
Opcode           :
Keywords         : 36028797018963968
RecordId         : 333
ProviderName     : PowerShell
ProviderId       :
LogName          : Windows PowerShell
ProcessId        :
ThreadId         :
MachineName      : DC2K12ML.ntdev.m1
UserId           :
TimeCreated      : 09/07/2013 20:42:17
ActivityId       :
RelatedActivityId:
ContainerLog      : windows powershell
MatchedQueryIds  : {}
Bookmark         : System.Diagnostics.Eventing.Reader.EventBookmark
LevelDisplayName  :
OpcodeDisplayName:
TaskDisplayName   :
KeywordsDisplayNames: {}
Properties        : {System.Diagnostics.Eventing.Reader.EventProperty,
System.Diagnostics.Eventing.Reader.EventProperty,
System.Diagnostics.Eventing.Reader.EventProperty}
(...)
```

La commande `Receive-PSSession` a opéré une reconnexion automatique avec la session spécifiée. Cette étape franchie, le résultat a pu être obtenu sous forme de liste. Nous aurions très bien pu convertir le résultat en tant que job, de la façon suivante :

```
PS> Receive-PSSession -Id 5 -OutTarget job
```

L'articulation de l'ensemble de ces cmdlets dans le cadre de la gestion de sessions PowerShell nécessite un temps de pratique indispensable afin de mieux maîtriser les contours de la souplesse offerte.



# 22

## Les workflows sous PowerShell

---

*La plus grande nouveauté apportée par PowerShell version 3 est sans conteste la technologie Windows Workflow Foundation (WWF). L'objectif annoncé, en conjuguant ces deux technologies, est de pouvoir écrire des workflows dans l'écosystème PowerShell. De ce fait, les commandes, quelle que soit leur nature, ont une puissance et une robustesse décuplées, car elles profitent de la technologie WWF. Et pour les entreprises, cette perspective duelle est un formidable moyen de revoir un certain nombre de modes d'automatisation liés, par exemple, à leurs infrastructures.*

*La technologie WWF apparue avec la version 3 du framework .NET demanderait un livre à elle toute seule, tant elle est complexe. Néanmoins, comme cet ouvrage s'adresse essentiellement aux administrateurs, nous étudierons comment écrire des workflows en langage PowerShell, en commençant par définir ce qu'est un workflow, puis en différenciant ce dernier d'une fonction, différence amenant à nous familiariser avec la syntaxe particulière d'un workflow et à mettre en lumière quelques exemples afin de mieux comprendre cette dualité.*

### SOMMAIRE

- ▶ Qu'est-ce qu'un workflow ?
- ▶ La différence entre un workflow et une fonction
- ▶ La syntaxe
- ▶ Quelques exemples

## Qu'est-ce qu'un workflow ?

Un workflow, dans son sens absolu, est une orchestration d'activités. Une activité représente une action à mener et il peut y en avoir une ou plusieurs. L'ensemble de ces activités est coordonné dans le but de consolider une action globale qui n'est autre que le workflow lui-même. Pour lier cette définition à l'environnement PowerShell, les activités en question ne sont rien d'autre que des commandes. Donc, dans cet environnement, un workflow est un ensemble d'activités ou commandes, pouvant être traité de manière séquentielle ou parallèle et qui propose une certaine robustesse et résilience.

## Les workflows comme synonyme de productivité

D'une manière générale, les workflows sont écrits dans des contextes liés à ce que l'on trouve dans les entreprises, c'est-à-dire des contextes environnementaux hétérogènes. De plus, leur durée d'exécution est souvent longue (jusqu'à plusieurs heures) et donc, la nécessité d'écrire un workflow relève d'abord de la planification et de l'architecture quant à la manière dont les problèmes seront résolus.

Les workflows peuvent être écrits de deux façons :

- en passant par le langage déclaratif XAML (*eXtensible Application Markup Language*), utilisé avec la technologie *Windows Workflow Foundation* ;
- en passant par le langage PowerShell (ce qui est l'objet de ce chapitre).

Pour un administrateur, l'utilisation des workflows dans l'écosystème PowerShell est un moyen très puissant de concilier automatisation avec robustesse et résilience. En effet, les workflows écrits de cette façon pourront survivre, par exemple, à des redémarrages, volontaires ou non, ou même à des interruptions ou discontinuités au niveau réseau.

Prenons un exemple concret. Admettons qu'au cours d'une vaste opération, il faille :

- créer et déployer des machines virtuelles ;
- démarrer ces machines virtuelles ;
- configurer un certain nombre de services que nous considérons comme importants ;
- installer et configurer des applications ;
- redémarrer les machines virtuelles ;
- configurer les journaux d'événements en nous basant sur la politique de l'entreprise ;
- créer des partages avec une politique de sécurité bien précise ;
- établir un rapport web de l'état de ces machines et éventuellement l'envoyer par courrier électronique aux personnes concernées par cette opération.

Cette chaîne d'actions pourrait être réfléchie dans le cadre d'une architecture orchestrée par un workflow. D'ailleurs, ce serait la voie ultime à privilégier, car il s'agit aussi d'être productif et d'opter pour les meilleures méthodes possible, même si celles-ci requièrent inévitablement un temps d'adaptation.

Typiquement, un workflow écrit via PowerShell ressemble à ce qui suit.

#### Syntaxe élémentaire d'un workflow

```
workflow ① nom_workflow ② {  
    <Liste d'activités à effectuer> ③  
}
```

Un workflow est déclaré à l'aide du mot-clé `workflow` ①. Il est aussi identifié par un nom ② et contient au moins une activité ③. L'exemple suivant illustre une définition très simple d'un workflow.

#### Exemple de déclaration d'un workflow

```
workflow Ps-Wf {  
    "This is our first workflow"  
}
```

Le workflow étant défini, il n'y a plus qu'à l'invoquer :

```
PS> Ps-Wf  
This is our first workflow
```

Le workflow que nous venons de créer a été ajouté à la session en cours :

```
PS> Get-Command - CommandType Workflow  
  
 CommandType      Name  
-----          ----  
 Workflow        Ps-Wf
```

## La relation PowerShell-WWF

La question que nous pouvons poser est celle de la relation entre PowerShell et WWF. A priori, il n'est pas du tout compliqué de déclarer un workflow avec PowerShell, mais en analysant davantage, on s'aperçoit que PowerShell effectue un travail gigantesque en arrière-plan. Pour nous en convaincre, listons les membres du workflow que nous venons de créer :

```
PS> $workflow = Get-Command - CommandType Workflow
PS> $workflow | gm

TypeName: System.Management.Automation.WorkflowInfo

Name          MemberType      Definition
----          -----          -----
Equals        Method         bool Equals(System.Object obj)
GetHashCode   Method         int GetHashCode()
GetType       Method         type GetType()
ResolveParameter Method       System.Management.Automation.Para...
ToString      Method         string ToString()
CmdletBinding  Property      bool CmdletBinding {get;}
CommandType   Property      System.Management.Automation.Comm...
DefaultParameterSet Property    string DefaultParameterSet {get;}
Definition    Property      string Definition {get;}
Description   Property      string Description {get;set;}
HelpFile      Property      string HelpFile {get;}
Module        Property      psmoduleinfo Module {get;}
ModuleName   Property      string ModuleName {get;}
Name          Property      string Name {get;}
NestedXamlDefinition Property  string NestedXamlDefinition {get;
Noun          Property      string Noun {get;}
Options        Property      System.Management.Automation.Scop...
OutputType     Property      System.Collections.ObjectModel.Re...
Parameters    Property      System.Collections.Generic.Dictio...
ParameterSets  Property      System.Collections.ObjectModel.Re...
RemotingCapability Property  System.Management.Automation.Remo...
ScriptBlock   Property      scriptblock ScriptBlock {get;}
Verb          Property      string Verb {get;}
Visibility    Property      System.Management.Automation.Sess...
WorkflowsCalled Property  System.Collections.ObjectModel.Re...
XamlDefinition Property  string XamlDefinition {get;}
HelpUri       ScriptProperty System.Object HelpUri {get=$oldPr
```

Notre workflow contient de nombreux membres. La propriété `ScriptBlock` fait apparaître son implémentation :

```
PS> $workflow.ScriptBlock

[CmdletBinding()]
param (
    [hashtable[]] $PSPParameterCollection,
    [string[]] $PSComputerName,
    [ValidateNotNullOrEmpty()] $PSCredential,
    [uint32] $PSConnectionRetryCount,
    [uint32] $PSConnectionRetryIntervalSec,
```

```
[ValidateRange(1, 2147483)][uint32] $PSRunningTimeoutSec,
[ValidateRange(1, 2147483)][uint32] $PSElapsedTimeoutSec,
[bool] $PSPersist,
[ValidateNotNullOrEmpty()][System.Management.Automation
    .Runspaces.AuthenticationMechanism] $PSAuthentication,
[ValidateNotNullOrEmpty()][System.Management
    .AuthenticationLevel] $PSAuthenticationLevel,
[ValidateNotNullOrEmpty()][string] $PSApplicationName,
[uint32] $PSPort,
[switch] $PSUseSSL,
[ValidateNotNullOrEmpty()][string] $PSConfigurationName,
[ValidateNotNullOrEmpty()][string[]] $PSConnectionURI,
[switch] $PSAllowRedirection,
[ValidateNotNullOrEmpty()][System.Management.Automation
    .Remoting.PSSessionOption] $PSSessionOption,
[ValidateNotNullOrEmpty()][string] $PSCertificateThumbprint,
[hashtable] $PSPrivateMetadata,
[switch] $AsJob,
[string] $JobName,
[Parameter(ValueFromPipeline=$true)]$InputObject
)
begin {
    function Ps-Wf {

        [CmdletBinding()]
        param (
            $PSInputCollection,
            [string[]] $PSComputerName,
            [ValidateNotNullOrEmpty()] $PSCredential,
            [uint32] $PSConnectionRetryCount,
            [uint32] $PSConnectionRetryIntervalSec,
            [ValidateRange(1, 2147483)][uint32] $PSRunningTimeoutSec,
            [ValidateRange(1, 2147483)][uint32] $PSElapsedTimeoutSec,
            [bool] $PSPersist,
            [ValidateNotNullOrEmpty()][System.Management.Automation
                .Runspaces.AuthenticationMechanism] $PSAuthentication,
            [ValidateNotNullOrEmpty()][System.Management
                .AuthenticationLevel] $PSAuthenticationLevel,
            [ValidateNotNullOrEmpty()][string] $PSApplicationName,
            [uint32] $PSPort,
            [switch] $PSUseSSL,
            [ValidateNotNullOrEmpty()][string] $PSConfigurationName,
            [ValidateNotNullOrEmpty()][string[]] $PSConnectionURI,
            [switch] $PSAllowRedirection,
            [ValidateNotNullOrEmpty()][System.Management.Automation
                .Remoting.PSSessionOption] $PSSessionOption,
            [ValidateNotNullOrEmpty()][string] $PSCertificateThumbprint,
            [hashtable] $PSPrivateMetadata,
```

```
[switch] $AsJob,
[string] $JobName,
[Parameter(ValueFromPipeline=$true)]$InputObject
)           $PSBoundParameters
}

$PSInputCollection = New-Object 'System.Collections.Generic
.List[PSObject]'
}
(...)
```

La sortie a été volontairement tronquée car elle est très longue. Toutefois, avec cet extrait, on peut se rendre compte de la complexité de l'implémentation des workflows, à laquelle il faut ajouter la représentation de ce même workflow d'une manière compréhensible par le moteur WWF :

```
PS> $workflow.XamlDefinition

<Activity
  x:Class="Microsoft.PowerShell.DynamicActivities.Activity_1829384205"
  xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
  xmlns:sad="clr-namespace:System.Activities.Debugger;
    assembly=System.Activities"
  xmlns:local="clr-namespace:Microsoft.PowerShell.DynamicActivities"
  xmlns:mva="clr-namespace:Microsoft.VisualBasic.Activities;
    assembly=System.Activities"
  mva:VisualBasic.Settings="Assembly references and imported namespaces
    serialized as XML namespaces"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:ns0="clr-namespace:System;assembly=mscorlib"
  xmlns:ns1="clr-namespace:Microsoft.PowerShell.Utility.Activities;
    assembly=Microsoft.PowerShell.Utility.Activities"
  xmlns:ns2="clr-namespace:Microsoft.PowerShell.Activities;
    assembly=Microsoft.PowerShell.Activities"
  xmlns:ns3="clr-namespace:System.Activities;assembly=System.Activities"
  xmlns:ns4="clr-namespace:System.Management.Automation;
    assembly=System.Management.Automation"
  >
<Sequence>
  <ns2:SetPSWorkflowData>
    <ns2:SetPSWorkflowData.OtherVariableName>Position
    </ns2:SetPSWorkflowData.OtherVariableName>
    <ns2:SetPSWorkflowData.Value>
      <ns3:InArgument x:TypeArguments="ns0:Object">
        <ns2:PowerShellValue x:TypeArguments="ns0:Object"
          Expression="'1:17:Ps-Wf'" />
      </ns3:InArgument>
```

```
</ns2:SetPSWorkflowData.Value>
</ns2:SetPSWorkflowData>
<ns1:WriteOutput>
    <ns1:WriteOutput.InputObject>
        <InArgument x:TypeArguments="ns4:PSObject[]">
            <ns2:PowerShellValue x:TypeArguments="ns4:PSObject[]">
                Expression=""This is our first workflow"" />
            </InArgument>
        </ns1:WriteOutput.InputObject>
    </ns1:WriteOutput>
    <Sequence.Variables>
        <Variable Name="WorkflowCommandName" x:TypeArguments="ns0:String"
            Default = "Ps-Wf" />
    </Sequence.Variables>
</Sequence>
</Activity>
```

Pourtant, le workflow en question est très simple et on ne peut plus court. Ce qu'il faut en conclure est que l'abstraction fournie par PowerShell est élevée. Vous aurez remarqué que le mode de déclaration d'un workflow est quasi identique à celui d'une fonction. Cela est vrai, mais la comparaison s'arrête là ; un workflow *n'est pas* une fonction.

## La différence entre un workflow et une fonction

Au début de cet ouvrage, nous avons insisté sur le caractère pluriel de la notion de commande (cmdlet, fonction...). PowerShell les considère comme telles même s'il sait parfaitement les distinguer sur un plan essentiel.

PowerShell version 3 introduit un nouveau genre de commande : les workflows. Ces derniers sont en apparence similaires aux fonctions, mais ils sont en réalité totalement différents. Voici les caractéristiques essentielles d'une fonction.

- Contrairement aux workflows, les commandes sont considérées comme telles dans une fonction.
- Les fonctions sont exécutées par PowerShell.
- Le contenu d'une fonction est exécuté de manière séquentielle (sauf dans le cadre de l'utilisation de jobs).
- Une fonction dispose de l'intégralité du langage et de sa syntaxe.
- La résilience d'une fonction est possible, mais à travers des astuces de programmation.
- Les fonctions ressemblent aux cmdlets et sont aussi leurs voies d'accès.

Voici les caractéristiques des workflows.

- Les commandes sont considérées comme des activités.
- Les workflows sont exécutés par le moteur WWF.
- Le contenu d'un workflow peut être exécuté de manière séquentielle, mais aussi parallèle.
- L'écosystème dans lequel évolue un workflow est limité en termes d'éléments de langage.
- Les workflows proposent une robustesse que n'ont pas les fonctions.
- Une persistance des données est possible.
- Les données ne peuvent pas dans l'absolu être partagées entre les activités, sauf dans certains cas.

À la lumière de ces caractéristiques, on peut considérer que les workflows n'ont effectivement rien à voir avec les fonctions, sauf en apparence.

## La syntaxe

La syntaxe et la sémantique d'un workflow sont très spécifiques. Pour rappel, voici la syntaxe de base.

### Syntaxe élémentaire d'un workflow

```
workflow nom_workflow {  
    <Liste d'activités à effectuer>  
}
```

À cette syntaxe de base viennent s'ajouter d'autres éléments donnant les orientations que peut prendre un workflow.

## Paralléliser des opérations

Une des forces de la technologie WWF est la possibilité de paralléliser des activités (ou commandes dans notre contexte). De plus, les modalités de cette potentielle parallélisation sont diverses. Il y a essentiellement deux éléments de langage pour profiter de la parallélisation des opérations : le premier est l'instruction `parallel` et le second l'instruction `foreach -parallel`.

L'instruction `parallel` contient des commandes exécutées en parallèle.

### Syntaxe de l'instruction parallel

```
workflow nom_workflow {  
    parallel ❶ {  
        <commande 1> ❷  
        <commande 2>  
        <commande 3>  
        <commande 4>  
    }  
}
```

Les commandes ❷ listées à l'intérieur du bloc `parallel` ❶ sont précisément exécutées en parallèle, même s'il n'y a pas d'ordre de démarrage précis.

L'instruction `foreach -parallel` diffère de `foreach`, en ce sens que la première traite chaque élément d'une collection de manière parallèle, contrairement à la seconde qui les traite de manière séquentielle. La syntaxe de l'instruction `foreach -parallel` s'articule comme suit.

### Syntaxe de l'instruction foreach -parallel

```
workflow nom_workflow {  
    foreach -parallel ❶ ($item in $collection) ❷ {  
        <actions> ❸  
    }  
}
```

On peut observer que l'instruction `foreach` prend un paramètre ❶ qui est exclusif au contexte inhérent aux workflows. L'instruction `foreach` énumère l'ensemble d'une collection ❷ et effectue de manière simultanée une ou des actions ❸ sur chaque élément de la collection.

## Effectuer des opérations de manière séquentielle

Non seulement on peut paralléliser des actions, mais il est également possible de les exécuter dans un ordre bien précis, avec l'instruction `sequence`.

### Syntaxe de l'instruction sequence

```
workflow nom_workflow {  
    sequence ❶ {  
        <commande 1> ❷  
        <commande 2>  
    }  
}
```

```

    <commande 3>
    <commande 4>
}
}

```

L'instruction `sequence` ① sert à réaliser les opérations ② dans l'ordre indiqué. Cela peut être nécessaire, par exemple dans des configurations ayant des dépendances.

## L'instruction `InlineScript`

Le bloc `InlineScript` a la particularité d'être exécuté dans un environnement PowerShell séparé de l'environnement des workflows. En conséquence, dans un bloc `InlineScript`, toute commande, tout objet ou méthode d'objet peuvent être invoqués, contrairement aux blocs `sequence` et `parallel` qui sont bridés parce que leur environnement est restreint.

### Syntaxe de l'instruction `InlineScript`

```

workflow nom_workflow {
    InlineScript ① {
        <commande 1> ②
        <commande 2>
        <commande 3>
        <commande 4>
    }
}

```

L'utilisation du bloc `InlineScript` ① permet aux différentes commandes ② d'accéder à leurs données respectives. Il est même possible de passer des données à l'instruction `InlineScript` à l'aide de la variable `$Using`. Cette dernière indique à PowerShell d'utiliser des données appartenant à la portée du workflow et de les passer au bloc `InlineScript` (qui n'est pas dans la même portée) :

### Utilisation de la variable `$Using` dans un bloc `InlineScript`

```

workflow inline-wf {
    Param([string]$Path)
    InlineScript {
        if (test-path -path $using:path) {
            & $using:path
        }
    }
}

```

Le transfert de données est donc possible avec les workflows, et ce, en partie grâce à la variable `$Using`, qui par ailleurs est utilisé dans d'autres contextes (ceux inhérents aux fonctions).

**NOTE Concernant la variable \$Using**

La variable `$Using` ne peut être utilisée que dans le bloc `InlineScript`. Pour passer des données à travers l'ensemble des instructions contenues dans un workflow, il faut utiliser le préfixe `$Workflow:` (ex : `$Workflow:var`).

## Les workflows en pratique

Les exemples qui vont suivre aideront à comprendre comment écrire un workflow. Tout d'abord, en voici un mettant en évidence l'exécution d'opérations en parallèle :

```
workflow Get-Infos {
    Parallel ① {
        Sequence ② {
            write-verbose -Message "Processing workflow $workflowcommandname"
            on -> $pscomputername
            write-verbose -Message "This is a sample demo workflow"
            write-verbose -Message "Getting informations about remote machines"
        }
        InlineScript ③ {
            $os = Get-WmiObject -class Win32_OperatingSystem -ComputerName
                $pscomputername
            $proc = Get-WmiObject -class Win32_Processor -Computername
                $pscomputername | Select-Object -first 1
            $obj = New-Object -TypeName PSObject

            $obj | Add-Member -NotePropertyName BuildNum -NotePropertyValue
                ($os.buildnumber)
            $obj | Add-Member -NotePropertyName Description -NotePropertyValue
                ($os.caption)
            $obj | Add-Member -NotePropertyName ServicePackVersion
                -NotePropertyValue ($os.servicepackmajorversion)
            $obj | Add-Member -NotePropertyName SystemArch -NotePropertyValue
                ($os.OSArchitecture)
```

```

$obj | Add-Member -NotePropertyName ProcessorArch -NotePropertyValue
    ($proc.addresswidth)
$obj | Add-Member -NotePropertyName Status -NotePropertyValue
    ($os.Status)

        Write-Output -InputObject $obj ④
    }
}
}

```

L'ensemble de ces opérations est contenu dans le bloc ① (`parallel`). À l'intérieur, un autre bloc ② (`sequence`) indique dans un ordre précis un certain nombre de messages. Puis suit l'instruction `InlineScript` ③ contenant d'une certaine manière le cœur du code qui nous intéresse : deux requêtes WMI sont réalisées sur une machine distante, et un objet est construit à la volée et affiché à l'écran à l'aide de la cmdlet `Write-Output` ④. Le workflow exécute donc tous ces blocs en parallèle, mais tout en gardant une certaine cohérence dans le déroulement des opérations. Maintenant que le workflow est écrit, nous n'avons plus qu'à l'exécuter (en faisant attention d'avoir les priviléges nécessaires à son lancement) :

```

PS> Get-Infos -PSComputerName 'Lendesk' -Verbose

VERBOSE: [Lendesk]:Processing workflow Get-Infos on -> Lendesk
VERBOSE: [Lendesk]:This is a sample demo workflow
VERBOSE: [Lendesk]:Getting informations about remote machines

BuildNum          : 9200
Description       : Microsoft Windows 8
ServicePackVersion : 0
SystemArch        : 64-bit
ProcessorArch     : 64
Status            : OK
PSComputerName    : Lendesk
PSSourceJobInstanceId : 4f4e7168-afcd-4fe2-b47a-569b8bb2ace7

```

Le résultat met bien en évidence cette cohérence, c'est-à-dire que les messages ont d'abord été affichés dans l'ordre indiqué et que l'objet a été affiché par la suite. Notons la présence du paramètre `-PSComputerName`, faisant partie des paramètres communs à l'ensemble des workflows. En effet, lors de la création d'un workflow, PowerShell ajoute automatiquement des paramètres pouvant être utilisés dans n'importe quel autre workflow.

**ALLER PLUS LOIN À propos des paramètres communs aux workflows**`Get-Help about_WorkflowCommonParameters`

Le workflow suivant illustre comment obtenir (parmi plusieurs possibilités) l'état de services spécifiques tournant sur une machine distante :

```
workflow Get-WFServices {  
    Param([array]$services) ❶  
    foreach -parallel ❷ ($service in $services) {  
        inlinescript ❸ {  
            $svc = Get-Service -ComputerName $PSCo  
            $object = New-Object -typename psobject -property  
                @{Name = $svc.Name ; Status = $svc.Status}  
            write-output -inputobject $object ❹  
        }  
    }  
}
```

Le workflow contient un paramètre ❶ (`-services`) qui peut prendre en charge plusieurs noms de services. Ensuite, l'instruction `foreach -parallel` ❷ parcourt la collection de services passés en paramètres et réalise une requête de manière simultanée pour obtenir le statut de ces services. Les requêtes sont réalisées dans un bloc `inlinescript` ❸ pour se situer dans un environnement PowerShell complet. Comme chacune des requêtes produit un nombre conséquent d'informations, un objet est construit ❹ sur la base de deux propriétés (`Name` et `Status`). Enfin, chaque objet est écrit dans le pipeline ❹. Lançons à présent le workflow en console :

```
PS> Get-WFServices -PSCo  
@('WinRM','WlanSvc','Wsearch','EFS','Browser')  
  
Name : Browser  
Status : Stopped  
PSCo  
PSSourceJobInstanceId : 75998966-e362-44c3-8c80-968653bf1fe6  
  
Name : EFS  
Status : Stopped  
PSCo  
PSSourceJobInstanceId : 75998966-e362-44c3-8c80-968653bf1fe6
```

```
Name          : WlanSvc
Status        : Running
PSComputerName : Lendesk
PSSourceJobInstanceId : 75998966-e362-44c3-8c80-968653bf1fe6

Name          : WinRM
Status        : Running
PSComputerName : Lendesk
PSSourceJobInstanceId : 75998966-e362-44c3-8c80-968653bf1fe6

Name          : Wsearch
Status        : Running
PSComputerName : Lendesk
PSSourceJobInstanceId : 75998966-e362-44c3-8c80-968653bf1fe6
```

Cinq noms de services ont été passés en arguments au workflow à l'aide du paramètre `-services`. Le workflow a donc effectué cinq requêtes en parallèle. La sortie le montre d'ailleurs très bien puisqu'elle est divisée en cinq parties. L'exemple que nous avons pris est basé sur cinq services, mais nous aurions très bien pu nous baser sur plus d'une dizaine de services et sur une centaine de machines distantes, ce qui nous aurait permis de constater le gain de temps que peut apporter l'utilisation des workflows.

# 23

## Les expressions régulières

---

L'écriture de scripts implique souvent de rechercher certaines occurrences de mots ou d'expressions respectant des formats spécifiques dans des fichiers ou autres journaux d'événements. L'utilisateur définit ces formats à l'aide de ce que l'on nomme des expressions régulières. Ces dernières forment un langage spécifique qui structure des motifs (ou patterns) décrivant des ensembles de chaînes de caractères. Elles sont utilisées dans de nombreux domaines (sciences, mathématiques, informatique...).

Dans ce chapitre, qui couvrira les notions de base de l'utilisation des expressions régulières avec PowerShell, nous commencerons par en apprendre la syntaxe. Élément fondamental à la création de motifs, la syntaxe est une condition sine qua non de l'utilisation de l'objet [\[Regex\]](#), qui est un moyen puissant pour profiter des possibilités offertes. Nous étudierons ensuite les opérateurs `-match` et `-notmatch`, natifs et donc intégrés à PowerShell. Enfin, nous nous pencherons sur la cmdlet `Select-String`, qu'on pourrait aisément comparer à l'outil `grep`.

### SOMMAIRE

- ▶ Un peu de syntaxe
- ▶ La classe `[Regex]`
- ▶ Les opérateurs `-match` et `-notmatch`
- ▶ La cmdlet `Select-String`

## Un peu de syntaxe

L'utilisation des expressions régulières requiert de s'approprier la syntaxe nécessaire pour écrire des motifs. Le tableau 23-1 met en évidence la syntaxe de base prise en charge par PowerShell et qui est en même temps la plus employée dans le cadre d'une utilisation quotidienne.

**Tableau 23-1** Syntaxe de base des expressions régulières

Format	Description	Exemple
<code>^</code>	Début de la chaîne de caractères	<code>"jeep" -match "^je"</code>
<code>\$</code>	Fin de la chaîne de caractères	<code>"jeep" -match "p\$"</code>
<code>*</code>	N'importe quelle instance du caractère précédent le symbole	<code>"jeep" -match "e*"</code>
<code>+</code>	Répétitions possibles d'un caractère ou d'une sous-expression	<code>"jeep" -match "je+p"</code>
<code>?</code>	Rien ou une instance du caractère précédent le symbole	<code>"jeep" -match "e?"</code>
<code>\</code>	Interprète littéralement un caractère.	<code>"jeep\$" -match "\\$"</code>
<code>[^]</code>	N'importe quel caractère sauf ceux entre crochets	<code>"jeep" -match "[^abc]"</code>
<code>.</code>	Un caractère	<code>"jeep" -match "je.p"</code>
<code>Valeur</code>	Correspondance exacte avec une valeur	<code>"jeep" -match "ee"</code>
<code>[Valeur]</code>	Correspondance sur au moins un caractère parmi ceux entre crochets	<code>"jeep" -match "[jkl]eep"</code>
<code>[Rangée]</code>	Correspondance sur au moins un caractère parmi une rangée de caractères	<code>"jeep" -match "[c-m]eep"</code>

Les caractères listés dans le tableau ci-dessous appartiennent chacun à une catégorie. En effet, le moteur d'expressions régulières utilisé est celui du framework .NET, dont l'analyse des motifs est basée sur la classification des caractères qui les structurent. Une catégorie importante est appelée *classe de caractères*.

**Tableau 23-2** Classes de caractères utilisées dans les expressions régulières

Format	Description	Exemple
<code>\w</code>	Un caractère (lettre ou chiffre)	<code>"kxvg mlkt" -match "\w"</code>
<code>\W</code>	Un caractère qui n'est ni une lettre, ni un chiffre.	<code>"k\x" -match "\W"</code>
<code>\s</code>	Un caractère espace	<code>"kxvg mlkt" -match "\s"</code>
<code>\S</code>	Un caractère qui n'est pas une espace.	<code>"kxvg mlkt" -match "\S+"</code>
<code>\d</code>	Un chiffre	<code>"9kxsd" -match "\d"</code>
<code>\D</code>	Un caractère qui n'est pas un chiffre.	<code>"9kxsd" -match "\D"</code>

Une autre catégorie importante est celle des *quantificateurs*. Un quantificateur indique combien d'instances d'un élément particulier (caractère, groupe, etc.) doivent être présentes afin qu'une correspondance s'établisse.

**Tableau 23–3** Liste des quantificateurs reconnus par PowerShell

Format	Description	Exemple
*	Zéro, une ou plusieurs instances d'un caractère ou d'une sous-expression	"kxj" -match "\w+"
?	Zéro ou une instance d'un caractère ou d'une sous-expression	"kxj" -match "\w?"
+	Une ou plusieurs instances d'un caractère ou d'une sous-expression	"kxj" -match "\w+"
{n}	Exactement <i>n</i> fois l'élément précédent	"kxtj" -match "\w{2}"
{n,}	Au moins <i>n</i> fois l'élément précédent	"kxtj" -match "\w{2,}"
{n,m}	Entre <i>n</i> et <i>m</i> fois l'élément précédent	"kxtj" -match "\w{2,3}"

Il est fondamental également d'évoquer les *séquences d'échappement*, catégorie qui ne doit pas être confondue avec le caractère d'échappement (`) spécifique à PowerShell. Cette catégorie est basée sur le symbole \ permettant entre autres d'interpréter un caractère de manière littérale, même si ce dernier est considéré comme un caractère spécial.

**Tableau 23–4** Principaux caractères d'échappement reconnus par PowerShell

Format	Description	Modèle
\t	Tabulation	(\w+)\t(\w+)
\r	Retour chariot	\r(\w+)
\n	Saut de ligne	\r\n(\w+)

Évidemment, les caractères que nous venons d'évoquer à travers ces tableaux font partie intégrante du socle de base de la syntaxe liée aux expressions régulières. Plonger dans le monde des expressions régulières exigerait au moins d'écrire un volume, car il s'agit d'une technologie incroyablement complexe. Cependant, une compréhension des bases nous permettra d'aborder de manière sereine leur utilisation avec PowerShell.

## La classe [Regex]

Le framework .NET propose une classe appelée [\[Regex\]](#), qui représente son moteur d'expressions régulières ; il est possible d'utiliser cette classe pour, par exemple, scanner, extraire, modifier, remplacer ou même effacer des données issues de sources différentes.

Commençons par créer l'objet [\[Regex\]](#) :

```
PS> $regexobject = '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'
```

Ici, l'objet est instancié d'une manière particulière. Nous pouvons remarquer que ce qui précède le nom de la variable `$regexobject` est un alias de type. En l'occurrence, l'alias est [\[Regex\]](#) et le type est [\[System.Text.RegularExpressions.Regex\]](#). De plus, la valeur affectée à l'objet `$regexobject` est un motif décrivant le format d'une adresse IP. À présent, listons les membres de cet objet :

```
PS> $regexobject | Get-Member
```

```
TypeName: System.Text.RegularExpressions.Regex
```

Name	MemberType	Definition
-----	-----	-----
Equals	Method	bool Equals(System.Object obj)
GetGroupNames	Method	string[] GetGroupNames()
GetGroupNumbers	Method	int[] GetGroupNumbers()
GetHashCode	Method	int GetHashCode()
GetObjectData	Method	void ISerializable.GetObjectDa
GetType	Method	type GetType()
GroupNameFromNumber	Method	string GroupNameFromNumber(int
GroupNumberFromName	Method	int GroupNumberFromName(string
IsMatch	Method	bool IsMatch(string input), bo
Match	Method	System.Text.RegularExpressions
Matches	Method	System.Text.RegularExpressions
Replace	Method	string Replace(string input, s
Split	Method	string[] Split(string input),
ToString	Method	string ToString()
MatchTimeout	Property	timespan MatchTimeout {get;}
Options	Property	System.Text.RegularExpressions
RightToLeft	Property	bool RightToLeft {get;}

Les méthodes proposées par la classe [\[Regex\]](#) illustrent la description de cette classe faite en début de section. Affichons maintenant le contenu d'un fichier listant des noms de serveurs avec les adresses IP correspondantes :

```
PS> Get-Content -Path .\ip_list.txt
```

```
srv2008 192.168.1.17
srv2008R2 192.168.1.22
srv2012 192.168.1.36
srv2003 192.168.1.23
srv2003R2 192.168.1.27
```

Vous l'aurez sans doute deviné, notre objectif est de scanner ce fichier pour en extraire les adresses IP sans les noms de serveurs respectifs :

```
PS> Get-Content -Path .\ip_list.txt | ForEach-Object {$regexobject.Match($_)}
```

```
Groups : {192.168.1.17}
Success : True
Captures : {192.168.1.17}
Index : 8
Length : 12
Value : 192.168.1.17
```

```
Groups : {192.168.1.22}
Success : True
Captures : {192.168.1.22}
Index : 10
Length : 12
Value : 192.168.1.22
```

```
Groups : {192.168.1.36}
Success : True
Captures : {192.168.1.36}
Index : 8
Length : 12
Value : 192.168.1.36
```

```
Groups : {192.168.1.23}
Success : True
Captures : {192.168.1.23}
Index : 8
Length : 12
Value : 192.168.1.23
```

```
Groups : {192.168.1.27}
Success : True
Captures : {192.168.1.27}
Index : 10
Length : 12
Value : 192.168.1.27
```

La méthode utilisée ici est `Match()` ; elle trouve les occurrences que nous cherchons et qui ont comme format le motif spécifié lors de la construction de l'objet. Cependant, la sortie est un peu trop verbeuse et nous sommes intéressés uniquement par les adresses IP. Donc, pour modifier le résultat, seule la propriété `Value` sera retenue :

```
PS> Get-Content -Path .\ip_list.txt | ForEach-Object  
{$regexobject.Match($_).Value}
```

```
192.168.1.17  
192.168.1.22  
192.168.1.36  
192.168.1.23  
192.168.1.27
```

Et voilà ! Nous avons réussi à dresser une liste d'adresses IP à partir d'un fichier texte. Occupons-nous maintenant des noms de serveurs. On peut remarquer que ces derniers commencent tous par les lettres '`srv`'. Ceci n'est pas gênant a priori, mais pour des raisons qui sont liées à une certaine homogénéisation, il faudrait que ces lettres soient représentées non pas en minuscules, mais en majuscules. Deux possibilités s'offrent à nous : faire les modifications à la main (à condition que la liste ne contienne pas des milliers de noms) ou automatiser le processus, ce qui ne prendra que très peu de temps :

```
PS> Get-Content -Path .\ip_list.txt | ForEach-Object  
{[regex]::Replace($_,'srv','SRV')}
```

```
SRV2008 192.168.1.17  
SRV2008R2 192.168.1.22  
SRV2012 192.168.1.36  
SRV2003 192.168.1.23  
SRV2003R2 192.168.1.27
```

Pour remplacer une occurrence par une autre, nous avons utilisé `replace()`, méthode statique et à laquelle nous avons passé trois arguments. Le premier argument est l'entrée, c'est-à-dire la ligne actuellement scannée dans le pipeline. Le deuxième argument est l'occurrence à remplacer. Enfin, le troisième argument est l'occurrence remplaçante. Là aussi, l'utilisation des expressions régulières accroît considérablement la productivité en termes de scripting.

## Les opérateurs `-match` et `-notmatch`

PowerShell donne aussi la possibilité d'utiliser les expressions régulières à l'aide des opérateurs `-match` et `-notmatch`. Toutefois, les perspectives que ces opérateurs offrent

ne sont pas les mêmes que pour la classe [Regex]. Le principe est simple : spécifier une entrée à gauche de l'opérateur et un motif à sa droite. PowerShell affiche la valeur \$True si une correspondance est trouvée, \$False dans le cas contraire :

```
PS> 'Windows PowerShell' -match 'She11'  
True
```

L'entrée que nous avons passée à l'opérateur `-match` est `Windows PowerShell` et le motif est `She11`. Comme une correspondance est établie, PowerShell renvoie la valeur `$True`, qu'il stocke dans la variable `$Matches` de manière transparente. Reprenons l'exemple précédent en le modifiant quelque peu :

```
PS> 'Windows PowerShell' -match 'Sh\w+'  
True
```

Ici, le motif a changé ; il est composé des lettres `Sh`, de la classe `\w` et du quantificateur `+`. À présent, observons le contenu de la variable `$Matches` :

```
PS> $Matches  


| Name | Value |
|------|-------|
| 0    | Shell |


```

La sortie affichée s'articule autour d'une paire clé-valeur, parce que cette variable n'est rien d'autre qu'un dictionnaire :

```
PS> $Matches | gm  
  
TypeName: System.Collections.Hashtable  
  


| Name          | MemberType | Definition                     |
|---------------|------------|--------------------------------|
| Add           | Method     | void Add(System.Object key, Sy |
| Clear         | Method     | void Clear(), void IDictionary |
| Clone         | Method     | System.Object Clone(), System. |
| Contains      | Method     | bool Contains(System.Object ke |
| ContainsKey   | Method     | bool ContainsKey(System.Object |
| ContainsValue | Method     | bool ContainsValue(System.Obje |
| CopyTo        | Method     | void CopyTo(array array, int a |
| Equals        | Method     | bool Equals(System.Object obj) |
| GetEnumerator | Method     | System.Collections.IDictionary |
| GetHashCode   | Method     | int GetHashCode()              |
| GetObjectData | Method     | void GetObjectData(System.Runt |
| GetType       | Method     | type GetType()                 |


```

OnDeserialization	Method	void OnDeserialization(System.
Remove	Method	void Remove(System.Object key)
ToString	Method	string ToString()
Item	ParameterizedProperty	System.Object Item(System.Object
Count	Property	int Count {get;}
IsFixedSize	Property	bool IsFixedSize {get;}
IsReadOnly	Property	bool IsReadOnly {get;}
IsSynchronized	Property	bool IsSynchronized {get;}
Keys	Property	System.Collections.ICollection
SyncRoot	Property	System.Object SyncRoot {get;}
Values	Property	System.Collections.ICollection

L'opérateur `-match` accepte aussi des collections de valeurs :

```
PS> 'Bash', 'Perl', 'Python' -match '^\\w{4}$'
Bash
Perl
```

Dans ce cas de figure, le contenu de la variable `$Matches` n'est pas incrémenté. L'opérateur `-notmatch` renvoie quant à lui une valeur `$True` lorsqu'il n'y a *pas* de correspondances :

```
PS> 'Perl' -notmatch 'Shell'
True
PS> 'Perl' -notmatch '\\w*r1'
False
PS> 'Windows PowerShell' -notmatch '(\\w+)\\s(\\w+)'
False
```

#### NOTE Concernant les opérateurs `-match` et `-notmatch`

Les opérateurs `-match` et `-notmatch` se concentrent uniquement sur les chaînes et non pas sur les autres types comme les tableaux d'objets.

## La cmdlet `Select-String`

En plus des opérateurs `-match` et `-notmatch` proposés, il existe aussi une autre alternative prenant la forme d'une cmdlet : `Select-String`. Elle est l'équivalent de l'outil grep dans le monde Unix/Linux, ou même de findstr dans le monde Windows. Elle sert à rechercher et à trouver des occurrences de textes ou de motifs à partir de sources diverses (fichiers texte, chaînes, etc.). En outre, les recherches peuvent se baser sur une ou plusieurs occurrence(s). Cela nous amène à constater qu'il y a dans cette cmdlet des caractéristiques venant des opérateurs `-match` et `-notmatch`, mais aussi de l'objet [\[Regex\]](#).

Pour mieux connaître cette cmdlet, voici un exemple simple :

```
PS> 'POWERSHELL','powershell','shell' | Select-String -Pattern 'powershell'  
POWERSHELL  
powershell
```

Trois chaînes ont été envoyées à la cmdlet `Select-String`. Cette dernière effectue une recherche sur chacune pour trouver une occurrence de texte `powershell` spécifiée à l'aide du paramètre `-pattern`. La sortie montre que deux occurrences ont été trouvées ; il s'agit du même mot, une fois en majuscules et l'autre en minuscules. Pour que la recherche soit sensible à la casse, il faut utiliser le paramètre `-casesensitive` :

```
PS> 'POWERSHELL','powershell','shell' | Select-String -Pattern  
'powershell' -Casesensitive  
powershell
```

Essayons à présent de trouver des occurrences à partir de fichiers, par exemple toutes les occurrences du mot `operators` dans les fichiers d'aide PowerShell :

```
PS> select-string -path $pshome\en-US\*.txt -pattern 'operators'  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:2: about_Arithmetic_Operators  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:5: Describes the operators  
that perform arithmetic in Windows PowerShell.  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:9: Arithmetic operators  
calculate numeric values. You can use one or  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:10: more arithmetic operators  
to add, subtract, multiply, and divide  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:20: Beginning in Windows  
PowerShell 2.0, all arithmetic operators work  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:29: Windows PowerShell  
supports the following arithmetic operators:  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:61: Windows PowerShell  
processes arithmetic operators in the following order:  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:153: multiplication operators  
in operations that include different object types:  
C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about_Arithmetic_Operators.help.txt:289: Although the addition
```

```
operators are very useful, use the assignment
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:290:    operators to add elements
to hash tables and arrays. For more information
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:291:    see
about_assignment_operators. The following examples use the +=
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:377:    ARITHMETIC OPERATORS AND
VARIABLES
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:378:    You can also use
arithmetic operators with variables. The operators act on
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:380:    arithmetic operators with
variables:
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:397:    ARITHMETIC OPERATORS AND
COMMANDS
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:398:    Typically, you use the
arithmetic operators in expressions with numbers,
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:399:    strings, and arrays.
However, you can also use arithmetic operators with
C:\Windows\System32\WindowsPowerShell\v1.0\en-
US\about_Arithmetic_Operators.help.txt:402:    The following examples
show how to use the arithmetic operators in
C:\Windows\System32\WindowsPowerShell\v1.0\en-
(...)
```

Comme la sortie est longue, nous ne retiendrons que le premier extrait, suffisant dans le cadre de notre démonstration. En l'observant attentivement, nous pouvons distinguer pour chaque occurrence :

- le fichier dans lequel cette occurrence apparaît ;
- la ligne où se trouve l'occurrence ;
- le texte contenant cette occurrence.

Si nous voulons non pas une occurrence par ligne mais plusieurs, le paramètre `-AllMatches` doit être invoqué :

```
PS> select-string -path $pshome\en-US\*.txt -pattern "operators" -AllMatches
```

La cmdlet `Select-String` est très performante, surtout dans le cadre de recherches massives liées à des fichiers volumineux. Son utilisation deviendra au fur et à mesure presque automatique, tant la recherche d'occurrences sur la base d'un ou de plusieurs fichier(s) est une opération récurrente dans le monde de l'administration système/réseau.

# 24

## Automatiser l'Active Directory

---

*Dans le contexte d'une infrastructure, il est important de concentrer sa gestion et son administration. De cette façon, les éléments constituant cette infrastructure pourront être répertoriés et gérés de manière tout à fait centralisée. Cette conception de l'administration est mise en œuvre par l'Active Directory, service d'annuaire utilisé par les entreprises pour stocker et centraliser des informations relatives aux ressources d'une infrastructure donnée. Active Directory est utilisé depuis la version 2000 de Windows Server. Depuis, cette technologie s'est sans cesse améliorée.*

*Le lien avec PowerShell est évident : automatiser l'administration de l'Active Directory, afin d'augmenter notamment l'efficacité des administrateurs. Un chapitre ne suffira pas pour énumérer les possibilités offertes par PowerShell dans ce domaine. Toutefois, nous essaierons de comprendre ce qu'est l'Active Directory pour mieux le situer dans la perspective qui est la nôtre. Nous verrons aussi comment gérer les utilisateurs et les groupes. Enfin, nous apprendrons à créer un domaine auquel sera rattachée une zone de recherche directe ainsi qu'une zone de recherche inversée.*

### SOMMAIRE

- ▶ Comprendre ce qu'est l'Active Directory
- ▶ Les utilisateurs et groupes Active Directory
- ▶ Créer un domaine Active Directory
- ▶ Créer une zone DNS de recherche directe et une zone DNS de recherche inversée

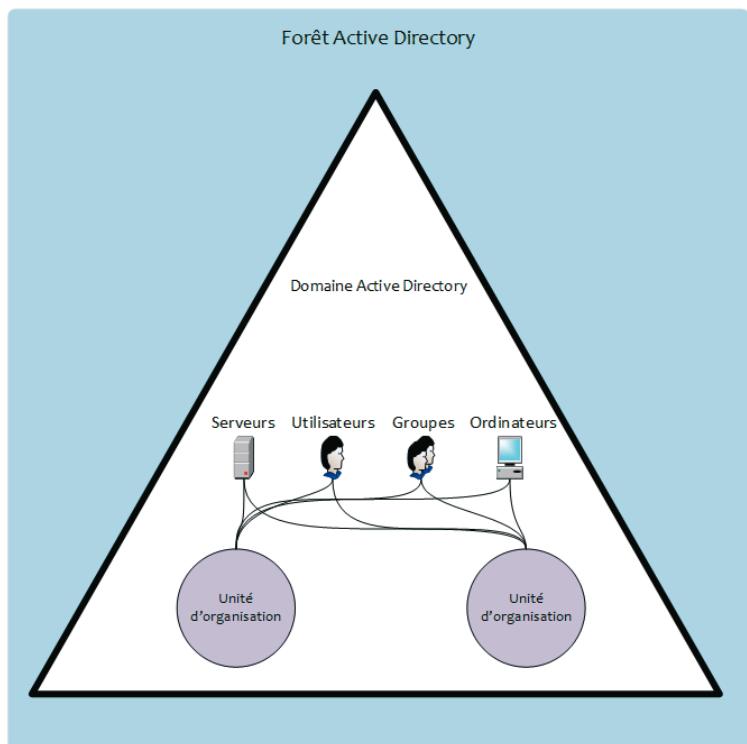
## Comprendre ce qu'est l'Active Directory

Avant d'opérer une quelconque automatisation, il est très important de définir de manière synthétique ce qu'est l'Active Directory.

### Qu'est-ce que l'Active Directory ?

Active Directory est un service d'annuaire centralisant un certain nombre de ressources existant au sein d'une infrastructure : utilisateurs, groupes, ordinateurs ou même les imprimantes. L'organisation de cette gestion centralisée est la fonction essentielle de l'Active Directory. L'ensemble de ces éléments administrables nécessitent d'être répertoriés de manière adéquate. Pour cela, l'Active Directory stocke ses informations dans une base de données centralisée dont la taille peut varier en fonction de la dimension de l'infrastructure. Le schéma suivant illustre comment s'articulent les éléments d'une infrastructure Active Directory.

**Figure 24-1**  
Mode de fonctionnement  
de l'Active Directory



Ce schéma regroupe des notions essentielles au fonctionnement d'une infrastructure Active Directory.

- **La forêt Active Directory**

Une forêt est une hiérarchie d'une ou plusieurs arborescence(s) de domaines.

- **Le domaine Active Directory**

Un domaine est une organisation logique contenant un certain nombre d'objets, qui peuvent entre autres être des utilisateurs, des ordinateurs, des imprimantes ou des documents.

- **Unité d'organisation**

Une unité d'organisation est un conteneur utilisé pour organiser des objets Active Directory. Les unités d'organisation facilitent l'administration des ressources et sa possible délégation.

Ces notions sont fondamentales pour tout administrateur d'une infrastructure Active Directory.

## Le module Active Directory

Depuis la version 2008 R2 de Windows Server, un module Active Directory est disponible nativement pour les administrateurs qui veulent automatiser les tâches d'administration. Il faut cependant souligner que l'administration de l'Active Directory était possible avant la version 2008 R2. En effet, les utilisateurs pouvaient (et peuvent toujours) utiliser l'ensemble des interfaces COM Active Directory (ADSI). Aujourd'hui, Microsoft encourage les utilisateurs de PowerShell à utiliser principalement le module Active Directory plutôt que l'ADSI.

Pour utiliser le module Active Directory, il faut l'importer (étape non nécessaire avec Windows Server 2012) :

```
PS> Import-Module -Name ActiveDirectory
```

Le module importé, listons les cmdlets à notre disposition :

```
PS> Get-Command -Module ActiveDirectory | Select-Object -Property Name
```

Name
---
Add-ADComputerServiceAccount
Add-ADDomainControllerPasswordReplicationPolicy
Add-ADFineGrainedPasswordPolicySubject
Add-ADGroupMember
Add-ADPrincipalGroupMembership

```
Clear-ADAccountExpiration
Disable-ADAccount
Disable-ADOptionalFeature
Enable-ADAccount
Enable-ADOptionalFeature
Get-ADAccountAuthorizationGroup
Get-ADAccountResultantPasswordReplicationPolicy
Get-ADComputer
Get-ADComputerServiceAccount
Get-ADDefaultDomainPasswordPolicy
Get-ADDomain
Get-ADDomainController
Get-ADDomainControllerPasswordReplicationPolicy
Get-ADDomainControllerPasswordReplicationPolicyUsage
Get-ADFineGrainedPasswordPolicy
Get-ADFineGrainedPasswordPolicySubject
Get-ADForest
Get-ADGroup
Get-ADGroupMember
Get-ADObject
Get-ADOptionalFeature
Get-ADOrganizationalUnit
Get-ADPrincipalGroupMembership
Get-ADRootDSE
Get-ADServiceAccount
Get-ADUser
Get-ADUserResultantPasswordPolicy
Install-ADServiceAccount
Move-ADDirectoryServer
Move-ADDirectoryServerOperationMasterRole
Move-ADObject
New-ADComputer
New-ADFineGrainedPasswordPolicy
New-ADGroup
New-ADObject
New-ADOrganizationalUnit
New-ADServiceAccount
New-ADUser
Remove-ADComputer
Remove-ADComputerServiceAccount
Remove-ADDomainControllerPasswordReplicationPolicy
Remove-ADFineGrainedPasswordPolicy
Remove-ADFineGrainedPasswordPolicySubject
Remove-ADGroup
Remove-ADGroupMember
Remove-ADObject
Remove-ADOrganizationalUnit
Remove-ADPrincipalGroupMembership
Remove-ADServiceAccount
Remove-ADUser
```

```
Rename-ADObject
Reset-ADServiceAccountPassword
Restore-ADObject
Search-ADAccount
Set-ADAccountControl
Set-ADAccountExpiration
Set-ADAccountPassword
Set-ADComputer
Set-ADDefaultDomainPasswordPolicy
Set-ADDomain
Set-ADDomainMode
Set-ADFineGrainedPasswordPolicy
Set-ADForest
Set-ADForestMode
Set-ADGroup
Set-ADObject
Set-ADOrganizationalUnit
Set-ADServiceAccount
Set-ADUser
Uninstall-ADServiceAccount
Unlock-ADAccount
```

De nombreuses cmdlets sont disponibles pour aider les administrateurs dans l'automatisation de leurs tâches d'administration. L'importation du module Active Directory entraîne aussi l'apparition d'un fournisseur qui lui est propre et qui donne accès à ses données comme dans une arborescence de système de fichiers. Grâce à cette perspective, il est possible de se connecter à plusieurs instances Active Directory appartenant à des domaines différents. Pour utiliser le fournisseur Active Directory, il faut se déplacer au niveau du lecteur qui lui est associé :

```
PS> cd AD:
PS AD:\>
```

À présent, listons les objets à notre disposition :

```
PS AD:\> ls
```

Name	ObjectClass	DistinguishedName
scripting	domainDNS	DC=scripting,DC=net
Configuration	configuration	CN=Configuration,DC=scripting,DC=net
Schema	dMD	CN=Schema,CN=Configuration,DC=scri...
DomainDnsZones	domainDNS	DC=DomainDnsZones,DC=scripting,DC=net
ForestDnsZones	domainDNS	DC=ForestDnsZones,DC=scripting,DC=net

Nous naviguons comme si nous nous trouvions dans un système de fichiers :

```
PS AD:\> ls

PS AD:\> cd '.\DC=scripting,DC=net'
PS AD:\DC=scripting,DC=net> ls

Name          ObjectClass      DistinguishedName
----          -----
Builtin        builtinDomain   CN=Builtin,DC=scripting,DC=ne
Computers      container       CN=Computers,DC=scripting,DC=
Domain Controllers  organizationalUnit  OU=Domain Controllers,DC=scri
ForeignSecurityPr...  container   CN=ForeignSecurityPrincipals,
Infrastructure    infrastructureUpdate  CN=Infrastructure,DC=scriptin
LostAndFound    lostAndFound   CN=LostAndFound,DC=scripting,
Managed Service A...  container   CN=Managed Service Accounts,D
NTDS Quotas     msDS-QuotaContainer  CN=NTDS Quotas,DC=scripting,D
Program Data    container       CN=Program Data,DC=scripting,
System          container       CN=System,DC=scripting,DC=net
Users           container       CN=Users,DC=scripting,DC=net

PS AD:\DC=scripting,DC=net> cd '.\OU=Domain Controllers'
PS AD:\OU=Domain Controllers,DC=scripting,DC=net> ls

Name          ObjectClass      DistinguishedName
----          -----
DC2008R2      computer        CN=DC2008R2,OU=Domain Control

PS AD:\OU=Domain Controllers,DC=scripting,DC=net>
```

L'utilisation conjointe du fournisseur et des cmdlets Active Directory accroît sans conteste l'efficacité de l'administration au quotidien. Cependant, une utilisation optimale de ce module requiert un certain temps avant d'atteindre cette efficacité.

## Les utilisateurs et groupes Active Directory

Gérer les comptes d'utilisateurs ainsi que les groupes est sans doute l'opération la plus partagée et la plus universelle en ce qui concerne l'administration de l'Active Directory. Cette gestion quotidienne implique un phénomène de répétition qu'il est bon d'automatiser. Pour cela, le module Active Directory propose tout un ensemble de cmdlets.

## Administre les comptes d'utilisateurs

Avant d'accéder à n'importe quelle ressource du domaine, un utilisateur doit pouvoir exister au sein de ce domaine. Ensuite, les administrateurs du domaine en question doivent être capables de collecter, modifier ou ajouter des informations relatives à ce compte d'utilisateur.

Tout d'abord, commençons par créer un utilisateur Active Directory. La cmdlet que nous utiliserons se nomme `New-ADUser` :

```
PS> New-ADUser Nawphel -OtherAttributes  
@{title="director";mail="NawphelA@contoso.com"}
```

Le nom de l'utilisateur est '`Nawphel`'. De plus, comme la cmdlet crée un objet utilisateur à la volée, le paramètre `-OtherAttributes` a été appelé afin de spécifier certaines propriétés. À présent, vérifions à l'aide de la cmdlet `Get-ADUser` si l'objet a bien été créé dans l'Active Directory :

```
PS> Get-ADUser -Filter {name -eq 'Nawphel'}  
  
DistinguishedName : CN=Nawphel,CN=Users,DC=contoso,DC=com  
Enabled : False  
GivenName :  
Name : Nawphel  
ObjectClass : user  
ObjectGUID : 637b5806-00da-452f-8611-89b2d8f6f097  
SamAccountName : Nawphel  
SID : S-1-5-21-2153971331-1430186003-2770964410-1230  
Surname :  
UserPrincipalName :
```

Le compte d'utilisateur a bien été créé. Le paramètre `-Filter` est très important : il s'agit en réalité d'un puissant filtre LDAP qui possède d'autres fonctionnalités, comme la prise en charge des variables PowerShell. Il faut aussi ajouter que ce paramètre est utilisé par toutes les cmdlets Active Directory composées avec le verbe `Get`.

Les propriétés affichées par défaut sont peu nombreuses par rapport au nombre de propriétés de l'objet. Pour toutes les afficher, il faut utiliser le paramètre `-Properties` :

```
PS> Get-ADUser -Filter {name -eq 'Nawphel'} -Properties *  
  
AccountExpirationDate :  
accountExpires : 9223372036854775807  
AccountLockoutTime :  
AccountNotDelegated : False  
AllowReversiblePasswordEncryption : False
```

```
BadLogonCount          : 0
badPasswordTime        : 0
badPwdCount            : 0
CannotChangePassword   : False
CanonicalName          : contoso.com/Users/Nawphel
Certificates           : {}
City                  :
CN                   : Nawphel
codePage              : 0
Company               :
Country               :
countryCode           : 0
Created               : 19/07/2013 14:17:00
createTimeStamp        : 19/07/2013 14:17:00
Deleted               :
Department            :
Description            :
DisplayName            :
DistinguishedName     : CN=Nawphel,CN=Users,DC=contoso,DC=com
Division               :
DoesNotRequirePreAuth : False
dSCorePropagationData : {31/12/1600 16:00:00}
EmailAddress           : NawphelA@contoso.com
EmployeeID             :
EmployeeNumber         :
Enabled                : False
Fax                   :
GivenName              :
HomeDirectory          :
HomedirRequired        : False
HomeDrive              :
HomePage               :
HomePhone              :
Initials               :
instanceType           : 4
isDeleted              :
LastBadPasswordAttempt :
LastKnownParent        :
lastLogoff              : 0
lastLogon               : 0
LastLogonDate          :
LockedOut              : False
logonCount              : 0
LogonWorkstations       :
mail                  : NawphelA@contoso.com
Manager                :
MemberOf               : {}
MNSLogonAccount        : False
MobilePhone             :
Modified               : 19/07/2013 14:17:00
```

```
modifyTimeStamp : 19/07/2013 14:17:00
msDS-User-Account-Control-Computed : 8388608
Name : Nawphel
nTSecurityDescriptor :
System.DirectoryServices.ActiveDirectorySecuri
ObjectCategory :
CN=Person,CN=Schema,CN=Configuration,DC=contos
ObjectClass : user
ObjectGUID : 637b5806-00da-452f-8611-89b2d8f6f097
objectSid : S-1-5-21-2153971331-1430186003-2770964410-1230
Office :
OfficePhone :
Organization :
OtherName :
PasswordExpired : True
PasswordLastSet :
PasswordNeverExpires : False
PasswordNotRequired : False
POBox :
PostalCode :
PrimaryGroup : CN=Domain Users,CN=Users,DC=contoso,DC=com
primaryGroupID : 513
ProfilePath :
ProtectedFromAccidentalDeletion : False
pwdLastSet : 0
SamAccountName : Nawphel
sAMAccountType : 805306368
ScriptPath :
sDRightsEffective : 15
ServicePrincipalNames : {}
SID : S-1-5-21-2153971331-1430186003-2770964410-1230
SIDHistory : {}
SmartcardLogonRequired : False
State :
StreetAddress :
Surname :
Title : director
TrustedForDelegation : False
TrustedToAuthForDelegation : False
UseDESKeyOnly : False
userAccountControl : 514
userCertificate : {}
UserPrincipalName :
uSNChanged : 238238
uSNCreated : 238237
whenChanged : 19/07/2013 14:17:00
whenCreated : 19/07/2013 14:17:00
```

La liste des propriétés est longue, mais nous avons toutes les informations nécessaires concernant l'utilisateur en question : son compte est créé, mais il n'est pas actif. En effet, la propriété `Enabled` a comme valeur `False`. Il faut donc l'activer :

```
PS> Enable-ADAccount -Identity 'Nawphel' -PassThru
```

```
DistinguishedName : CN=Nawphel,CN=Users,DC=contoso,DC=com
Enabled          : True
Name             : Nawphel
ObjectClass      : user
ObjectGUID       : 637b5806-00da-452f-8611-89b2d8f6f097
SamAccountName   : Nawphel
SID              : S-1-5-21-2153971331-1430186003-2770964410-1230
UserPrincipalName :
```

Il manque encore quelque chose : le mot de passe. La cmdlet qui nous aidera à accomplir cette tâche est `Set-ADAccountPassword` :

```
PS> Set-ADAccountPassword -Identity 'Nawphel'
```

```
Please enter the current password for 'CN=Nawphel,CN=Users,DC=contoso,DC=com'  
Password:  
Please enter the desired password for 'CN=Nawphel,CN=Users,DC=contoso,DC=com'  
Password: *****  
Repeat Password: *****
```

Un domaine Active Directory contient des unités d'organisation. Il est fondamental de les connaître pour mieux cerner le domaine (`contoso.com`) que nous ciblons :

```
PS> Get-ADOrganizationalUnit -Filter *
```

City	:
Country	:
DistinguishedName	: OU=Domain Controllers,DC=contoso,DC=com
LinkedGroupPolicyObjects	: {CN={6AC1786C-016F-11D2-945F-00C04fB984F9}, CN=Policies,
ManagedBy	:
Name	: Domain Controllers
ObjectClass	: organizationalUnit
ObjectGUID	: a7e78dcf-0e5e-43cc-9fdf-b839d6e4829e
PostalCode	:
State	:
StreetAddress	:
City	:
Country	:
DistinguishedName	: OU=Accounts,DC=contoso,DC=com

```
LinkedGroupPolicyObjects : {}
ManagedBy           :
Name                : Accounts
ObjectClass         : organizationalUnit
ObjectGUID          : f244ced0-3762-4645-8937-62cf4b68aecf
PostalCode          :
State               :
StreetAddress        :

City                :
Country             :
DistinguishedName   : OU=Executives,OU=Accounts,DC=contoso,DC=com
LinkedGroupPolicyObjects : {}
ManagedBy           :
Name                : Executives
ObjectClass         : organizationalUnit
ObjectGUID          : e70702b5-6744-4ca5-833e-a6396f76c3a8
PostalCode          :
State               :
StreetAddress        :

City                :
Country             :
DistinguishedName   : OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
LinkedGroupPolicyObjects : {}
ManagedBy           :
Name                : Mailboxes
ObjectClass         : organizationalUnit
ObjectGUID          : f5a8dc63-1441-479b-91f1-2dcf93ec9067
PostalCode          :
State               :
StreetAddress        :

City                :
Country             :
DistinguishedName   : OU=Legal,OU=Accounts,DC=contoso,DC=com
LinkedGroupPolicyObjects : {}
ManagedBy           :
Name                : Legal
ObjectClass         : organizationalUnit
ObjectGUID          : ae0e90f0-f32c-4119-8f12-bd42998ad1a8
PostalCode          :
State               :
StreetAddress        :

City                :
Country             :
DistinguishedName   : OU=IT,OU=Accounts,DC=contoso,DC=com
LinkedGroupPolicyObjects : {}
ManagedBy           :
```

```
Name          : IT
ObjectClass   : organizationalUnit
ObjectGUID    : 02a47b7c-02b6-4572-8e85-15f0aa3c4092
PostalCode    :
State         :
StreetAddress  :

City          :
Country        :
DistinguishedName : OU=Microsoft Exchange Security
Groups,DC=contoso,DC=com
LinkedGroupPolicyObjects : {}
ManagedBy      :
Name           : Microsoft Exchange Security Groups
ObjectClass    : organizationalUnit
ObjectGUID     : 14e32b72-22ab-4f94-8f2c-e037edb6807a
PostalCode    :
State         :
StreetAddress  :

City          :
Country        :
DistinguishedName : OU=RTC Special Accounts,DC=contoso,DC=com
LinkedGroupPolicyObjects : {}
ManagedBy      :
Name           : RTC Special Accounts
ObjectClass    : organizationalUnit
ObjectGUID     : 661a5455-9308-4d26-970e-1135f53d869f
PostalCode    :
State         :
StreetAddress  :
```

La sortie est longue. Essayons d'appliquer un filtre pour mieux nous y retrouver :

```
PS> Get-ADOrganizationalUnit -Filter * | select name
```

```
name
-----
Domain Controllers
Accounts
Executives
Mailboxes
Legal
IT
Microsoft Exchange Security Groups
RTC Special Accounts
```

Le compte d'utilisateur '**Nawphe1**' doit être déplacé dans l'unité d'organisation '**IT**'. Avant d'opérer ce déplacement, listons les membres de cette unité d'organisation :

```
PS> Get-ADUser -Filter * -SearchBase"OU=IT,OU=Accounts,DC=contoso,DC=com"

DistinguishedName : CN=Karen Berg,OU=IT,OU=Accounts,DC=contoso,DC=com
Enabled          : True
GivenName        : Karen
Name             : Karen Berg
ObjectClass      : user
ObjectGUID       : 82fc1f52-1f6b-465d-8e8d-7f67db9de360
SamAccountName   : KarenB
SID              : S-1-5-21-2153971331-1430186003-2770964410-1160
Surname          : Berg
UserPrincipalName : KarenB@contoso.com

DistinguishedName : CN=Kim Akers,OU=IT,OU=Accounts,DC=contoso,DC=com
Enabled          : True
GivenName        : Kim
Name             : Kim Akers
ObjectClass      : user
ObjectGUID       : 2c9202db-eb0c-4885-9340-054906e38862
SamAccountName   : KimA
SID              : S-1-5-21-2153971331-1430186003-2770964410-1163
Surname          : Akers
UserPrincipalName : KimA@contoso.com

DistinguishedName : CN=Paul Duffy,OU=IT,OU=Accounts,DC=contoso,DC=com
Enabled          : True
GivenName        : Paul
Name             : Paul Duffy
ObjectClass      : user
ObjectGUID       : 7fa0a057-96fe-4d83-8e10-b7440066be03
SamAccountName   : PaulD
SID              : S-1-5-21-2153971331-1430186003-2770964410-1174
Surname          : Duffy
UserPrincipalName : PaulD@contoso.com

DistinguishedName : CN=Erik Ryan,OU=IT,OU=Accounts,DC=contoso,DC=com
Enabled          : True
GivenName        : Erik
Name             : Erik Ryan
ObjectClass      : user
ObjectGUID       : 36e4b2e9-e5f6-462f-abc5-130047d02545
SamAccountName   : ErikR
SID              : S-1-5-21-2153971331-1430186003-2770964410-1214
Surname          : Ryan
UserPrincipalName : ErikR@contoso.com
```

Dans notre exemple, il y a donc au total quatre comptes d'utilisateurs dans cette unité d'organisation à laquelle nous voulons ajouter l'utilisateur '**Nawphel**' :

```
PS> Move-ADObject -Identity 'CN=Nawphel,CN=Users,DC=contoso,DC=com' -TargetPath  
'OU=IT,OU=Accounts,DC=contoso,DC=com'
```

La cmdlet **Move-ADObject** que nous avons utilisée pour opérer ce déplacement ne produit pas de sortie. Une des alternatives pour vérifier que le déplacement a bien eu lieu est de passer de nouveau par **Get-ADUser** :

```
PS> Get-ADUser -Filter {name -eq 'Nawphel'} -SearchBase  
"OU=IT,OU=Accounts,DC=contoso,DC=com"
```

```
DistinguishedName : CN=Nawphel,OU=IT,OU=Accounts,DC=contoso,DC=com  
Enabled : True  
GivenName :  
Name : Nawphel  
ObjectClass : user  
ObjectGUID : 637b5806-00da-452f-8611-89b2d8f6f097  
SamAccountName : Nawphel  
SID : S-1-5-21-2153971331-1430186003-2770964410-1230  
Surname :  
UserPrincipalName :
```

Le compte d'utilisateur a bien été ajouté à l'unité d'organisation **IT**. À travers toutes ces opérations, on constate que la gestion des comptes d'utilisateurs Active Directory via PowerShell consiste en une utilisation conjointe de plusieurs cmdlets, accomplissant chacune une tâche spécifique. Ce phénomène est le même pour toutes les composantes d'administration de l'Active Directory, comme la gestion des groupes.

## Administre les groupes

Les groupes Active Directory représentent un formidable moyen de gérer les ressources d'un domaine. En général, un groupe contient des utilisateurs, mais aussi des ordinateurs ou alors d'autres groupes. Les administrateurs Active Directory savent très bien que définir un groupe n'est pas aussi simple que cela en a l'air. Cette définition doit en effet tenir compte de l'architecture de l'infrastructure Active Directory.

C'est la cmdlet **Get-ADGroup** qui liste les groupes Active Directory :

```
PS> Get-ADGroup -Filter * | select name  
  
name  
----  
Administrators
```

```
Users
Guests
Print Operators
Backup Operators
Replicator
Remote Desktop Users
Network Configuration Operators
Performance Monitor Users
Performance Log Users
Distributed COM Users
IIS_IUSRS
Cryptographic Operators
Event Log Readers
Certificate Service DCOM Access
Domain Computers
Domain Controllers
Schema Admins
Enterprise Admins
Cert Publishers
Domain Admins
Domain Users
(...)
```

Dans la section précédente, nous avons créé un compte d'utilisateur. Cet utilisateur appartient forcément par défaut à un groupe :

```
PS> Get-ADUser -Filter {name -eq 'Nawphel'} -Properties PrimaryGroup

DistinguishedName : CN=Nawphel,OU=IT,OU=Accounts,DC=contoso,DC=com
Enabled          : True
GivenName        :
Name             : Nawphel
ObjectClass      : user
ObjectGUID       : 637b5806-00da-452f-8611-89b2d8f6f097
PrimaryGroup     : CN=Domain Users,CN=Users,DC=contoso,DC=com
SamAccountName   : Nawphel
SID              : S-1-5-21-2153971331-1430186003-2770964410-1230
Surname          :
UserPrincipalName :
```

L'affichage met en évidence que la propriété `PrimaryGroup` a comme valeur '`Domain Users`', c'est-à-dire utilisateurs du domaine. Maintenant que nous avons cette information de base, ajoutons le compte d'utilisateur '`Nawphel`' au groupe '`Financial DL`' à l'aide de la commande `Add-ADGroupMember` :

```
PS> Add-ADGroupMember -Identity 'Financial DL' -Members 'Nawphel' -PassThru
```

```
DistinguishedName : CN=Financial DL,CN=Users,DC=contoso,DC=com
GroupCategory     : Distribution
GroupScope        : Universal
Name              : Financial DL
ObjectClass       : group
ObjectGUID        : ba2edab5-f85c-48de-99d9-09958708d358
SamAccountName   : Financial DL
SID               : S-1-5-21-2153971331-1430186003-2770964410-1223
```

La commande s'est exécutée correctement, c'est-à-dire que l'objet affiché à l'écran représente le groupe '**Financial DL**' dont l'utilisateur '**Nawphel**' est un nouveau membre. Nous pouvons le vérifier de la façon suivante :

```
PS> Get-ADGroupMember -Identity 'Financial DL'
```

```
distinguishedName : CN=Nawphel,OU=IT,OU=Accounts,DC=contoso,DC=com
name              : Nawphel
objectClass       : user
objectGUID        : 637b5806-00da-452f-8611-89b2d8f6f097
SamAccountName   : Nawphel
SID               : S-1-5-21-2153971331-1430186003-2770964410-1230

distinguishedName : CN=Elisabetta Scotti,OU=Executives,OU=Accounts,DC=contoso,
                   DC=com
name              : Elisabetta Scotti
objectClass       : user
objectGUID        : 9b262d9f-c99a-418a-9283-258f8579942d
SamAccountName   : ElisabettaS
SID               : S-1-5-21-2153971331-1430186003-2770964410-1148

distinguishedName : CN=Bob Kelly,OU=Legal,OU=Accounts,DC=contoso,DC=com
name              : Bob Kelly
objectClass       : user
objectGUID        : 7b457923-dc5c-4e8f-8442-98bf6db4a8b6
SamAccountName   : BobK
SID               : S-1-5-21-2153971331-1430186003-2770964410-1140
```

La gestion des groupes Active Directory est donc similaire à celle des comptes d'utilisateurs. Elle consiste en l'agrégation de plusieurs commandes pour atteindre un but unique. Dans ces circonstances, il appartient aux administrateurs de les utiliser au maximum afin de mieux les maîtriser et donc d'être efficaces dans l'automatisation de leurs tâches quotidiennes.

## Créer un domaine Active Directory

La création d'un domaine Active Directory est une opération qui certes n'est pas quotidienne, mais qui peut être automatisée. L'outil central servant à créer un domaine est **dcpromo.exe**. Lorsqu'un nouveau domaine est créé à l'aide d'un serveur, ce dernier peut être automatiquement promu en tant que contrôleur de domaine Active Directory.

Nous allons donc créer un domaine supplémentaire nommé `it.scripting.net` et qui sera enfant du domaine `scripting.net`. L'opération sera évidemment réalisée sur un serveur différent et qui sera donc le premier contrôleur de domaine du domaine enfant. Le tableau suivant liste les paramètres que nous allons utiliser pour créer ce nouveau domaine.

**Tableau 24-1** Liste (partielle) des paramètres de l'outil `dcpromo.exe`

Paramètre	Description
<code>/unattend</code>	Effectue une installation automatique.
<code>/adv</code>	Active les options avancées.
<code>/replicaornewdomain</code>	Spécifie s'il faut installer un nouveau contrôleur de domaine ou un domaine supplémentaire.
<code>/newdomain</code>	Spécifie le type de domaine à créer. Il peut s'agir d'une nouvelle arborescence de domaine, d'un domaine enfant ou d'une nouvelle forêt.
<code>/parentdomaindnsname</code>	Spécifie le nom de domaine complet d'un domaine parent existant.
<code>/newdomaindnsname</code>	Spécifie le nom de domaine pleinement qualifié ou FQDN du nouveau domaine.
<code>/childname</code>	Spécifie le nom DNS du domaine enfant.
<code>/domainnetbiosname</code>	Spécifie un nom netbios pour le nouveau domaine.
<code>/installdns</code>	Spécifie si le service DNS doit être installé ou non.
<code>/databasepath</code>	Spécifie le chemin où sera stockée la base de données Active Directory.
<code>/logpath</code>	Spécifie le chemin où seront stockés les journaux d'événements Active Directory.
<code>/sysvolpath</code>	Spécifie le chemin où sera stocké le dossier <code>SYSVOL</code> .
<code>/safemodeadminpassword</code>	Spécifie un mot de passe administrateur pour les opérations ' <code>'Safe Mode'</code> '.
<code>/domainlevel</code>	Spécifie le niveau fonctionnel de domaine.
<code>/rebootoncompletion</code>	Indique si le serveur doit redémarrer lorsque l'opération sera terminée.
<code>/username</code>	Spécifie le nom d'utilisateur qui servira à effectuer l'opération.
<code>/userdomain</code>	Spécifie le nom de domaine du compte d'utilisateur servant à effectuer l'opération.
<code>/password</code>	Spécifie le mot de passe du compte d'utilisateur servant à effectuer l'opération.

À partir de ces paramètres, voici la commande qui nous servira à créer un nouveau domaine :

```
PS> cmd /c 'dcpromo /unattend /adv /replicaornewdomain:domain /newdomain:child
/parentdomaindnsname:scripting.net /newdomaindnsname:it.scripting.net
/childname:it /domainnetbiosname:it /installdns:yes /databasepath:"C:\ntds"
/logpath:"C:\ADLogs" /sysvolpath:"C:\sysvol" /safemodeadminpassword:"#mx/
563msx" /domainlevel:4 /rebootoncompletion:yes /username:"administrateur"
/userdomain:"scripting.net" /password:"#mx/563msx"'
```

Vérification pour déterminer si les fichiers binaires des services de domaine Active Directory sont installés...

Installation des services de domaine Active Directory

Validation de l'environnement et des paramètres...

-----  
Les actions suivantes seront effectuées :

Configurer ce serveur en tant que premier contrôleur de domaine Active Directory dans un nouveau domaine enfant.

Le nom du nouveau domaine est « it.scripting.net ».

Le nom NetBIOS du domaine est « IT ».

Ce nouveau domaine est un domaine enfant du domaine « scripting.net ».

Niveau fonctionnel du domaine : Windows Server 2008 R2

Site :

Options supplémentaires :

Contrôleur de domaine en lecture seule : « Non »

Catalogue global : Non

Serveur DNS : Oui

Créer une délégation DNS : Oui

Contrôleur de domaine source : n'importe quel contrôleur de domaine accessible en écriture

Dossier de la base de données : C:\ntds

Dossier des fichiers journaux : C:\ADLogs

Dossier Sysvol : C:\sysvol

Le service Serveur DNS sera installé sur cet ordinateur.

Le service Serveur DNS sera configuré sur cet ordinateur.

Cet ordinateur sera configuré pour utiliser ce serveur DNS en tant que serveur DNS préféré.

Le mot de passe du nouvel administrateur de domaine sera le même que celui de l'administrateur local de cet ordinateur

Démarrage en cours...

Installation du service DNS en cours d'exécution...

Appuyez sur Ctrl+C pour : Annuler

En attente de la fin de l'installation du service DNS

....

En attente de la détection du service Serveur DNS... 0

En attente du démarrage du service Serveur DNS... 0

Vérification pour déterminer si la Console de gestion des stratégies de groupe doit être installée...

.....  
Installation de la Console de gestion des stratégies de groupe...

.....  
Création du volume système C:\sysvol

.....  
Analyse d'une forêt existante...  
(...)

Une fois l'opération terminée, le nouveau contrôleur de domaine devrait redémarrer. Dans la section suivante, nous créerons dans ce même sous-domaine une zone DNS de recherche directe ainsi qu'une zone DNS de recherche inversée.

## Les zones DNS de recherches directes et inversées

Le service DNS (*Domain Name System*) est sans doute l'élément dont dépend le plus Active Directory. En effet, il est utilisé par Active Directory pour localiser les contrôleurs de domaines, trouver des ressources précises, ou tout simplement pour la communication entre les différentes machines d'un domaine. L'importance du DNS dans l'écosystème de l'Active Directory est capitale. Sans ce service, l'Active Directory ne pourrait même pas fonctionner.

Le principe de fonctionnement du service DNS est simple : établir une correspondance entre une adresse IP et un nom, à l'instar d'un annuaire téléphonique. Au cœur du service DNS existe la notion de zone DNS. Une zone DNS est une sorte de

conteneur logique où sont justement représentés ces liens nom-adresse IP. Lorsqu'un serveur DNS est sollicité pour répondre à une requête, il interroge la ou les zone(s) qu'il contrôle.

Dans la section précédente, nous avons créé un domaine nommé `it.scripting.net`, enfant du domaine `scripting.net`. Dans cette section, nous y créerons une zone de recherche directe et une zone de recherche inversée.

## Créer une zone DNS de recherche directe

Les zones DNS de recherche directe servent à effectuer des recherches à partir de noms d'hôtes pour trouver les adresses IP correspondantes. Nous allons créer une zone de recherche directe au sein du domaine `it.scripting.net`. Pour ce faire, nous passerons par la classe WMI `MicrosoftDNS_Zone`. Commençons par instancier cette classe :

```
PS> $DNSDirectZone = [WMICLASS]"root\MicrosoftDNS:MicrosoftDNS_Zone"
```

Listons les membres de l'objet créé pour mieux le connaître :

```
PS> $DNSDirectZone | gm
```

Name	MemberType	Definition
-----	-----	-----
Name	AliasProperty	Name = __Class
<b>CreateZone</b>	Method	System.Management..
__CLASS	Property	System.String __CLASS {get;set;}
__DERIVATION	Property	System.String[] __DERIVATION
__DYNASTY	Property	System.String __DYNASTY {get;set;}
__GENUS	Property	System.Int32 __GENUS {get;set;}
__NAMESPACE	Property	System.String __NAMESPACE {get;set;}
__PATH	Property	System.String __PATH {get;set;}
__PROPERTY_COUNT	Property	System.Int32 __PROPERTY_COUNT
__RELPATH	Property	System.String __RELPATH {get;set;}
__SERVER	Property	System.String __SERVER {get;set;}
__SUPERCLASS	Property	System.String __SUPERCLASS {get;set;}
ConvertFromDateTime	ScriptMethod	System.Object ConvertFromDateTime();
Convert.ToDateTime	ScriptMethod	System.Object Convert.ToDateTime();

Un membre retient l'attention. Il s'agit de la méthode `CreateZone()`. Vous l'aurez compris, cette méthode va nous permettre de créer une zone de recherche directe :

```
PS> $DNSDirectZone.CreateZone("m1.it.scripting.net",0,$true)
```

```
__GENUS      : 2
__CLASS     : __PARAMETERS
```

```
__SUPERCLASS      : 
__DYNASTY        : __PARAMETERS
__RELPATH         :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER          :
__NAMESPACE       :
__PATH            :
RR                :
MicrosoftDNS_Zone.ContainerName="m1.it.scripting.net",DnsServerName=".",
Name="m1.it.scripting.net"
```

La méthode `CreateZone()` prend un certain nombre d'arguments. Dans notre contexte, nous lui en avons passé trois :

- le nom de la zone DNS de recherche directe ;
- la valeur `0` indiquant qu'on veut une zone primaire ;
- la valeur `$true` indiquant que la zone doit être intégrée à l'Active Directory.

La sortie nous montre que la zone DNS a bien été créée. Nous pouvons maintenant passer à la création d'une zone de recherche inversée.

## Créer une zone DNS de recherche inversée

Le principe des zones DNS de recherche inversée est de partir de l'adresse IP pour trouver le nom d'hôte. La méthode que l'on utilisera est exactement la même que pour la création d'une zone de recherche directe :

```
PS> $DNSReverseZone = [WMICLASS]"root\MicrosoftDNS:MicrosoftDNS_Zone"
```

Toutefois, le premier argument devra contenir `in-addr.arpa` comme partie du nom DNS de la zone de recherche inversée :

```
PS> $DNSReverseZone.CreateZone("2.0.10.in-addr.arpa",0,$true)
```

```
__GENUS      : 2
__CLASS      : __PARAMETERS
__SUPERCLASS :
__DYNASTY    : __PARAMETERS
__RELPATH    :
__PROPERTY_COUNT : 1
__DERIVATION : {}
__SERVER    :
__NAMESPACE  :
```

```
    PATH : MicrosoftDNS_Zone.ContainerName="2.0.10.in-addr.arpa",
RR      DnsServerName=".",Name="2.0.10.in-addr.arpa"
```

L'autre partie de cet argument consiste en un identifiant réseau dans un ordre inversé ([2.0.10](#)). De plus, cette zone est aussi intégrée à l'Active Directory et est une zone primaire.

Il apparaît clairement que l'administration de l'Active Directory peut être dirigée de plusieurs façons. Il est possible d'utiliser des outils natifs comme [dcpromo.exe](#), mais aussi WMI et maintenant un module PowerShell dédié. Il n'y a pas un seul outil à privilégier, mais tous, sans exception. Le fait est qu'ils sont tous très complémentaires et leur utilisation conjointe est incontestablement un facteur d'efficacité.

# 25

## Utiliser le framework .NET

---

*Pour diverses raisons, Microsoft a décidé que les utilisateurs devaient avoir accès au framework .NET. Et cette ouverture est sans doute une voie vers plus de puissance au niveau du code écrit, car les possibilités ne se limiteraient pas seulement aux cmdlets disponibles, mais s'étendraient à l'ensemble du framework .NET. Il s'agit là d'une vision à long terme, car les perspectives ouvertes attireront de plus en plus de développeurs .NET, qui verront dans ce langage des complémentarités avec, par exemple, C# ou Visual Basic .NET.*

*Dans ce chapitre, nous nous familiariserons avec le framework .NET pour mieux le comprendre. Puis nous apprendrons à l'utiliser à travers certains exemples comme l'ajout dynamique de types .NET dans une session PowerShell, l'envoi d'e-mails ainsi que l'obtention d'informations relatives au DNS.*

### SOMMAIRE

- ▶ Comprendre le framework .NET
- ▶ Ajouter un type .NET à une session PowerShell
- ▶ Envoyer un e-mail
- ▶ Collecter des informations DNS

## Comprendre le framework .NET

Le framework .NET est une technologie puissante produisant un environnement de développement extrêmement cohérent. En effet, l'ambition première de cette technologie est de permettre aux programmeurs de développer plus facilement, car un grand nombre (de plus en plus) de fonctionnalités sont disponibles sur cette plate-forme.

Le cœur du framework .NET réside dans l'immense bibliothèque de classes qu'il met à disposition des programmeurs. Ces classes, contenues dans des espaces de noms, contiennent quant à elles des membres :

- **Les constructeurs**

Utilisés lorsqu'un objet d'un type donné est créé. L'étape de la construction de l'objet permet notamment l'initialisation de ses données.

- **Les propriétés**

Elles fournissent ou modifient des informations d'état de l'objet en question.

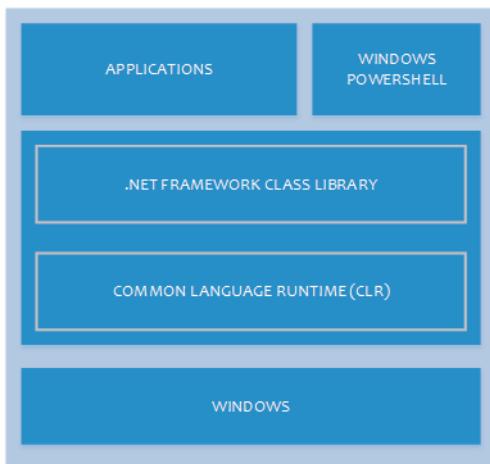
- **Les méthodes**

Elles définissent les actions qu'un objet peut effectuer.

Le framework .NET est aussi le cœur de PowerShell ; sans lui, PowerShell ne serait pas ce qu'il est aujourd'hui. Le schéma suivant illustre l'articulation Windows - framework .NET - PowerShell.

**Figure 25–1**

Le framework .NET constitue le noyau central de l'écosystème PowerShell.



En observant le schéma, on distingue deux éléments fondamentaux qui composent le framework .NET.

- **Le CLR (*Common Language Runtime*)**

C'est le fondement absolu du framework .NET. Il s'agit d'une machine virtuelle créée par Microsoft. Le CLR permet, entre autres, de fournir l'environnement d'exécution des programmes, de gérer l'allocation de mémoire aux programmes ou même de gérer les fonctionnalités d'exécution à distance.

- **Une bibliothèque de classes**

Le framework .NET fournit une vaste collection de classes grâce auxquelles les développeurs profitent des fonctionnalités offertes. L'objectif à travers cette bibliothèque de classes est d'accroître l'efficacité des programmes écrits, tout en obtenant un gain de temps significatif.

Le framework .NET est en réalité bien plus complexe que cela, mais le mode de compréhension que nous avons adopté est très largement suffisant par rapport à nos besoins.

## Ajouter un type .NET à une session PowerShell

Lorsque par exemple une console PowerShell est exécutée, elle charge un certain nombre de bibliothèques (ou DLL) indispensables à son bon fonctionnement. Ces bibliothèques constituent le socle fondamental permettant d'utiliser PowerShell de manière tout à fait normale. Néanmoins, il est possible d'ajouter dynamiquement des bibliothèques au cours d'une session. Et l'aspect dynamique de cette possibilité donne notamment comme conséquence un environnement bien plus léger.

La cmdlet `Add-Type` est une commande très puissante. Elle est l'outil privilégié lorsqu'il s'agit d'ajouter des types .NET, qu'ils soient issus de bibliothèques diverses ou alors encapsulés dans des variables. Par exemple, voici une classe très simple écrite en C# et encapsulée dans une variable.

### Création d'une classe en langage C#

```
❶ $Source = @"
public class BaseType ❷
{
    public void toupper(string value) ❸
    {
        System.Console.WriteLine("Conversion operation -> {0}",
value.ToUpper());
    }
}
```

```
public static int add(int x, int y) ④
{
    return x + y;
}

public static int mul(int x, int y) ⑤
{
    return x * y;
}

public static int div(int x, int y) ⑥
{
    return x / y;
}

"@"

```

Le code n'est pas écrit en langage PowerShell, mais en C#. Dans la perspective qui est la nôtre, cela ne pose pas de problèmes particuliers, car le code peut être compris dans sa globalité. D'abord, une variable nommée `$Source` ① a été créée. Vient ensuite la création d'une classe, `[BaseType]` ②, qui contient quatre méthodes.

- `toupper(string value)` ③ prend une chaîne comme paramètre et en renvoie une copie en majuscules.
- `add(int x, int y)` ④ additionne deux valeurs qui lui sont passées en paramètres.
- `mul(int x, int y)` ⑤ multiplie deux valeurs qui lui sont passées en paramètres.
- `div(int x, int y)` ⑥ divise deux valeurs qui lui sont passées en paramètres.

La classe `[BaseType]` est également circonscrite par une chaîne *here-string* `@"..."@`. Parmi les quatre méthodes, trois sont statiques (`add`, `mul`, `div`) et une autre ne l'est pas (`toupper`). Maintenant que notre classe est créée, utilisons la cmdlet `Add-Type` pour l'ajouter à une session PowerShell :

```
PS> Add-Type -TypeDefinition $Source
```

Ici, le paramètre `-TypeDefinition` sert à spécifier le code source contenant le type que nous voulons ajouter. Passons maintenant à l'instanciation de cette classe :

```
PS> $Object = New-Object -TypeName BaseType
```

Listons ses membres :

```
PS> $Object | Get-Member

TypeName: BaseType

Name      MemberType Definition
----      -----
Equals    Method     bool Equals(System.Object obj)
GetHashCode Method    int GetHashCode()
GetType   Method    type GetType()
ToString  Method   string ToString()
toupper   Method   void toupper(string value)
```

L'affichage nous montre uniquement une méthode sur les quatre qu'est censé contenir l'objet créé. Ce comportement est tout à fait normal, parce que les autres méthodes sont statiques ; pour les voir, il faut appeler le paramètre `-Static` de la cmdlet `Get-Member` :

```
PS> $Object | Get-Member -Static

TypeName: BaseType

Name      MemberType Definition
----      -----
add      Method     static int add(int x, int y)
div      Method     static int div(int x, int y)
Equals   Method     static bool Equals(System.Object objA,
                                         System.Object objB)
mul      Method     static int mul(int x, int y)
ReferenceEquals Method static bool ReferenceEquals(System.Object objA,
                                         System.Object objB)
```

L'objet étant constitué, il peut être utilisé comme les autres objets :

```
PS> $Object.toupper("powershell")
Conversion operation -> POWERSHELL
PS> $Object.toupper("Lisp")
Conversion operation -> LISP
PS> $Object.toupper("kais")
Conversion operation -> KAIS
PS> [Basetype]::add(7,7)
14
PS> [Basetype]::mul(7,9)
63
PS> [Basetype]::div(25,5)
5
```

La classe importée l'a été à partir d'une variable contenant la définition. Mais qu'en est-il s'il s'agit par exemple d'une DLL ? Pour y répondre, transformons la définition créée en une bibliothèque :

```
PS> Add-Type -TypeDefintion $Source -OutputType Library -OutputAssembly  
"./CodeDefinition.dll"
```

Cette opération sert à créer une bibliothèque dynamique nommée `CodeDefinition.dll`. Le paramètre `-OutputType` spécifie le type d'assemblage et le paramètre `-OutputAssembly` spécifie quant à lui le nom de l'assemblage créé. À présent, pour charger cette bibliothèque en mémoire, il faut une nouvelle fois appeler la commande `Add-Type` :

```
PS> Add-Type -Path .\CodeDefinition.dll -PassThru
```

IsPublic	IsSerial	Name	BaseType
True	False	BaseType	System.Object

La DLL a bien été importée et le type `[BaseType]` est de nouveau utilisable :

```
PS> [BaseType]::mul(8,69)  
552  
PS> [BaseType]::div(8300,45)  
184
```

D'autres méthodes sont disponibles pour importer des types, mais privilégier la cmdlet `Add-Type` est une bien meilleure solution, car elle concentre plusieurs fonctionnalités en son sein.

## Envoyer un e-mail

Parmi toutes les fonctionnalités offertes par le framework .NET se trouve l'envoi d'e-mails. Cette opération est très répandue, y compris dans le monde de l'administration système (par exemple, lorsque des scripts informent certaines personnes du bon ou mauvais déroulement des opérations). En passant par le framework .NET, l'envoi de messages électroniques se fait à l'aide de la classe `[System.Net.Mail.MailMessage]`, qui modélise la structure d'un e-mail :

```
PS> $email = new-object system.net.mail.mailmessage
PS> $email | Get-Member

TypeName: System.Net.Mail.MailMessage

Name          MemberType
----          -----
Dispose       Method
Equals        Method
GetHashCode   Method
GetType       Method
ToString      Method
AlternateViews Property
Attachments   Property
Bcc           Property
Body          Property
BodyEncoding   Property
BodyTransferEncoding Property
CC            Property
DeliveryNotificationOptions Property
From          Property
Headers       Property
HeadersEncoding Property
IsBodyHtml    Property
Priority      Property
ReplyTo       Property
ReplyToList   Property
Sender        Property
Subject       Property
SubjectEncoding Property
To            Property
```

L'objet `$email` constituera le message en tant que tel. Pour la définition des adresses, nous devrons utiliser la classe `[System.Net.Mail.MailAddress]` :

```
PS> $from = new-object system.net.mail.mailaddress("MarcL@contoso.com")
PS> $to = new-object system.net.mail.mailaddress("NawphelK@contoso.com")
```

La prochaine étape consiste à annexer les deux objets `$from` et `$to` à l'objet `$email` :

```
PS> $email.From = $from
PS> $email.To.Add($to)
```

Définissons l'objet ainsi que le corps du message :

```
PS> $email.Subject = "Message de Marc."
PS> $email.Body = "Bonjour Nawphel, c'est Marc. Ce message est juste un test de
ma part. Surtout n'en tiens pas compte."
```

Le message est maintenant créé. Il ne nous reste plus qu'à initialiser un autre objet à partir de la classe `[System.Net.Mail.SmtpClient]`, afin d'envoyer ce message :

```
PS> $smtp = New-Object system.net.mail.smtpclient("SLC-DC01")
```

Enfin, l'envoi du message se fera à l'aide de la méthode `Send()` :

```
PS> $smtp.Send($email)
```

L'exemple que nous venons d'étudier est simple, mais d'autres paramètres pourraient entrer en jeu comme le port SMTP ou des priviléges différents. De plus, l'exercice suppose que le serveur SMTP soit correctement configuré.

## Collector des informations DNS

Utiliser le DNS pour obtenir des informations au sujet d'une machine particulière est une pratique courante dans le monde de l'administration système/réseau. Ces informations sont effectivement très pertinentes dans le cadre de recherches particulièrement importantes.

La classe .NET (mais il y en a d'autres) qui est dédiée à ces tâches se nomme `[System.Net.Dns]` :

```
PS> [System.Net.Dns] | gm -Static
```

```
TypeName: System.Net.Dns
```

Name	MemberType	Definition
BeginGetHostAddresses	Method	static System.IAsyncResult BeginGetHo
BeginGetHostName	Method	static System.IAsyncResult BeginGetHo
BeginGetHostEntry	Method	static System.IAsyncResult BeginGetHo
BeginResolve	Method	static System.IAsyncResult BeginResol
EndGetHostAddresses	Method	static IPAddress[] EndGetHostAddresse
EndGetHostName	Method	static System.Net.IPHostEntry EndGetH
EndGetHostEntry	Method	static System.Net.IPHostEntry EndGetH
EndResolve	Method	static System.Net.IPHostEntry EndReso

Equals	Method	static bool Equals(System.Object objA
GetHostAddresses	Method	static IPAddress[] GetHostAddresses(S
GetHostAddressesAsync	Method	static System.Threading.Tasks.Task<IP
GetHostByAddress	Method	static System.Net.IPHostEntry GetHost
GetHostByName	Method	static System.Net.IPHostEntry GetHost
GetHostEntry	Method	static System.Net.IPHostEntry GetHost
GetHostEntryAsync	Method	static System.Threading.Tasks.Task<S
GetHostName	Method	static string GetHostName()
ReferenceEquals	Method	static bool ReferenceEquals(System.Ob
Resolve	Method	static System.Net.IPHostEntry Resolve

La classe `[System.Net.Dns]` est statique, comme tous ses membres. Il n'y a donc pas besoin de l'instancier pour l'utiliser. L'exemple suivant collecte toutes les adresses Ipv4 et Ipv6 de l'hôte '`Lendesk`' :

```
PS> [System.Net.Dns]::GetHostAddresses('Lendesk')
```

```
Address      : 
AddressFamily : InterNetworkV6
ScopeId      : 25
IsIPv6Multicast : False
IsIPv6LinkLocal : True
IsIPv6SiteLocal : False
IsIPv6Teredo : False
IsIPv4MappedToIPv6 : False
IPAddressToString : fe80::c499:edc:cc84:4a8c%25

Address      : 
AddressFamily : InterNetworkV6
ScopeId      : 21
IsIPv6Multicast : False
IsIPv6LinkLocal : True
IsIPv6SiteLocal : False
IsIPv6Teredo : False
IsIPv4MappedToIPv6 : False
IPAddressToString : fe80::85ed:b67a:bae1:a6c2%21

Address      : 
AddressFamily : InterNetworkV6
ScopeId      : 20
IsIPv6Multicast : False
IsIPv6LinkLocal : True
IsIPv6SiteLocal : False
IsIPv6Teredo : False
IsIPv4MappedToIPv6 : False
IPAddressToString : fe80::3449:5e47:c483:1d5f%20

Address      : 50440384
```

```
AddressFamily      : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
IsIPv6Teredo     : False
IsIPv4MappedToIPv6 : False
IPAddressToString : 192.168.1.3

Address          : 20490432
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
IsIPv6Teredo     : False
IsIPv4MappedToIPv6 : False
IPAddressToString : 192.168.56.1

Address          : 18852032
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
IsIPv6Teredo     : False
IsIPv4MappedToIPv6 : False
IPAddressToString : 192.168.31.1

Address          : 18786496
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
IsIPv6Teredo     : False
IsIPv4MappedToIPv6 : False
IPAddressToString : 192.168.30.1
```

Évidemment, il est possible de filtrer ces informations de multiples façons. Prenons par exemple la méthode `GetHostByName()`. Elle fournit des informations DNS concernant une machine particulière :

```
PS> [System.Net.Dns]::GetHostByName('Lendesk')

HostName  AddressList
-----  -----
Lendesk   {192.168.56.1, 192.168.31.1, 192.168.30.1, 192.168.1.3}
```

Comme le montre la sortie, il s'agit uniquement d'adresses Ipv4. Pour constituer notre liste, invoquons les propriétés `AddressList` et `IPAddressToString` :

```
PS> [System.Net.Dns]::GetHostByName('Lendesk').AddressList.IPAddressToString  
192.168.56.1  
192.168.31.1  
192.168.30.1  
192.168.1.3
```

La résolution DNS par l'adresse IP est aussi possible :

```
PS> [System.Net.Dns]::GetHostEntry('192.168.1.3').Hostname  
Lendesk
```

La classe `[System.Net.Dns]` est et restera une classe fréquemment utilisée, surtout en matière de scripting. Son mode d'utilisation reste cependant particulier. Il n'y a en effet pas besoin de l'instancier et tous ses membres sont statiques.



# 26

## Développer une interface graphique avec PowerShell

---

*Le framework .NET met à disposition des développeurs un certain nombre de classes pour créer des interfaces graphiques, qu'il s'agisse d'interfaces Windows Forms ou Windows Presentation Foundation. En conséquence, il est aussi possible de créer des interfaces graphiques avec PowerShell, ce qui élargit encore plus les perspectives en matière de scripting. En outre, cela autorise une plus grande diffusion des scripts écrits, car ces derniers sont plus universels derrière une interface graphique.*

*Une première approche en la matière est de différencier l'optique Forms de l'optique Presentation Foundation. En effet, ces deux technologies ont des points de convergence, mais restent essentiellement différentes. La création de ce genre d'interfaces étant un exercice généralement long, nous utiliserons PowerShell Studio 2012 pour nous aider dans le processus de création. Nous verrons donc comment produire une interface graphique, puis nous écrirons le code nécessaire, qui nous permettra d'exploiter complètement notre interface.*

### SOMMAIRE

- ▶ Windows Forms ou Windows Presentation Foundation ?
- ▶ PowerShell Studio 2012
- ▶ Créer une interface graphique
- ▶ Écrire le code

## Windows Forms ou Windows Presentation Foundation ?

Avant d'entrer dans le vif du sujet, il est intéressant d'évoquer les deux technologies que sont *Windows Forms* et *Windows Presentation Foundation* (ou WPF). Forms est antérieur à WPF et est à l'heure actuelle la méthode la plus utilisée en ce qui concerne la construction d'interfaces graphiques en .NET. Le nombre de contrôles qu'il rend disponibles est conséquent, ce qui implique que les applications utilisant cette technologie sont potentiellement plus riches.

WPF est une seconde approche de création d'interfaces graphiques. Cette technologie se base sur un paradigme déclaratif. Il s'agit donc ici d'une autre orientation en termes de programmation, qui amène d'une certaine manière les développeurs à prendre parti, même si ce n'est pas définitif. WPF propose sensiblement les mêmes fonctionnalités que Forms, avec une supériorité incontestable dans le rendu graphique comme la gestion de la 3D, les animations ou le contenu multimédia.

Une fois ces éléments mis en évidence, il reste à déterminer quelle orientation choisir. Du point de vue d'un administrateur ou d'un ingénieur système, le choix est libre car les deux orientations répondront parfaitement à leurs besoins. Cependant, l'option WPF est purement esthétique, c'est-à-dire que pour une même interface écrite avec ces deux technologies, il y aura un avantage esthétique en faveur de WPF. Toutefois, cet aspect ne devrait pas primer dans la logique d'un administrateur. En fait ce n'est pas son but premier. L'outil créé doit répondre aux exigences à partir desquelles il a été développé. Un regard doit toutefois être porté sur l'aspect, parce qu'il s'agit quand même d'une interface graphique, sans que cela soit prioritaire comme cela pourrait être le cas pour le développement de grosses applications où ces critères sont très importants.

Dans la réalité, les gens sont très partagés. Il y a les pro-WPF, qui préfèrent miser sur la potentialité et l'avenir de cette technologie. Et de l'autre côté, il y a les pro-Forms, qui misent quant à eux sur une technologie éprouvée et largement documentée. Cette division est plus le fait de programmeurs. Nous pourrions considérer qu'elle ne concerne pas les administrateurs. C'était en effet le cas jusqu'à il y a environ sept ans, mais depuis, ce (faux) débat est devenu une question importante dans le monde de l'administration système/réseau. La réponse à cette question n'est pas définitive, car beaucoup de gens, y compris chez les développeurs, ne savent pas que WPF n'est pas là pour remplacer à terme Forms, mais s'inscrit au contraire dans une dimension parallèle. Il n'y a pas de logique de remplacement, mais plutôt une logique de complémentarité. Et la différence est grande.

La question de la préférence est donc plus subjective qu'il n'y paraît. Ceux qui ont une préférence pour l'une ou l'autre technologie sont souvent exclusifs dans leur choix. Pour un administrateur, le choix devrait plutôt se porter sur Forms. La docu-

mentation est plus fournie, le nombre de contrôles est important et le développement est plus rapide, ce qui est fondamental.

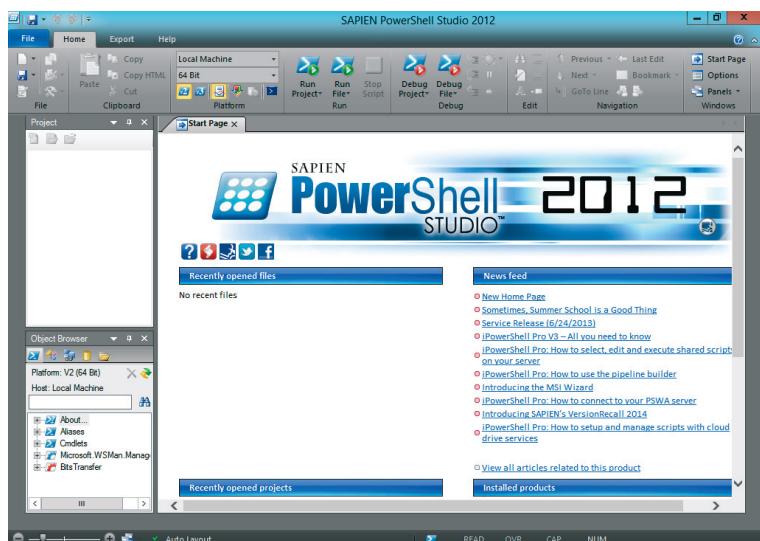
## PowerShell Studio 2012

La création d'une interface graphique complexe demande tout un cheminement. Pour peu qu'il n'y ait pas d'outils tiers pour optimiser la durée du développement, le temps de création peut véritablement être très long.

Le facteur temps est ici très important, car un administrateur ne peut pas se permettre de passer trop de temps sur la création d'interfaces graphiques. Cela ne fait pas partie de ses compétences. C'est dans ce contexte qu'entre en jeu PowerShell Studio 2012. Cet outil aide essentiellement au développement rapide d'interfaces graphiques.

**Figure 26–1**

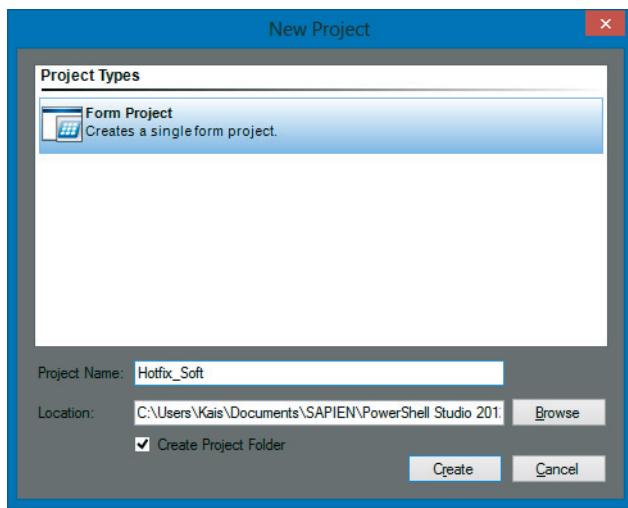
PowerShell Studio 2012 est à l'heure actuelle le meilleur outil de création d'interfaces graphiques avec PowerShell.



## Créer une interface graphique

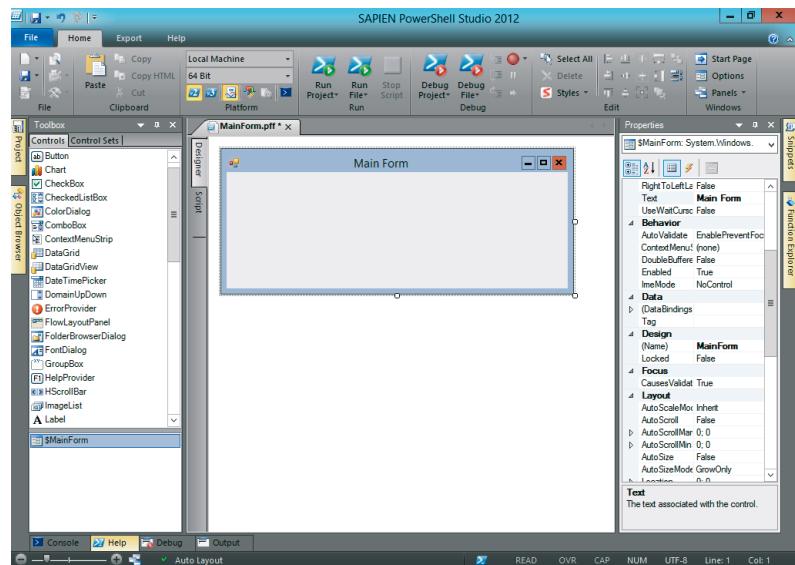
L'utilisation de PowerShell Studio 2012 est similaire à celle de Visual Studio. Le principe repose essentiellement sur la création de projets. Pour ce faire, il faut cliquer sur le bouton *New*, puis *New Form Project*.

**Figure 26–2**  
Créer une interface graphique passe par la création d'un projet.



Le projet créé doit évidemment avoir un nom, ici [Hotfix\\_Soft](#). Il nous servira à collecter les *hotfixes* (ou correctifs) installés sur une machine locale ou distante. Après avoir cliqué sur le bouton [Create](#), la fenêtre suivante apparaît.

**Figure 26–3**  
PowerShell Studio 2012 est une plate-forme riche ressemblant fortement à celle de Visual Studio.



Nous pouvons à présent définir notre interface graphique. Les contrôles se trouvent à gauche de l'écran ([Toolbox](#)). Ce sont des objets à part entière. Ils ont donc des pro-

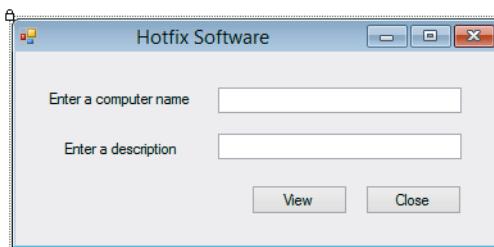
priétés et des méthodes, visibles à droite de l'écran (*Properties*). Voici la liste des contrôles qui seront ajoutés à l'interface :

- un bouton qui servira à fermer l'application ;
- un bouton qui servira à obtenir la liste des correctifs ;
- une zone de texte permettant d'indiquer le nom de la machine à partir de laquelle une liste sera collectée ;
- une autre zone de texte spécifiant le type de correctif à lister ;
- une étiquette identifiant la zone de texte relative au nom de machine ;
- une autre étiquette identifiant la zone relative au type de correctif ;
- un objet (*errorprovider*) pour détecter d'éventuelles erreurs en lien avec les zones de texte.

De plus, le nom de l'interface sera 'Hotfix Software' (en modifiant la propriété **Text** de la fenêtre principale).

**Figure 26–4**

L'interface graphique dispose d'objets élémentaires, mais suffisants.



C'est cette interface que l'utilisateur aura en face de lui lorsqu'il utilisera cet outil. L'étape suivante va consister à écrire le code qui sera exécuté lorsque l'utilisateur interagira avec l'outil en question.

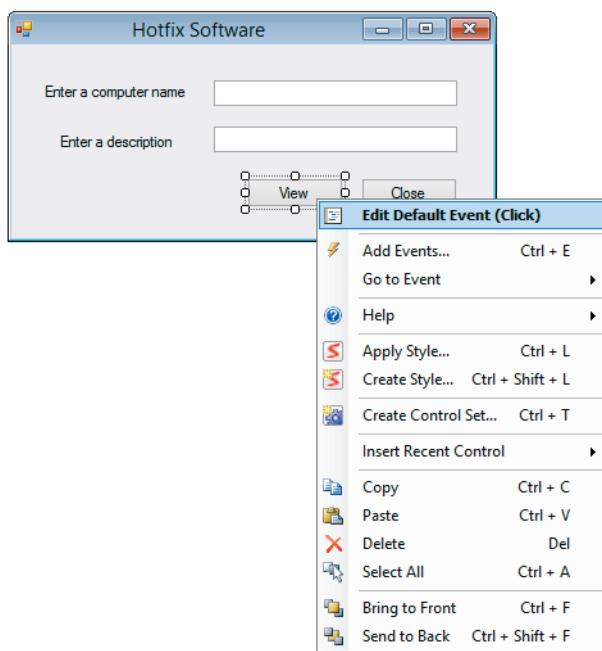
## Écrire le code

Le principe d'utilisation de l'outil Hotfix Software est simple : l'utilisateur entre un nom de machine locale ou distante, ainsi qu'une description du genre de correctif. Ces deux informations sont obligatoires pour que l'outil fonctionne correctement. Si une des deux informations venait à manquer, alors une erreur serait signalée.

Le bouton *Close* sert à fermer l'application et le bouton *View* lance une autre fenêtre montrant les résultats sous forme de tableau. L'avantage est que ce dernier sera interactif et que les résultats pourront aisément être filtrés.

L'écriture du code sera donc basée sur l'ensemble de ces éléments. Chaque élément a un nom par défaut (mais qui peut être modifié à l'aide de la propriété `Name`), qui représentera l'objet que nous devrons utiliser. Pour écrire le code lié à un événement d'un objet particulier, il faut cliquer droit sur l'objet et sélectionner `Edit Default Event`.

**Figure 26–5**  
Gérer les événements  
d'un objet donne accès  
à l'éditeur de code.



Puis l'éditeur de script s'ouvre et le code peut donc être ajouté.

**Figure 26–6**  
L'éditeur de code de PowerShell  
Studio 2012

```

14 $buttonView_Click={
15     #TODO: Place custom script here
16 }
17
18

```

L'ensemble du code en lien avec les événements d'un objet devra être écrit dans un bloc de script. Commençons par le bouton `View`.

#### Événement Click du bouton View

```

$buttonView_Click={
    #TODO: Place custom script here
    if ($textBoxCN.Text -eq '') { ①

```

```
$ErrorChecker.SetError($textboxCN,"Please enter a valid computer name")
} else {
    $ErrorChecker.SetError($textboxCN,"")
}

if ($textboxDes.Text -eq '') { ②
    $ErrorChecker.SetError($textboxDes,"Please enter a valid description")
} else {
    $ErrorChecker.SetError($textboxDes,"")
}

③ if ((-not $textboxCN.Text -eq '') -and (-not $textboxDes.Text -eq '')) {
④     try {
        Get-HotFix -ComputerName $textboxCN.Text -Description $textboxDes.Text
        -ErrorAction 'Stop' | Out-GridView -Title 'List of hotfixes that are installed
        on $($textboxCN.Text) computer'
    } catch {
        ⑤ [void][System.Windows.Forms.MessageBox]::Show("$(($textboxCN.text) is
        not reachable or the description is not valid.", "Error")
    }
}
}
```

Tout d'abord, lorsqu'un utilisateur clique sur le bouton *View*, il faut vérifier que les zones de texte `$textboxCN` ① (pour le nom de la machine) et `$textboxDes` ② (pour le type de correctif) ne soient pas vides. Si l'une d'elles l'est, une erreur doit alors s'afficher.

Si les deux zones de texte sont remplies comme il se doit ③, alors la cmdlet `Get-Hotfix` pourra être invoquée avec ses paramètres `-ComputerName` et `-Description`. Évidemment, des erreurs sont susceptibles d'apparaître si le nom de la machine est mal orthographié ou si la machine en question n'est pas joignable. En conséquence, la commande `Get-Hotfix` est encapsulée dans une instruction `try-catch` ④, cette dernière contrôlant tout simplement les exceptions pouvant se produire. Enfin, si une erreur se produit, une fenêtre spécifique ⑤ indique à l'utilisateur que la machine n'est pas joignable.

Le bouton *Close* est quant à lui beaucoup plus simple à gérer.

#### Événement Click du bouton Close

```
$buttonClose_Click={
    #TODO: Place custom script here
    $MainForm.Close()
}
```

Cliquer sur ce bouton ferme l'application. L'action doit donc s'effectuer au niveau de la fenêtre principale, `$MainForm`. La méthode à utiliser est `Close()`.

Pour créer notre outil, nous n'avons finalement écrit que très peu de lignes de code. En effet, une grande partie du code est écrite par PowerShell Studio lui-même.

#### Code produit par PowerShell Studio

```
#-----
# Source File Information (DO NOT MODIFY)
# Source ID: 42520586-4ddf-43e7-8fb-2320b2715d40
# Source File: C:\Users\Kais\Documents\SAPIEN\PowerShell Studio
# 2012\Projects\Hotfix_Soft\MainForm.pff
#-----
#region File Recovery Data (DO NOT MODIFY)
<#RecoveryData:
/RYAAB+LCAAAAAAABADNWOFv6jYQ/z5p/00UzwwIgUIniNSm7Va90odeWPc+IE0muVAPJ0aOA2Ta
Hz87JJCQMAIP1AoJyPn09/PdL+ez+9/Apktg0QPiyPj5JOXpf2V4hn1EnjCBV+SBMUTYf6Lmqy9c
t98oDMdG8ukNWIcpb2j1Vr+RFWymnf4NNld4tICBakUBB6/+J/YdugrqcvbNd00pG6opyVSDdr0p
PzXFDAkPGQx8CD1DpKaMwnb9heIxNQ0/mDa7aKO3bnRbvU2NHu3quILsAM1XYyq20+Y0Eyoqib1
OaMkUG0kAuuI0QuwHiU2Jshgcwv/A6qhd29qitbR+o1U6YCrDhJPMQPM4hER1k94DY6F/RmBo8Yy
s0o28EfVLY4YH9EAcxEj1TAFwUHWZgD+UdsxrL1q/E65i9eKRV2+QqwE4ONSTJqYvFDkqMZXX/5K
fPFYvxH/pPrH0/2CpkCunu8HCgyGFzIwscP/y3sx0C/URpuYaq2actMux0VQ6vbdVjDc8EtrC0ct
vYL+GE2ffQfWqtGpoh2n+VESQ0GKs4NX0fa04jkIwxA7DoENwYokalwy/tk4YFjvEQrAMju/TIJW
lWhvSFDewiLmiegoAo58yywE3zx4IfyQM58T9dXZwIXfqZ0Ld70synQ6Z1UCHYeqtyf/Via/eVLy
i1T59EkwX38oB63eqTkW6+agtYFuNafck79q2dgGrsxCQ3g3By0urrog5rVX4SMz8pp6HZ0roLF
xuxQFawK5Y8A3nAQihL3cffInpuUUKYaYxaW2mdbJde62vPc4v+KRXvd0ucmyhuG1dnvaq99Dk+k
y2vSpNkrHd0kIpLLsUQ6PEqs/MNxvjwyRpkAscTiQHJ12sTezHew58AOHqoEicT5B1jCpgyZvoEL
gmg2FM5sjW0scghNR+MuCMATOCFItrNJZCTrHiIfzcATsa3fhZx6MVd369err1/Xpq7e69wgR79p
gy46w62ncs8XiPG/5q+TJG+T1bYZDcSzrf760j48MRGtFWXzyVL0rjd17XaSrNgh5Ci4i9PhPKh5
41YC/sDQShysz4Hc1N2023U1zek0kY70g5z4PwLWC2zKCJ5eILIVQoI4+rAUSufVMvfdIx+G8vvw
pSK9MBPFhbLIArbENpz1b1yGaPtIShawfUwLYH8kdinR71i6vt9A1F1EMnvZRmf7nZeU2DJpIny2
k3u9nGpeKqosdiHgjo04wMrerCDbapthICpx0p6Z+9DAiCDuihph/CJ7vvRht4mEvjW+kxt98m9n
SFfArHcgJL2g1l1DQZjuNPlo9S2wQ4bFNrbdi/IGe5ehu3WUssU25o2L2ofGkjk+6kqk2buoDKq
pVJ5U4H8KBv1fY1JFxHds/dcaoqyZ3kj4C0yB7BcXHqXvDE4PCSQebuuKMhQzfuk3L1zI8N+wYXs
dfd/t6Fnfv0WAAA=>
#endregion
#-----
# Code Generated By: SAPIEN Technologies, Inc., PowerShell Studio 2012 v3.1.21
# Generated By: Kais
#-----

#-----
# Generated Form Function
#-----
function Call-MainForm_pff {

#-----
#region Import the Assemblies
#-----
[void][reflection.assembly]::Load("System, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089")
```

```
[void][reflection.assembly]::Load("System.Windows.Forms, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089")
[void][reflection.assembly]::Load("System.Drawing, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")
[void][reflection.assembly]::Load("mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089")
[void][reflection.assembly]::Load("System.Data, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089")
[void][reflection.assembly]::Load("System.Xml, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089")
[void][reflection.assembly]::Load("System.DirectoryServices, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")
#endregion Import Assemblies

#-----
#region Generated Form Objects
#-----
[System.Windows.Forms.Application]::EnableVisualStyles()
$MainForm = New-Object 'System.Windows.Forms.Form'
$DescriptionLabel = New-Object 'System.Windows.Forms.Label'
$ComputerNameLabel = New-Object 'System.Windows.Forms.Label'
$textboxDes = New-Object 'System.Windows.Forms.TextBox'
$textboxCN = New-Object 'System.Windows.Forms.TextBox'
$buttonClose = New-Object 'System.Windows.Forms.Button'
$buttonView = New-Object 'System.Windows.Forms.Button'
$ErrorChecker = New-Object 'System.Windows.Forms.ErrorProvider'
$InitialFormWindowState = New-Object 'System.Windows.Forms.FormWindowState'
#endregion Generated Form Objects

#-----
# User Generated Script
#-----

$OnLoadFormEvent={
#TODO: Initialize Form Controls here

}

$buttonClose_Click={
#TODO: Place custom script here
$MainForm.Close()
}

$buttonView_Click={
#TODO: Place custom script here
if ($textboxCN.Text -eq '') {
    $ErrorChecker.SetError($textboxCN,"Please Enter a valid computer name")
} else {
```

```
$ErrorChecker.SetError($textboxCN,"")
}

if ($textboxDes.Text -eq '') {
    $ErrorChecker.SetError($textboxDes,"Please Enter a valid description")
} else {
    $ErrorChecker.SetError($textboxDes,"")
}

if ((-not $textboxCN.Text -eq '') -and (-not $textboxDes.Text -eq '')) {
    try {
        Get-HotFix -ComputerName $textboxCN.Text -Description
        $textboxDes.Text -ErrorAction 'Stop' | Out-GridView -Title "List of hotfixes
        that are installed on $($textboxCN.Text) computer"
    } catch {
        [void][System.Windows.Forms.MessageBox]::Show("$(($textboxCN.text) is
        not reachable or the description is not valid.", "Error")
    }
}
# --End User Generated Script--
#-----
#region Generated Events
#-----

$Form_StateCorrection_Load=
{
    #Correct the initial state of the form to prevent the .Net maximized form
issue
    $MainForm.WindowState = $InitialFormWindowState
}

$Form_Cleanup_FormClosed=
{
    #Remove all event handlers from the controls
    try
    {
        $buttonClose.remove_Click($buttonClose_Click)
        $buttonView.remove_Click($buttonView_Click)
        $MainForm.remove_Load($OnLoadFormEvent)
        $MainForm.remove_Load($Form_StateCorrection_Load)
        $MainForm.remove_FormClosed($Form_Cleanup_FormClosed)
    }
    catch [Exception]
    { }
}
#endregion Generated Events
```

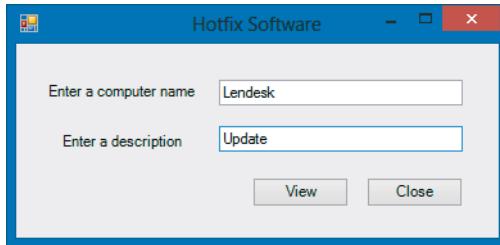
```
#-----
#region Generated Form Code
#-----
#
# MainForm
#
$MainForm.Controls.Add($DescriptionLabel)
$MainForm.Controls.Add($ComputerNameLabel)
$MainForm.Controls.Add($textboxDes)
$MainForm.Controls.Add($textboxCN)
$MainForm.Controls.Add($buttonClose)
$MainForm.Controls.Add($buttonView)
$MainForm.ClientSize = '376, 151'
$MainForm.FormBorderStyle = 'FixedSingle'
$MainForm.Name = "MainForm"
$MainForm.StartPosition = 'CenterScreen'
$MainForm.Text = "Hotfix Software"
$MainForm.add_Load($OnLoadFormEvent)
#
# DescriptionLabel
#
$DescriptionLabel.Location = '12, 64'
$DescriptionLabel.Name = "DescriptionLabel"
$DescriptionLabel.Size = '142, 23'
$DescriptionLabel.TabIndex = 5
$DescriptionLabel.Text = "Enter a description"
$DescriptionLabel.TextAlign = 'MiddleCenter'
#
# ComputerNameLabel
#
$ComputerNameLabel.Location = '12, 25'
$ComputerNameLabel.Name = "ComputerNameLabel"
$ComputerNameLabel.Size = '140, 23'
$ComputerNameLabel.TabIndex = 4
$ComputerNameLabel.Text = "Enter a computer name"
$ComputerNameLabel.TextAlign = 'MiddleCenter'
#
# textboxDes
#
$textboxDes.Location = '158, 64'
$textboxDes.Name = "textboxDes"
$textboxDes.Size = '190, 20'
$textboxDes.TabIndex = 3
#
# textboxCN
#
$textboxCN.Location = '158, 28'
$textboxCN.Name = "textboxCN"
$textboxCN.Size = '190, 20'
$textboxCN.TabIndex = 2
```

```
#  
# buttonClose  
#  
$buttonClose.Location = '273, 104'  
$buttonClose.Name = "buttonClose"  
$buttonClose.Size = '75, 23'  
$buttonClose.TabIndex = 1  
$buttonClose.Text = "Close"  
$buttonClose.UseVisualStyleBackColor = $True  
$buttonClose.add_Click($buttonClose_Click)  
#  
# buttonView  
#  
$buttonView.Location = '184, 104'  
$buttonView.Name = "buttonView"  
$buttonView.Size = '75, 23'  
$buttonView.TabIndex = 0  
$buttonView.Text = "View"  
$buttonView.UseVisualStyleBackColor = $True  
$buttonView.add_Click($buttonView_Click)  
#  
# ErrorChecker  
#  
$ErrorChecker.ContainerControl = $MainForm  
#endregion Generated Form Code  
  
#-----  
  
#Save the initial state of the form  
$InitialFormWindowState = $MainForm.WindowState  
#Init the OnLoad event to correct the initial state of the form  
$MainForm.add_Load($Form_StateCorrection_Load)  
#Clean up the control events  
$MainForm.add_FormClosed($Form_Cleanup_FormClosed)  
#Show the Form  
return $MainForm.ShowDialog()  
  
} #End Function  
  
#Call the form  
Call-MainForm_pff | Out-Null
```

Cela représente plus de 220 lignes de code. Le temps que nous avons gagné n'est donc pas négligeable. Pour vérifier que l'application fonctionne bien, il faut exécuter le projet en cliquant sur *Home*, puis *Run project*.

**Figure 26–7**

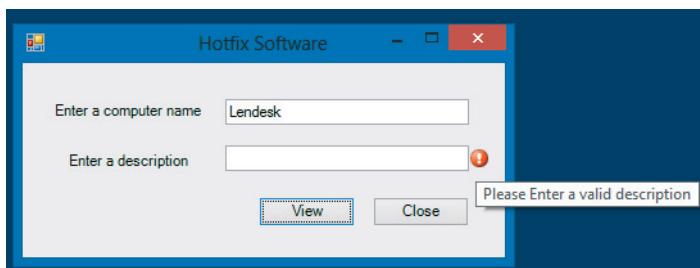
Lorsque l'application s'affiche à l'écran, il faut remplir les deux zones de texte.



Si un des deux champs n'est pas rempli, une erreur est signalée à l'utilisateur.

**Figure 26–8**

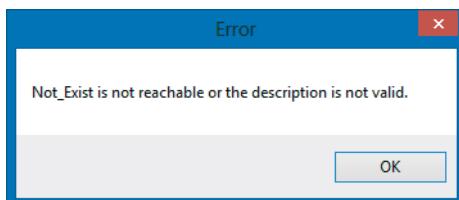
La gestion des erreurs se fait ici sur les deux zones de texte.



Si la machine n'est pas joignable ou si le nom n'est pas correctement orthographié, une fenêtre avec un message d'erreur s'affiche.

**Figure 26–9**

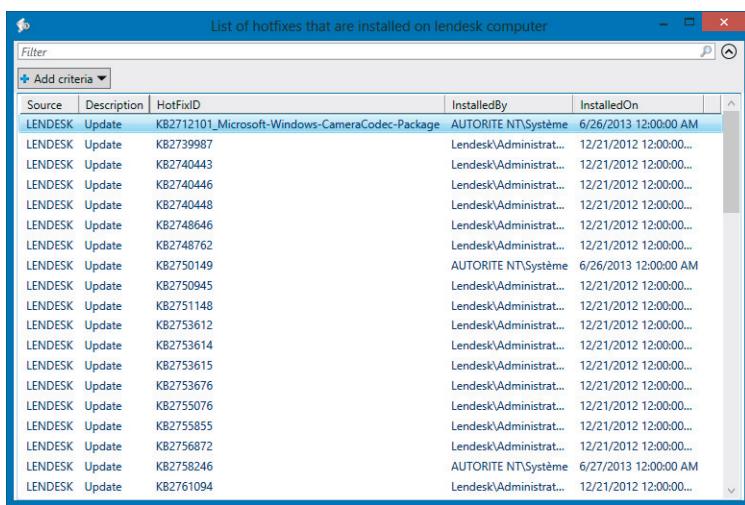
Des erreurs liées au bon déroulement de la cmdlet Get-Hotfix entraîneront un message d'erreur.



Dans le cas contraire, après avoir cliqué sur le bouton *View*, une autre fenêtre apparaît et la liste des correctifs prend la forme d'un tableau interactif.

**Figure 26–10**

La cmdlet Out-GridView est très utile lorsqu'il s'agit de donner une forme interactive aux informations.



The screenshot shows a Windows PowerShell window titled "List of hotfixes that are installed on lendesk computer". The window contains a grid view of data with the following columns: Source, Description, HotFixID, InstalledBy, and InstalledOn. The data is as follows:

Source	Description	HotFixID	InstalledBy	InstalledOn
LENDESK	Update	KB2712101_Microsoft-Windows-CameraCodec-Package	AUTORITE NT\Système	6/26/2013 12:00:00 AM
LENDESK	Update	KB2739987	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2740443	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2740446	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2740448	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2748646	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2748762	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2750149	AUTORITE NT\Système	6/26/2013 12:00:00 AM
LENDESK	Update	KB2750945	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2751148	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2753612	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2753614	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2753615	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2753676	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2755076	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2755855	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2756872	Lendesk\Administrat...	12/21/2012 12:00:00...
LENDESK	Update	KB2758246	AUTORITE NT\Système	6/27/2013 12:00:00 AM
LENDESK	Update	KB2761094	Lendesk\Administrat...	12/21/2012 12:00:00...

L'outil fonctionnant parfaitement bien, il peut être distribué (sous forme d'un pseudo-exécutable) aux personnes désireuses de l'utiliser. Bien sûr, il s'agit là d'une première version et l'application sera éventuellement modifiée en fonction des besoins.

L'exemple que nous avons étudié est simple, mais il est un premier pas dans le développement d'interfaces graphiques avec PowerShell Studio 2012.

# 27

## PowerShell sous Linux : rêve ou réalité ?

---

*Windows PowerShell est, comme nous le savons bien, dédié aux plates-formes Microsoft. Beaucoup de personnes se sont demandé si PowerShell pouvait être adapté à un environnement Linux, car une telle possibilité pourrait bien attirer énormément de gens du monde de Microsoft vers celui de l'open source. La réponse à cette question est oui. Il est effectivement possible que PowerShell fonctionne sur une distribution RedHat ou Debian, mais cela suppose d'écrire sa propre version car, pour des raisons évidentes, la version actuelle fonctionnant sous Windows ne fonctionne pas sous Linux. Donc, le travail est colossal et il n'y a eu qu'un seul projet à ce sujet, qui a été très vite abandonné.*

*Il y a un peu plus d'un an, j'ai décidé de créer mon propre projet et de sérieusement travailler sur la question d'une version stable de PowerShell sous Linux. Ce chapitre esquissera quelques aspects de mon travail. L'objectif est de montrer qu'il est possible d'administrer une plate-forme Linux avec PowerShell, mais au prix d'un investissement extrêmement lourd. Nous verrons comment gérer les services, les modules kernel et les disques avec cette version spéciale de PowerShell. Ensuite, nous expliquerons comment utiliser le framework .NET sur cette version de PowerShell.*

### SOMMAIRE

- ▶ PowerShell sous Linux est une réalité
- ▶ Gérer les services
- ▶ Gérer les modules kernel
- ▶ Gérer les disques
- ▶ Utiliser le framework .NET

## PowerShell sous Linux est une réalité

Avant de travailler sur cet outil dont le nom est [PSX], je me suis interrogé sur la probabilité de réussite d'un projet d'une telle envergure. En effet, PowerShell est un moteur éminemment complexe par essence et très difficile à maîtriser de l'intérieur. Le postulat de départ était donc qu'un projet de ce type était possible, mais il fallait repenser un certain nombre de mécanismes comme la liaison de paramètres ou le mode de fonctionnement du pipeline. En effet, même si du point de vue de l'utilisateur, le résultat était absolument le même, l'articulation des principes fonctionnels de PowerShell sous Linux serait essentiellement différente par rapport à un fonctionnement dans un environnement Windows.

Le fonctionnement d'un système Unix/Linux est bien différent de celui d'un système Windows, au sens où presque tout est fichier pour le premier genre de système, alors que le second est basé essentiellement sur le paradigme objet. Cette différence est importante parce que créer un shell avec un ensemble de commandes suppose de bien connaître en amont le système visé. Il se trouve que PowerShell est aussi basé sur le paradigme objet, qui est même le cœur et la force de son principe de fonctionnement. De l'autre côté, il existe par exemple Bash ou Ksh (Korn Shell) qui sont de formidables outils pour administrer une plate-forme Unix/Linux, mais qui sont purement orientés texte : tandis que PowerShell repose sur un pipeline d'objets, Bash et Ksh reposent quant à eux sur un pipeline de textes.

Vous aurez donc compris que la conception de PowerShell sous Linux exige une architecture autre que celle que nous connaissons. Heureusement, j'ai pu m'appuyer sur le framework Mono qui est une version open source du framework .NET. N'oublions pas que ce dernier est le socle de PowerShell ; sans lui, les perspectives proposées par PowerShell seraient bien plus réduites.

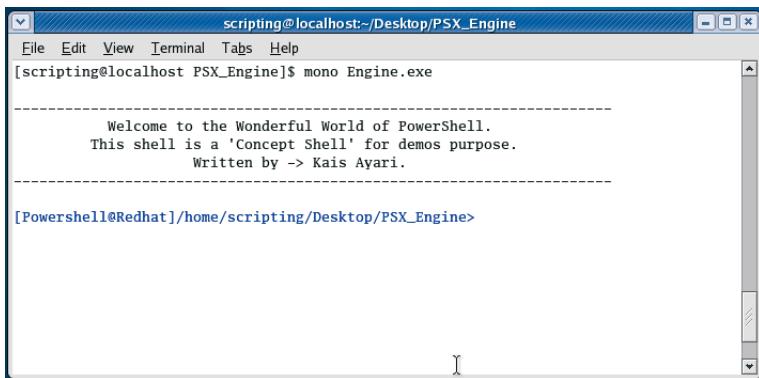
Pour concevoir l'architecture d'une version sous Linux de PowerShell, il m'a fallu de manière progressive penser les éléments suivants :

- un hôte ;
- un moteur (*Object Flow Engine*) ;
- une grammaire ;
- un processeur de pipeline (*Pipeline Processor*) ;
- un processeur de script (*Script Processor*) ;
- un processeur de commande (*Command Processor*) ;
- un contexte d'exécution dynamique ;
- un parser.

La liste n'est pas exhaustive, mais l'ensemble de ces éléments constitue les fondements de PowerShell. Travailler sur le développement de chacune de ces composantes est une tâche extrêmement ardue. Je ne m'étalerai pas dans cet ouvrage sur la façon dont j'ai écrit ce shell parce qu'il s'adresse essentiellement à des administrateurs et que, par ailleurs, décrire plusieurs dizaines de milliers de lignes de code dans un seul chapitre n'est pas efficace. Le but est de dire avec force que PowerShell sous Linux est une réalité.

Ceux qui connaissent un tout petit peu Linux savent qu'il y a plusieurs distributions. J'ai donc dû choisir sur quelles plates-formes j'allais développer. Le choix s'est porté sur les systèmes RedHat et Debian pour des raisons personnelles. Dans ce chapitre, la démonstration se fera donc à partir d'une version de PowerShell sous chacune de ces deux versions.

**Figure 27-1**  
En lançant PowerShell,  
le prompt change.



Évidemment, nous n'évoquerons pas dans ce chapitre toutes les commandes écrites, environ soixante-dix. Les axes sur lesquels j'ai insisté sont les suivants :

- les services ;
- les disques ;
- la gestion des configurations réseau ;
- les providers ;
- les modules kernel ;
- les commandes de base comme `Where-Object` et `ForEach-Object` ;
- la compilation de programmes à partir du code source (`Configure-SourceCode`, `Make-SourceCode` et `Install-CompiledCode`) ;
- les modules PowerShell.

**ALLER PLUS LOIN** **Vidéo de démonstration sur le projet PowerShell sous Linux**

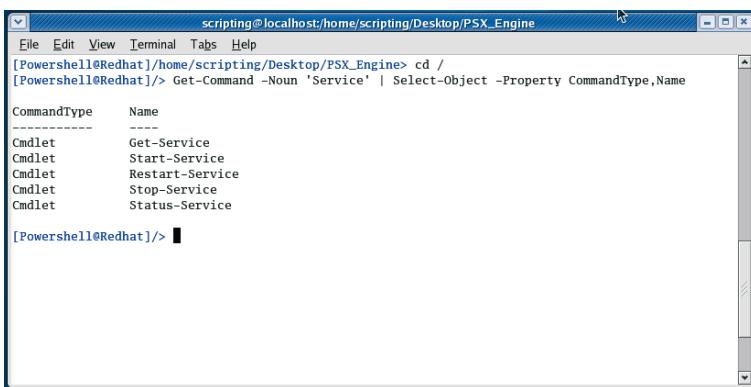
- ▶ <http://www.youtube.com/watch?v=80Rw9Txco5w>

L'exécution de PowerShell passe par l'appel de l'exécutable `mono`, car toute la programmation (C++/C#) est basée sur ce framework.

## Gérer les services

La gestion des services sous Linux est une opération bien connue des administrateurs système. Démarrer, redémarrer ou arrêter un service est une tâche d'administration quotidienne. Dans le monde Unix/Linux, la philosophie qui prévaut est particulière. J'ai donc développé un ensemble de cmdlets pour gérer aisément les services d'un système GNU/Linux.

**Figure 27–2**  
La gestion des services sous Linux est un élément très important.

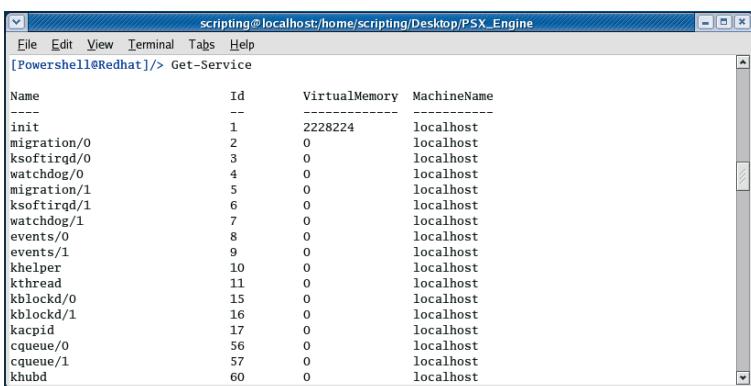


```
scripting@localhost:/home/scripting/Desktop/PSX_Engine
[Powerhell@Redhat]/home/scripting/Desktop/PSX_Engine> cd /
[Powerhell@Redhat]/> Get-Command -Noun 'Service' | Select-Object -Property CommandType,Name
CommandType      Name
-----      -----
Cmdlet          Get-Service
Cmdlet          Start-Service
Cmdlet          Restart-Service
Cmdlet          Stop-Service
Cmdlet          Status-Service

[Powerhell@Redhat]/>
```

Commençons par lister les services à l'aide de la commande `Get-Service`.

**Figure 27–3**  
La cmdlet `Get-Service` est une commande bien connue chez les utilisateurs de PowerShell.

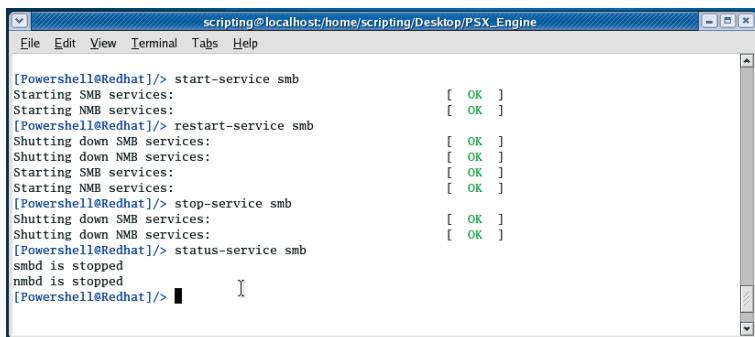


Name	Id	VirtualMemory	MachineName
init	1	2228224	localhost
migration/0	2	0	localhost
ksoftirqd/0	3	0	localhost
watchdog/0	4	0	localhost
migration/1	5	0	localhost
ksoftirqd/1	6	0	localhost
watchdog/1	7	0	localhost
events/0	8	0	localhost
events/1	9	0	localhost
khelper	10	0	localhost
kthread	11	0	localhost
kballocd/0	15	0	localhost
kballocd/1	16	0	localhost
kacpid	17	0	localhost
cqueue/0	56	0	localhost
cqueue/1	57	0	localhost
khubd	60	0	localhost

Pour démarrer, redémarrer, arrêter ou simplement vérifier l'état d'un service, nous utiliserons respectivement `Start-Service`, `Restart-Service`, `Stop-Service` et `Status-Service`.

**Figure 27–4**

Les cmdlets agissant sur les services peuvent idéalement être utilisées ensemble dans un même pipeline.



```
scripting@localhost:/home/scripting/Desktop/PSX_Engine
File Edit View Terminal Tabs Help
[Powershell@Redhat] /> start-service smb
Starting SMB services: [ OK ]
Starting NMB services: [ OK ]
[Powershell@Redhat] /> restart-service smb
Shutting down SMB services: [ OK ]
Shutting down NMB services: [ OK ]
Starting SMB services: [ OK ]
Starting NMB services: [ OK ]
[Powershell@Redhat] /> stop-service smb
Shutting down SMB services: [ OK ]
Shutting down NMB services: [ OK ]
smbd is stopped
nmbd is stopped
[Powershell@Redhat] /> status-service smb
[Powershell@Redhat] />
```

Actuellement, il y a exactement sur ce projet cinq cmdlets dédiées à la gestion des services, mais d'autres verront le jour.

## Gérer les modules kernel

Un module kernel, ou noyau, est un fichier séparé du noyau Linux pour rendre ce dernier modulaire. En général, il s'agit de pilotes ou de protocoles réseau. La particularité des modules kernel est qu'ils réduisent significativement la taille du noyau Linux. De plus, lorsqu'un de ces modules est ajouté, il est inutile de recompiler le noyau.

Sur le projet `[PSX]`, il y a trois cmdlets liées aux modules kernel.

- `Get-KernelModules` liste l'ensemble des modules noyau chargés en mémoire.
- `Import-KernelModule` importe dynamiquement un module noyau en mémoire.
- `Remove-KernelModule` libère dynamiquement un module noyau de la mémoire.

Donc, pour lister les modules, il faut utiliser la cmdlet `Get-KernelModules`.

**Figure 27–5**  
Les modules noyau chargés en mémoire sont nombreux.

```
scripting@localhost:/home/scripting/Desktop/PSX_Engine
File Edit View Terminal Tabs Help
[Powershell@Redhat]/> get-kernelmodules
vfat 15937 0 - Live 0xf8cf000
fat 51037 1 vfat, Live 0xf8e08000
usb_storage 80929 0 - Live 0xf8df3000
autofs4 28741 3 - Live 0xf8d9b000
hidp 22977 2 - Live 0xf8d16000
rfcomm 42457 0 - Live 0xf8dca000
l2cap 29761 10 hidp,rfcomm, Live 0xf8ca1000
bluetooth 53797 5 hidp,rfcomm,l2cap, Live 0xf8d8c000
lockd 63209 0 - Live 0xf8d7b000
sunrpc 149373 2 lockd, Live 0xf8da4000
be2iscsi 60757 0 - Live 0xf8d6b000
ib_iser 35609 0 - Live 0xf8d61000
rdma_cm 39929 1 ib_iser, Live 0xf8d56000
ib_cm 38061 1 rdma_cm, Live 0xf8d4b000
iw_cm 13253 1 rdma_cm, Live 0xf8cf7000
ib_sa 40117 2 rdma_cm,ib_cm, Live 0xf8d40000
ib_mad 38741 2 ib_cm,ib_sa, Live 0xf8d35000
ib_core 65985 6 ib_iser,rdma_cm,ib_cm,iw_cm,ib_sa,ib_mad, Live 0xf8d23000
ib_addr 12741 1 rdma_cm, Live 0xf8cf2000
iscsi_tcp 20041 0 - Live 0xf8c6f000
bnx2i 47325 0 - Live 0xf8d01000
cnic 52309 1 bnx2i, Live 0xf8cb9000
uio 14793 1 cnic, Live 0xf8c75000
cxgb3i 31177 0 - Live 0xf8c98000
libcxgb3i 54477 1 cxgb3i, Live 0xf8caa000
cxgb3 168985 1 cxgb3i, Live 0xf8cc7000
8021q 26313 1 cxgb3, Live 0xf8c90000
```

Essayons par exemple de libérer la mémoire du module `vfat`, pour ensuite l'importer de nouveau.

**Figure 27–6**  
PowerShell facilite la gestion des modules kernel.

```
scripting@localhost:/home/scripting/Desktop/PSX_Engine
File Edit View Terminal Tabs Help
[Powershell@Redhat]/> remove-kernelmodule vfat
The vfat module was removed from the kernel.
[Powershell@Redhat]/>
[Powershell@Redhat]/> import-kernelmodule vfat
The vfat module was added to the kernel.
[Powershell@Redhat]/>
```

Notez que lorsqu'un module kernel est importé, les dépendances associées à ce module sont résolues.

## Gérer les disques

Manipuler les disques est, tout comme la gestion des modules kernel, une tâche très récurrente en matière d'administration système. J'ai donc programmé quelques cmdlets pour effectuer des opérations de base comme monter ou démonter un disque avec PowerShell. Tout d'abord, listons les partitions actuellement montées.

**Figure 27–7**

La cmdlet Get-Drive liste l'ensemble des partitions montées.

```
scripting@localhost:/home/scripting/Desktop/PSX_Engine
File Edit View Terminal Tabs Help
[Powershell@Redhat]/home/scripting> get-drive
Name          File System   Total Size(GB) Free Space(GB)
----          -----
/             ext           16              7
/dev           tmpfs         0                0
/proc          proc          0                0
/sys           sysfs         0                0
/proc/bus/usb  usbdev       0                0
/boot          ext           0                0
/misc          autofs        0                0
[Powerhell@Redhat]/home/scripting>
```

Pour lister ces partitions, j'ai écrit la cmdlet [Get-Drive](#) qui renvoie des objets de type [PSCustomObject](#). À présent, insérons une clé USB et relançons la commande [Get-Drive](#) afin de vérifier que PowerShell la détecte bien.

**Figure 27–8**

PowerShell a parfaitement détecté le disque USB.

```
scripting@localhost:/home/scripting/Desktop/PSX_Engine
File Edit View Terminal Tabs Help
[Powershell@Redhat]/home/scripting> get-drive
Name          File System   Total Size(GB) Free Space(GB)
----          -----
/             ext           16              7
/dev           tmpfs         0                0
/proc          proc          0                0
/sys           sysfs         0                0
/proc/bus/usb  usbdev       0                0
/boot          ext           0                0
/misc          autofs        0                0
/media/SECURE-KAIS  msdos        3                0
[Powerhell@Redhat]/home/scripting>
```

La dernière ligne montre effectivement que le disque USB a bien été monté. Nous pourrions le démonter en utilisant des commandes Unix/Linux, mais en PowerShell nous utiliserons la cmdlet [Unmount-Drive](#).

**Figure 27–9**

La cmdlet Unmount-Drive accepte des objets en provenance du pipeline.

```
scripting@localhost:/home/scripting/Desktop/PSX_Engine
File Edit View Terminal Tabs Help
[Powershell@Redhat]/> Get-Drive -Name '/media/SECURE-KAIS' | Unmount-Drive
/media/SECURE-KAIS drive is now unmounted from the system.
[Powerhell@Redhat]/>
```

Ici, deux options sont possibles :

- soit utiliser directement [Unmount-Drive](#) avec son paramètre [-Name](#) ;
- soit utiliser conjointement [Get-Drive](#) et [Unmount-Drive](#).

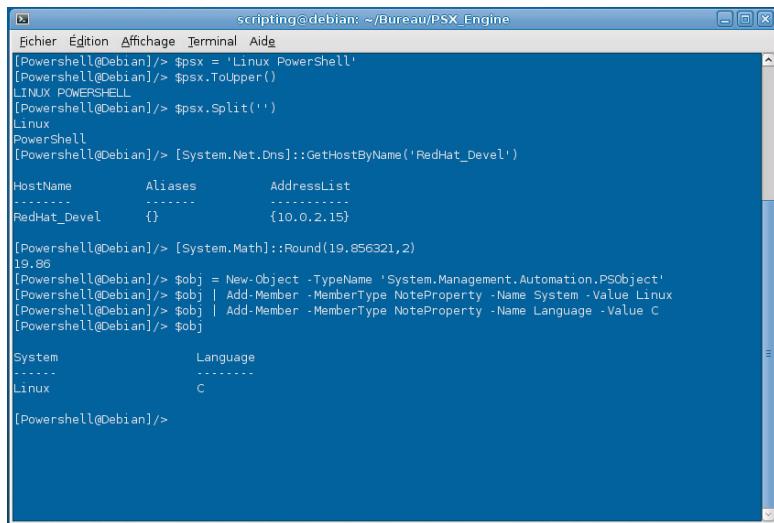
La seconde option est bien plus dans l'esprit de PowerShell. Si nous voulons remonter le disque USB, la commande `Mount-Drive` est aussi disponible, mais nous nous arrêterons là pour cette partie.

## Utiliser le framework .NET

L'utilisation directe du framework .NET est une des forces principales de PowerShell. En effet, en plus de l'aspect didactique de la chose, cela confère au code écrit beaucoup plus de puissance et de flexibilité. Là aussi, j'ai programmé [PSX] de façon à être le plus proche possible de PowerShell, même si l'architecture algorithmique est différente.

**Figure 27-10**

L'utilisation directe du framework .NET accroît sensiblement la puissance de PowerShell.



```
scripting@debian: ~/Bureau/PSX_Engine
Fichier Édition Affichage Terminal Aide
[Powershell@Debian]/> $psx = 'Linux PowerShell'
[Powershell@Debian]/> $psx.ToUpper()
LINUX POWERSHELL
[Powershell@Debian]/> $psx.Split('')
Linux
PowerShell
[Powershell@Debian]/> [System.Net.Dns]::GetHostByName('RedHat_Devel')

Hostname     Aliases     AddressList
-----       -----     -----
RedHat_Devel    {}          {10.0.2.15}

[Powershell@Debian]/> [System.Math]::Round(19.856321,2)
19.86
[Powershell@Debian]/> $obj = New-Object -TypeName 'System.Management.Automation.PSObject'
[Powershell@Debian]/> $obj | Add-Member -MemberType NoteProperty -Name System -Value Linux
[Powershell@Debian]/> $obj | Add-Member -MemberType NoteProperty -Name Language -Value C
[Powershell@Debian]/> $obj

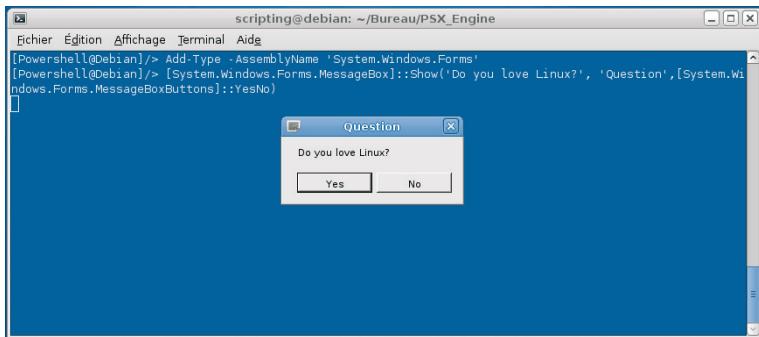
System      Language
----        -----
Linux          C

[Powershell@Debian]/>
```

Dans ces exemples mis en évidence à partir d'une plate-forme Debian, une chaîne est encapsulée dans une variable, laquelle est manipulée à l'aide des méthodes `ToUpper()` et `Split()` de la classe `[System.String]`. Puis est invoquée la méthode statique `GetHostByName()` de la classe `[System.Net.Dns]` afin de résoudre un nom DNS (celui d'un serveur RedHat). La classe `[System.Math]` est appelée à son tour pour arrondir une valeur particulière. Enfin, un objet est constitué à partir de la classe `[PSObject]`. Pour cela, les cmdlets `New-Object` et `Add-Member` ont été invoquées dans le but de construire l'objet `$obj` et de définir deux propriétés (`System` et `Language`). [PSX] permet donc la création d'objets de manière dynamique.

Il est même possible d'ajouter des bibliothèques (ou DLL) à l'aide de la cmdlet [Add-Type](#) et d'utiliser de manière dynamique les types .NET importés.

**Figure 27-11**  
PSX permet d'importer  
des bibliothèques et de profiter  
de la puissance  
du framework .NET.



PowerShell sous Linux est donc une réalité concrète que je vis au quotidien. Toutefois, il reste un certain nombre de bogues à corriger sur le projet [\[PSX\]](#) et de nouveaux axes d'orientation sont à développer. La patience est ici une arme indispensable.



# Index

---

! 100  
 \$\$ 93  
 \$( ) 116  
 \$? 93  
 \$^ 93  
 \$\_ 62, 93, 178  
 \$Args 93  
 \$args 147  
 \$ConfirmPreference 95  
 \$ConsoleFileName 93  
 \$DebugPreference 95  
 \$Error 93, 173, 176, 180  
 \$ErrorActionPreference 95, 176, 177  
 \$ErrorView 95  
 \$ExecutionContext 93  
 \$False 93  
 \$FormatEnumerationLimit 95  
 \$Home 93  
 \$Host 93  
 \$LastExitCode 93  
 \$Matches 94, 103, 285, 286  
 \$MaximumAliasCount 95  
 \$MaximumDriveCount 95  
 \$MaximumErrorCount 95  
 \$MaximumHistoryCount 95  
 \$MyInvocation 94  
 \$NULL 94  
 \$Null 205  
 \$null 110  
 \$OFS 95  
 \$OutputEncoding 95  
 \$PID 94  
 \$Profile 94  
 \$PSBoundParameters 94  
 \$PSCmdlet 94  
 \$PsCulture 94  
 \$PSDefaultParameterValues 95  
 \$PSEmailServer 95  
 \$PSHOME 66  
 \$PsHome 94  
 \$PsItem 94  
 \$PSModuleAutoloadingPreference 95  
 \$PSSessionConfigurationName 95  
 \$PSSessionOption 95  
 \$PsVersionTable 94  
 \$Pwd 94  
 \$Source 314  
 \$True 94  
 \$Using 274, 275  
 \$VerbosePreference 95  
 \$WarningPreference 96  
 \$WhatIfPreference 96  
 \$Workflow: 275  
 %= 104  
 & 30, 114  
 \* 72, 107  
 \*= 104  
 \*> 108  
 \*>&1 108  
 \*>> 108  
 ++ 99, 104  
 += 104  
 , 116  
 -- 99, 104  
 -= 104  
 -allmatches 288  
 -and 100  
 -ArgumentList 63  
 -argumentlist 193  
 -Arguments 195  
 -as 113

-AsHashTable 55  
-AutoSize 68  
-Begin 61, 62  
-CaseSensitive 141  
-casesensitive 287  
-Certificate 84  
-Column 76  
-ComObject 228, 235  
-ComputerName 191, 198, 213, 214, 256, 260  
-contains 101, 102  
-Credential 191, 256  
-DependentServices 199  
-Depth 74  
-Descending 53  
-DifferenceObject 56  
-End 61, 62  
-EntryType 211  
-eq 101  
-ErrorAction 176, 177  
-Exact 141  
-ExpandProperty 51  
-f 115  
-File 141  
-FilePath 84  
-Filter 191, 295  
-FilterScript 60  
-First 49, 52  
-ge 101  
-gt 101  
-HideTableHeaders 68  
-in 101, 104  
-Include 219, 220  
-IncludeEqual 56  
-Index 52  
-InDisconnectedSession 262  
-InputObject 200  
-is 113  
-isnot 113  
-join 112  
-Last 49  
-le 101  
-like 101, 103  
-List 191  
-ListAvailable 163  
-LogName 208  
-lt 101  
-match 94, 101, 102, 284, 286  
-MaximumSize 212  
-MemberName 63  
-MethodName 195  
-Module 164  
-Name 202, 218  
-name 193  
-NameSpace 191  
-ne 101  
-Newest 209  
-noelement 54  
-not 100  
-notcontains 101, 102  
-NotePropertyName 64  
-NotePropertyValue 64  
-notin 101, 104  
-notlike 101, 103  
-notmatch 94, 101, 102, 284, 286  
-or 100  
-OtherAttributes 295  
-OutputAssembly 316  
-OutputType 316  
-OverflowAction 213  
-Passthru 200  
-pattern 287  
-Persist 235  
-Process 61, 62  
-Properties 295  
-Property 50, 52, 54, 56, 61, 72, 76, 147  
-PSComputername 276  
-PSPrinter 217  
-query 193  
-Recurse 219, 220  
-ReferenceObject 56  
-Regex 141, 143  
-replace 101, 103  
-RequiredServices 199  
-RetentionDays 213  
-Scope 80  
-ScriptBlock 256  
-services 277  
-Session 260

- split 111
- StartupType 202
- Static 315
- Status 203
- TypeDefinition 314
- TypeName 63
- Unique 53
- Value 61
- WhatIf 96
- Wildcard 141
- xor 100
  - . 114, 115
  - .. 115
- .csv 39
- .DESCRIPTION 153
- .EXTERNALHELP 153
- .LINK 153
- .NET 5, 18, 38, 46, 63, 81, 91, 105, 113, 115, 153, 156, 165, 169, 180, 189, 223, 227, 229, 242, 265, 280, 282, 311, 312, 324, 337, 338, 344
  - bibliothèque de classes 313
  - CLR 313
  - collecter des informations DNS 318
  - Common Language Runtime 313
  - constructeur 312
  - courriel 316
  - espace de noms 312
  - méthode 312
    - propriété 312
- .NOTES 153
- .PARAMETER 153
  - .ps1 162
  - .psm1 162, 166, 167
- .SYNOPSIS 153
  - /= 104
  - :: 115
  - <= 56
  - = 104
  - == 56
  - => 56
  - > 108
  - >> 108
- @() 116
- [ ] 117
- [Alias] 158
- [AllowEmptyString] 158
- [AllowNull] 158
- [BaseType] 314
- [CmdletBinding] 156, 157, 167
- [ordered] 127
- [Parameter] 157
- [PSObject] 344
- [Regex] 282
- [System.Math] 344
- [System.Net.Dns] 318, 321, 344
- [System.Net.Mail.MailAddress] 317
- [System.Net.Mail.MailMessage] 316
- [System.Net.Mail.SmtpClient] 318
- [System.String] 344
- [System.Text.RegularExpressions.Regex] 282
- [ValidateCount] 158
- [ValidateLength] 158
- [WIN32] 195
  - [Win32\_Directory] 186
  - [Win32\_DiskDrive] 186
  - [Win32\_LogicalDisk] 234
  - [Win32\_NTEventLogFile] 213
  - [Win32\_Process] 195
  - [Win32\_Processor] 70, 72, 187, 244
  - [Win32\_Service] 205
    - ` 281
    - | 34, 116
- 1 107
- 2 107
- 2> 108
- 2>&1 108
- 2>> 108
- 3 107
- 3> 108
- 3>&1 108
- 3>> 108
- 4 107
- 4> 108
- 4>&1 108

4>> 108  
5 54, 107  
5> 108  
5>&1 108  
5>> 108

## A

accéder automatiquement à une page web 230  
accès à distance  
  configurer 252  
  connexion persistante 257  
  connexion temporaire 254  
  contexte 254  
  créer une session 258  
  déconnecter d'une session 261  
  détruire une session 260  
  fermer une session 259  
  quel type de connexion ? 260  
Active Directory VI, VIII, 78, 79, 289, 290  
  activer un compte d'utilisateur 298  
  ajouter un utilisateur 302  
  ajouter un utilisateur à un groupe 303  
  base de données 305  
  contrôleur de domaine 305, 307  
  créer un domaine 305  
  créer un utilisateur 295  
  définir un mot de passe 298  
  domaine 291, 298  
  Enable-ADAccount 298  
  forêt 291, 305  
  fournisseur 293  
  groupe 294, 302  
  journal d'événements 305  
  lecteur 293  
  lister les groupes 302  
  lister les membres d'un groupe 304  
  module 291  
  nom DNS 305  
  nom FQDN 305  
  nom netbios 305  
  recherche directe 307  
  recherche inversée 307  
  Safe Mode 305  
  trouver un utilisateur 295

unité d'organisation 291, 298, 302  
utilisateur 294, 295  
zone DNS de recherche 307  
zone DNS de recherche directe 308  
zone DNS de recherche inversée 309  
Active Directory Services Interfaces 227  
Add-ADGroupMember 303  
Add-Member 64, 344  
Add-Type 313, 314, 316, 345  
AddressList 321  
administration à distance 251  
ADSI 227, 291  
afficher les données  
  afficher par défaut 67  
  Format-Wide 75  
  liste de propriétés 67, 70  
  personnaliser l'affichage 73  
  personnaliser les propriétés 69  
  tableau 67  
  une propriété sous forme de tableau 75  
aide  
  intégrée 10, 66, 152  
  mot-clé 153  
ajouter un type .NET à une session  
  PowerShell 313, 314  
alias 26, 29, 282  
Alignment 69  
analyse syntaxique 24, 25  
AppendChild 243  
authentification 191

**B**

BackupEventLog 214  
base de registres VIII, 169, 215  
  ajouter une information 221  
  chercher une information 219  
  effacer une information 222  
  gérer à distance 223  
  lister les clés 217  
  lister les sous-clés 218  
  naviguer 216  
BEGIN 167  
begin 149, 150  
bloc de script 30, 59, 61, 62, 146, 254, 328

- bool 150  
boucle VIII, 129  
    compteur 135  
    do..until 134  
    do..while 133  
    for 130  
    foreach 131  
    while 135
- C**
- caractère  
    générique 23  
    spécial 25  
CategoryInfo 175  
cd 217  
certificat 80  
    autorité de certification 81  
    autosigné 81  
    créer 81  
    MakeCert.exe 82  
    obtenir 80  
chaîne  
    here-string 314  
    regex 141  
CIM 185, 186, 194  
    cmdlet 193, 194  
    créer une session CIM 195  
    invoyer une méthode 195  
    lister les classes 194  
    obtenir une instance 194  
    session 195  
    supprimer une session CIM 196  
CIMOM (CIM Object Manager) 187  
Clear-EventLog 214  
Clear-Variable 91  
CLI 4  
cmd.exe V  
cmdlet 18, 34, 36, 37, 41, 147, 169  
collection 119  
COM VIII, 193, 227, 291  
    créer un fichier texte 236  
    créer un objet 228  
    fermer un fichier texte 237  
    FileSystem 235  
    lire un fichier texte 237  
    manipuler un objet 228  
    OpenTextFile 237  
    ouvrir un fichier texte 237  
    ProgID 228  
    ReadAll 237  
Command-Line Interface 4  
command.com V  
commande 18, 34  
    native 18, 21  
    workflow 271  
Common Information Model 185, 186  
communication à distance 223  
Compare-Object 26, 55, 56, 59  
compilation 339  
Component Object Model *voir* COM  
configuration réseau 339  
Configure-SourceCode 339  
Connect-PSSession 258, 261  
connexion persistante 15  
console 4  
contexte d'exécution 93  
Continue 176, 177  
contrôle 324, 326  
    de flux VIII  
correctif 326  
CreateElement 242  
CreateTextFile 236  
CreateZone 308, 309  
culture 94  
CurrentDirectory 228  
CurrentUser 78
- D**
- dcpromo.exe 305  
    /adv 305  
    /childname 305  
    /databasepath 305  
    /domainlevel 305  
    /domainnetbiosname 305  
    /installdns 305  
    /logpath 305  
    /newdomain 305  
    /newdomaindnsname 305

/parentdomaindnsname 305  
/password 305  
/rebootoncompletion 305  
/replicaornewdomain 305  
/safemodeadminpassword 305  
/sysvolpath 305  
/unattend 305  
/userdomain 305  
/username 305  
délimiteur 111  
dépendance 342  
dictionnaire VIII, 124  
afficher 125  
créer 124  
membre 126  
utiliser 125  
Disconnect-PSSession 258, 261  
Dispose 225  
disque 339, 342  
démonter 343  
lister 342  
DLL 18, 313, 316, 345  
DNS 307  
documentation 229  
Domain Name System 307  
DoNotOverwrite 213

**E**  
éliminer les doublons 53  
Enable-PSRemoting 252  
Enabled 298  
end 150, 151  
Enter-PSSession 258  
Env: 91  
Environment 91  
erreur de pipeline 40, 44  
ErrorDetails 175  
ErrorRecord 174  
Exception 175  
Exec 229  
Exit-PSSession 258, 259  
Export-Alias 29  
Export-Clixml 244, 246  
Export-ModuleMember 162

Export-PSSession 258  
Expression 69  
expression régulière VIII, 141, 143, 279  
\$ 280  
\* 280, 281  
+ 280, 281  
. 280  
? 280, 281  
[^] 280  
\ 280  
\D 280  
\d 280  
\n 281  
\r 281  
\S 280  
\s 280  
\t 281  
\W 280  
\w 280  
^ 280  
{n,} 281  
{n,m} 281  
{n} 281  
caractère d'échappement 281  
classe de caractères 280  
quantificateur 281  
remplacer une occurrence 284  
séquence d'échappement 281  
syntaxe 280  
trouver une occurrence 284  
eXtensible Application Markup Language *voir*  
    XAML  
extension 162

**F**  
FileSystem 235  
filtre LDAP 295  
flux d'objets 34, 35, 49  
fonction VIII, 18, 19, 145, 146  
    \$args 147  
    aide 152  
    anonyme 146  
    avancée 156  
    déclarer 146

- documenter 152, 154  
lambda 146  
méthode avancée 149  
nom 146  
paramètre 147, 150  
paramètre avancé 157  
paramètre nommé 148  
paramètre positionnel 147, 149  
portée 150  
syntaxe 149
- foreach -parallel 272, 273, 277  
ForEach-Object 283  
Foreach-Object 15, 31, 61, 62, 63, 133, 339  
Format-Custom 65, 73, 74  
Format-List 65, 70, 75, 209, 210  
Format-Table 65, 67, 75  
Format-Wide 65  
FormatString 69  
fournisseur 91, 169, 187, 215, 216, 293  
    lister 216  
FQDN 191  
framework .NET VIII  
Fully Qualified Domain Name 191  
FullyQualifiedErrorId 175  
function 146
- G**  
gérer les erreurs 171, 178  
    analyser 173  
    erreur 172  
        exception 172, 175  
gestion d'erreur VIII  
Get-ADGroup 302  
Get-ADGroupMember 304  
Get-ADOrganizationalUnit 298  
Get-ADUser 295, 301, 302  
Get-Alias 26, 29  
Get-AuthenticodeSignature 85  
Get-ChildItem 218, 219  
Get-CimAssociatedInstance 194  
Get-CimClass 194  
Get-CimInstance 194  
Get-CimSession 194  
Get-Command 19, 164, 267, 291
- Get-Content 240, 283  
Get-Drive 343  
Get-EventLog 208, 210  
Get-ExecutionPolicy 80  
Get-Help 10, 91, 152, 154  
Get-Hotfix 335  
Get-Item 73, 236  
Get-ItemProperty 217, 222  
Get-KernelModules 341  
Get-Member 47, 123, 126, 187, 203, 224, 228,  
    230, 234, 235, 240, 246, 268, 282, 285, 308,  
    315, 317  
Get-Module 162, 163  
Get-Process 11, 132  
Get-PSDrive 216  
Get-PSProvider 91, 216  
Get-PSSession 258, 259, 261  
Get-Service 18, 22, 23, 30, 34, 36, 37, 38, 39, 40,  
    59, 75, 121, 198, 340  
Get-Variable 91  
Get-WinEvent 262  
Get-WmiObject 57, 70, 151, 173, 176, 177, 187,  
    190, 192, 203, 213, 234, 259  
GetHostName 320, 344  
GetValue 225  
GetValueNames 224  
Global: 150  
Group-Object 55  
groupe de travail 79
- H**  
help 10  
HelpMessage 158  
HKCU: 217  
HKEY\_CURRENT\_USER 217  
hotfix 326
- I**  
identifiant réseau 310  
Ignore 176  
Import-Alias 29  
Import-Clixml 244, 246  
Import-Csv 39, 40  
Import-KernelModule 341

Import-Module 162, 165, 291

Import-PSSession 258

InlineScript 274

Inquire 176

Install-CompiledCode 339

instruction conditionnelle 137

    break 142

    if..else 138

    if..elseif..else 139

    switch 140

Integrated Scripting Environment 6

interface graphique 323

    code 327

    créer 325

Internet Explorer 227, 230

InternetExplorer.Application 230

InvocationInfo 175

Invoke-CimMethod 194, 195

Invoke-Command 254, 256, 257, 260, 262

Invoke-WmiMethod 190, 192

invoquer 267

IPAddressToString 321

ISE 6, 7, 8, 15

## J

job 14, 271

journal d'événements 305

    Voir log 207

Jscript 227

## K

Keys 126

## L

Label 69

LanmanServer 203

lecteur 91, 95

    lister 216

Length 48

liaison de paramètres 37, 38, 39, 40, 41, 43

ligne de commande V, VI, VII, 20

Limit-EventLog 212

Linux VIII

Local: 150

LocalMachine 78

log 207

    configurer 212

    explorer le contenu 208

    filtrer les informations 210, 211, 212

    limiter l'affichage 209

    lister 208

    machine distante 213, 214

    sauvegarder 213

    supprimer 214

LogFileName 213

## M

machine virtuelle 313

Make-SourceCode 339

man 10

Mandatory 157

Manipuler 228

MapNetworkDrive 234

mapper un lecteur réseau 233

Match 284

méthode statique 193, 195

Microsoft V, VI, VII, 169, 186, 189, 193, 291,

    313, 337

Microsoft Developer Network 189

Microsoft.PowerShell 252

Microsoft.PowerShell.Workflow 252

MicrosoftDNS\_Zone 308

mise en forme 65

MMC VI

mode

    commande 25

    console 91, 97, 119

    scripting 97, 119

module VII, VIII, 161, 162, 169

    binaire 165

    créer 165

    kernel 339, 341

    manifeste 163

    PowerShell 339

    préchargé 162

    script PowerShell 165, 166

Modules 52

Mono 338, 340

motif 280, 282, 284, 285

Mount-Drive 344  
MSDN 189, 204

## N

Name 69, 75, 132  
Navigate 231  
NETBIOS 191  
New-ADUser 295  
New-Alias 29  
New-CimInstance 194  
New-CimSession 194, 195  
New-CimSessionOption 194  
New-Item 221  
New-ItemProperty 221  
New-Module 163  
New-ModuleManifest 163  
New-Object 63, 228, 230, 235, 314, 317, 344  
New-PSDrive 235  
New-PSSession 258, 261  
New-Variable 91  
nommage des paramètres 22  
nonterminating 176

## O

objet VI, VII, 46  
ajouter des membres 64  
comparer 55  
comparer des collections 55  
comparer des propriétés 56  
créer 63  
durée de vie 64  
extraire le contenu d'une propriété 51  
filtrer 59, 60  
grouper 54  
méthode 46  
propriété 46  
sélectionner 49  
sélectionner une propriété 50  
traiter chacun 61, 62  
trier 52  
trier (ordre décroissant) 53  
trier sans doublon 53  
type 46  
utiliser 47

Opened 259  
OpenRemoteBaseKey 223  
OpenSubKey 224  
opérateur VIII, 97  
    arithmétique 98  
    binaire 111  
    comparaison 101  
    d'affectation 104, 122  
    d'appel 114  
    d'égalité 102  
    d'index 117, 123, 125  
    de correspondance 102  
    de déréférencement 114, 125  
    de fractionnement 111  
    de jointure 111, 112  
    de membre statique 115  
    de mise en forme 115  
    de pipeline 116  
    de plage 115, 120, 122  
    de redirection 107, 110  
    de relation 102  
    de sous-expression 116, 131  
    de sous-expression de tableau 116, 121  
    de transtypage 117, 120  
    de type 112  
    dot sourcing 115  
    logique 100  
    précédence 98, 99, 112  
    spécial 114  
    unaire 99, 111  
    virgule 116  
opérateur d'appel 30  
opération d'édition 5  
ordre croissant 52  
ordre décroissant 53  
orienté objet 46  
Out-File 36, 110  
Out-GridView 199  
Out-Gridview 336  
OverwriteAsNeeded 213  
OverwriteOlder 213

## P

package 162

paire clé/valeur 124  
paradigme objet 338  
parallel 272  
param 149, 150  
ParameterSetName 158  
paramètre 12, 20, 22, 30, 37, 38, 39, 40, 41, 43  
    commun aux workflows 276, 277  
    liaison 338  
pipeline VII, 23, 34, 35, 36, 37, 40, 44, 52, 61, 64,  
    93, 116, 150, 151, 158, 160, 172, 175, 277,  
    284, 338  
PipelineIterationInfo 175  
portée 115, 180  
Position 158  
PowerShell Studio 2012 325  
Private: 150  
PROCESS 167  
Process 78  
process 150, 151  
propriété 52  
provider 339  
PS1XML 74  
PSChildName 220  
PSComputerName 256  
PSCustomObject 343  
PsCustomObject 63  
PSSession 15, 257, 261  
    chercher 261  
    connecter 261  
    créer 261  
    déconnecter 261  
PSSnapin 169

**R**

raccourci clavier 9  
Read-Host 134  
Receive-PSSession 258, 263  
Register-WmiEvent 190  
RegistryKey 223  
règle de nommage 146  
Remove-CimSession 194, 196  
Remove-EventLog 214  
Remove-ItemProperty 222  
Remove-KernelModule 341

Remove-Module 163  
Remove-PSSession 258, 260  
Remove-Variable 91  
Remove-WmiObject 190, 193  
RemoveNetworkDrive 235  
Replace 48  
replace 284  
Restart-Service 201, 341  
Resume-Service 202  
rôle de passerelle 14  
root 186

**S**

s'authentifier automatiquement sur un site  
    web 232  
Save-Help 13  
script 6, 18, 20, 91  
    signer 80, 83  
Script: 150  
scriptbloc 146  
ScriptBlock 268  
scriptblock 29  
scripting VII  
SDK 81  
sécurité 77  
    des scripts 21  
    faille 77  
Select-Object 49, 52, 70, 291  
Select-String 286  
Select-Xml 247  
Send 318  
sequence 273  
service 197, 339  
    arrêter 200, 201, 341  
    configurer le mode de démarrage 202  
    démarrer 200, 341  
    gérer 340  
    interrompre 200, 201  
    lister 198  
    lister les dépendances 199  
    modifier le compte de connexion 203  
    redémarrer 200, 201, 341  
    vérifier l'état 341  
service d'annuaire 290

- session persistante 15  
Set-ADAccountPassword 298  
Set-Alias 29  
Set-AuthenticodeSignature 84  
Set-ExecutionPolicy 80  
Set-ItemProperty 223  
Set-Location 217  
Set-Service 202  
Set-Variable 91  
Set-WmiInstance 190  
SetValue 225  
shell V, VI, VII, VIII, 4, 146, 148, 153, 160, 338  
Show-Command 23  
SilentlyContinue 176  
Software Development Kit 81  
Sort-Object 49, 52, 53  
Split 344  
Start-Service 200, 341  
Status 54  
Status-Service 341  
Stop 176  
Stop-Service 201, 341  
stratégie d'exécution 20, 78  
    AllSigned 79  
    Bypass 79  
    connaître la stratégie en cours 80  
    fonctionnement 78  
    modifier 80  
    modifier pour une étendue particulière 80  
    précédence 80  
    RemoteSigned 79  
    Restricted 79  
    type 79  
    Undefined 79  
    Unrestricted 79  
string 150  
Suspend-Service 202  
System 179  
System.Xml.XmlDocument 241  
SYSVOL 305
- T**  
table de hachage 124  
tableau 119, 120
- associatif 124  
créer 120  
position d'index 122  
utiliser 121  
**T**ableaux VIII  
TargetObject 175  
Test-ModuleManifest 163  
ToLower 48  
ToUpper 48, 344  
Trace-Command 40, 41, 44  
trap 178  
trouver une unité d'organisation 298  
try..catch..finally 180
- U**  
Unix/Linux V, 45, 54, 286, 337, 338, 341  
Unmount-Drive 343  
Update-Help 13
- V**  
ValueFromPipeline 158  
ValueFromPipelineByPropertyName 158  
Values 127  
variable VIII, 89, 90, 105, 150  
    automatique 91, 93, 94, 103, 178  
    d'environnement 91  
    de préférence 91, 94, 177  
    définition 90  
    modifiée par l'utilisateur 94  
    utilisateur 91  
VBScript 77, 90, 227  
Visual Studio 325  
volet  
    de console 7, 10  
    de script 7, 10
- W**  
Where-Object 15, 31, 35, 36, 59, 60, 61, 198, 211, 339  
Width 69  
Windows V, VI, VII, 4, 7, 18, 21, 160, 162, 196, 207, 215, 216, 286, 312, 337, 338  
    Server 291  
Windows Forms 324

Windows Management Instrumentation 185,  
186, 227  
Windows Presentation Foundation 324  
Windows Workflow Foundation 265  
WinRM 35, 38, 252  
WMI VIII, 57, 70, 72, 151, 167, 172, 173, 185,  
186, 194, 213, 227, 244, 276, 308  
authentification 191  
cmdlet 190  
documentation 187  
espace de noms 191  
lister les classes 191  
modifier un objet 192  
obtenir une instance 190  
Root\Microsoft 191  
utiliser 189  
WQL 193  
workflow VIII, 15, 265, 266, 267, 275  
définir 267  
opérations séquentielles 273  
paralleliser les opérations 272

syntaxe 272  
vs fonction 271  
WPF 324  
Write-Host 132, 135  
Write-Output 276  
WriteLine 237  
Wscript.Network 233  
Wscript.Shell 228  
WSMan 252  
WWF 265, 267

**X**

XAML 266  
XML VIII, 66, 152, 239  
ajouter un élément 242  
manipuler un fichier XML 240  
rechercher des informations 247  
sérialisation 244

XPath 247

**Z**

zone DNS primaire 309, 310