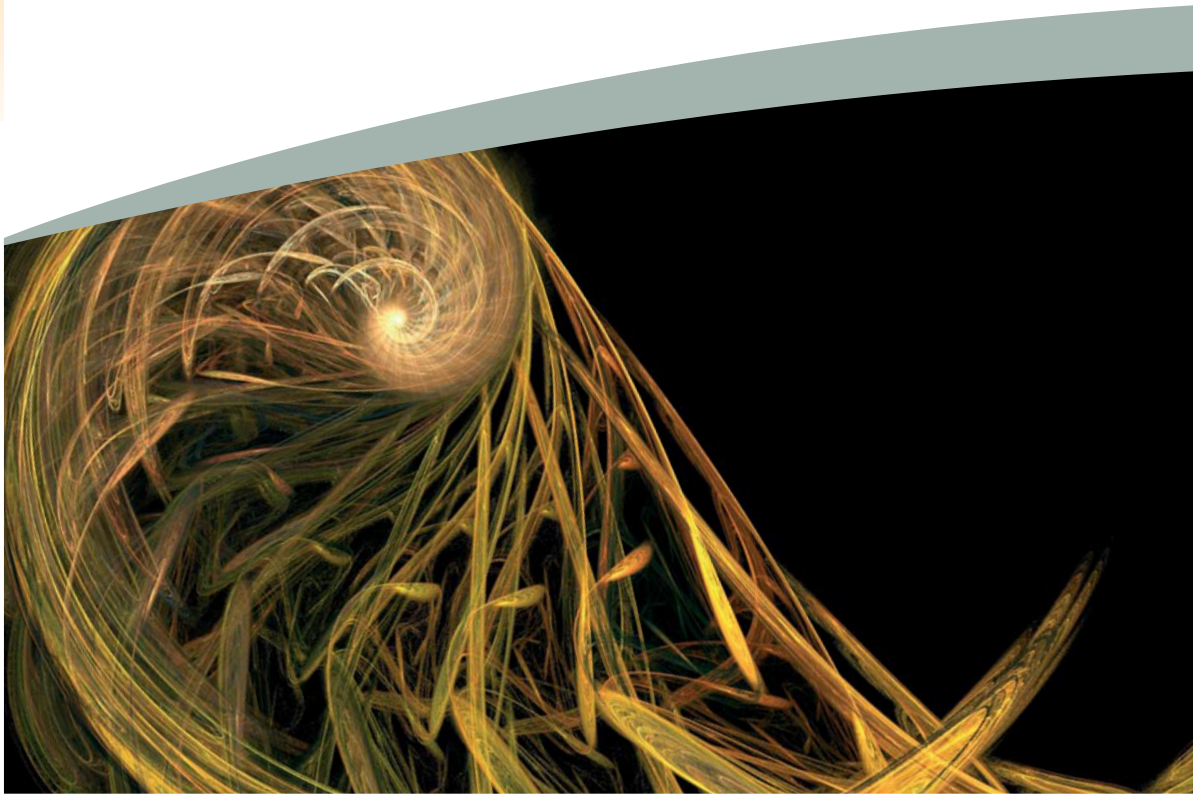
Référence

Windows® PowerShell

Tyson Kopczynski



Réseaux et télécom

Programmation

Génie logiciel

Sécurité

Système d'exploitation

PEARSON

Windows® PowerShell

Tyson Kopczynski

CampusPress

Publié par CampusPress

47 bis, rue des Vinaigriers

75010 PARIS

Tél. : 01 72 74 90 00

Réalisation pao : Léa B.

ISBN : 978-2-7440-4015-3

Titre original : *Windows® PowerShell Unleashed*

Traduction : Hervé Soulard

ISBN original : 978-0-672-32953-1

Table des matières

Partie II. Appliquer ses connaissances à PowerShell

Partie III. Utiliser PowerShell pour les besoins d'automation

À propos de l'auteur

Avec plus de neuf années d'expérience dans le domaine informatique, Tyson Kopczynski est devenu un spécialiste d'Active Directory, des stratégies de groupe, des scripts Windows, de Windows Rights Management Services, de PKI et de la sécurité des technologies de l'information. Il a contribué à l'écriture de livres tels que Microsoft Internet Security and Acceleration (ISA) Server 2004 Unleashed et Microsoft Windows Server 2003 Unleashed (R2 Edition). Par ailleurs, il a écrit plusieurs articles techniques et des guides détaillés sur les différentes technologies qu'il maîtrise. En tant que consultant pour Convergent Computing (CCO), Tyson a pu travailler avec la nouvelle génération de technologies Microsoft depuis leur début et a joué un rôle essentiel dans le développement des pratiques d'écriture de scripts. Tyson est également titulaire de nombreuses certifications en sécurité, dont GIAC Security Essentials Certification (GSEC), Microsoft Certified Systems Engineer (MCSE) Security, CompTIA Security+ et GIAC Certified Incident Handler (GCIH).

Introduction

Lorsque j'ai commencé l'écriture de *Windows PowerShell*, j'étais en train de lire un ouvrage sur l'infrastructure à clé publique (PKI, *Public Key Infrastructure*). Les informations de fond et de référence sur PKI étaient certes très intéressantes mais il manquait des détails sur la mise en application de cette infrastructure dans un environnement réel. En lisant bon nombre de livres techniques, j'ai souvent regretté l'absence de présentation pratique. C'est pourquoi j'ai décidé d'aborder cet ouvrage sur PowerShell de manière différente de la plupart des livres techniques habituels.

Vous lisez les résultats de ce choix. Bien que ce livre contienne des informations de référence détaillées sur PowerShell, j'ai essayé de montrer aux lecteurs comment ils pouvaient employer cet outil pour répondre à leurs besoins précis. Cette approche n'est sans doute pas nouvelle ni révolutionnaire, mais j'espère qu'elle vous apportera une vue unique sur l'un des futurs produits les plus impressionnants de Microsoft.

Cette dernière phrase n'est en aucun cas une publicité pour Microsoft. L'équipe de PowerShell a réellement créé un interpréteur de commandes (*shell*) agréable, simple, amusant et, à n'en pas douter, puissant. Je suis impatient de connaître ce que Microsoft a en réserve pour PowerShell et dans quels produits il sera utilisé.

Notre public

Cet ouvrage est destiné aux administrateurs système de niveau intermédiaire, qui ont investi du temps et de l'énergie à apprendre l'écriture de scripts Windows et qui souhaitent convertir cette compétence en connaissances PowerShell, tout en voyant comment il peut répondre à leurs besoins réels. Il a été écrit afin que quiconque possédant une expérience des scripts puisse comprendre les objectifs de PowerShell et son utilisation, mais il n'en est pas un guide complet. Vous devez le voir comme une ressource permettant de vous apprendre à exploiter PowerShell dans votre propre environnement. Sa structure reflète donc cet objectif en incluant de nombreux exemples de commandes et de scripts opérationnels.

Organisation de ce livre

Cet ouvrage est divisé en trois parties :

- *Partie I, "Introduction à PowerShell"*. Cette partie présente PowerShell et son utilisation, explique pourquoi PowerShell est né, décrit son utilisation générale, détaille la signature de code et établit les meilleures pratiques PowerShell.
- *Partie II, "Appliquer ses connaissances à PowerShell"*. Cette partie explique point à point comment exploiter ses connaissances en écriture de scripts Windows pour apprendre le développement de scripts PowerShell. Elle traite de sujets comme la manipulation du système de fichiers de Windows, le Registre, WMI (*Windows Management Instrumentation*) et ADSI (*Active Directory Services Interfaces*). Pour vous aider, elle propose des exemples de tâches d'automation et des scripts opérationnels, tant en VBScript qu'en PowerShell.
- *Partie III, "Utiliser PowerShell pour les besoins d'automation"*. Cette partie a pour objectif d'aller plus loin sur l'utilisation de PowerShell dans la gestion de systèmes. Elle décrit comment employer PowerShell pour répondre aux besoins de sécurité, automatiser les modifications sur de nombreux systèmes et gérer Exchange Server 2007.

Conventions typographiques

Les commandes, les scripts et tout ce qui a trait au code sont présentés dans une police particulière `à chasse constante`. Le texte en gras indique la définition d'un terme. L'italique est utilisé pour désigner des variables et parfois pour une mise en exergue. Les lettres majuscules et minuscules, les noms et la structure sont utilisés de manière cohérente afin que les exemples de commandes et de scripts soient plus lisibles. Par ailleurs, vous rencontrerez des cas où des commandes ou des scripts n'ont pas été totalement optimisés. Ce choix est volontaire, car il facilite la compréhension de ces exemples et se conforme aux pratiques encourageant une meilleure lisibilité du code. Pour plus de détails sur la présentation, les conventions et les pratiques employées pour les commandes et les scripts dans ce livre, consultez le Chapitre 5, "Suivre les bonnes pratiques".

Voici les autres conventions typographiques de cet ouvrage[1] :

Zones de code en noir

```
Ces zones de code contiennent des commandes à exécuter dans une session
PowerShell ou Bash.
```

Zones de code en gris

```
Ces zones de code contiennent le code source de scripts, de fichiers de
configuration ou d'autres éléments qui ne sont pas exécutés directement
dans une session shell.
```

ATTENTION

Les avertissements signalent des actions à éviter.

INFO

Ces notes proposent des informations supplémentaires sur le sujet en cours.

1. N.d.T. : Dans les zones de code, les césures sont reproduites dans le livre telles qu'elles apparaissent à l'écran. En revanche, nous avons pris le parti de corriger les éventuelles fautes d'orthographe.

I

Introduction à PowerShell

1

Introduction aux shells et à PowerShell

Dans ce chapitre

- Rôle du shell
- Historique des shells
- Arrivée de PowerShell

Les shells, ou interpréteurs de commandes, sont indispensables aux systèmes d'exploitation car ils permettent d'effectuer toutes sortes d'opérations, comme le parcours du système de fichiers, l'exécution de commandes ou le lancement d'applications. Tout utilisateur d'un ordinateur a été confronté au shell, que ce soit en saisissant des commandes à une invite ou en cliquant sur une icône pour faire démarrer une application. Les shells sont incontournables lorsque vous utilisez un système informatique.

Au cours de ce chapitre, nous allons faire connaissance avec le shell et voir tout ce que nous pouvons en tirer. Pour cela, nous examinerons plusieurs commandes de base et nous les combinerons dans un script shell afin d'en augmenter les possibilités. Ensuite, nous présenterons l'évolution des shells depuis ces trente-cinq dernières années. Enfin, nous donnerons les raisons de l'existence de PowerShell et ses implications pour les administrateurs système et les auteurs de scripts.

Rôle du shell

Un **shell** est une interface qui permet aux utilisateurs d'interagir avec le système d'exploitation. Un shell n'est pas considéré comme une application car il est incontournable, mais il équivaut à n'importe quel autre processus s'exécutant sur un système. Le shell diffère d'une application dans la mesure où son rôle est de permettre aux utilisateurs d'exécuter des applications. Dans certains systèmes d'exploitation, comme UNIX, Linux ou VMS, il s'agit d'une interface en ligne de commande (CLI, *Command-line Interface*). Dans d'autres, comme Windows et Mac OS X, il s'agit d'une interface utilisateur graphique.

Par ailleurs, lorsqu'on parle de shells, il est fréquent de négliger deux types de systèmes pourtant très répandus : les équipements réseau et les kiosques. Les dispositifs réseau possèdent généralement un shell graphique (une interface Web sur du matériel grand public) ou un interpréteur de ligne de commande (sur le matériel industriel). Les kiosques sont tout à fait différents. Puisqu'ils sont nombreux à être construits à partir d'applications s'exécutant au-dessus d'un système d'exploitation plus robuste, les interfaces sont rarement des shells. Cependant, si le kiosque s'appuie sur un système d'exploitation dédié, l'interface peut être vue comme un shell. Malheureusement, les interfaces des kiosques sont encore désignées de manière générique comme des shells car il est difficile d'expliquer la différence aux utilisateurs non techniques (ce qui conduit à l'automation des tâches et, par conséquent, à une plus grande efficacité d'exécution ainsi qu'à une meilleure précision et cohérence de réalisation).

Les shells en ligne de commande et les shells graphiques ont des avantages et des inconvénients. Par exemple, la plupart des shells CLI autorisent un enchaînement puissant des commandes (des commandes envoient leur sortie à d'autres pour traitement ; il s'agit d'un fonctionnement en **pipeline** ou **tube**). En revanche, les shells graphiques exigent que les commandes fonctionnent en totale indépendance. Par ailleurs, la plupart des shells graphiques proposent une navigation simple, tandis que les versions CLI supposent la connaissance du système afin de ne pas avoir à tester plusieurs commandes pour mener à bien une tâche d'automation. Votre choix du shell dépend de vos habitudes et de ce qui est le mieux adapté à la réalisation manuelle d'une tâche.

Même si les shells graphiques existent, le terme "shell" est employé presque exclusivement pour décrire un environnement en ligne de commande, non une tâche effectuée avec une application graphique comme l'Explorateur Windows. De même, l'écriture de scripts shell fait référence à l'assemblage de commandes normalement saisies sur la ligne de commande ou dans un fichier exécutable.

Premières utilisations du shell

De nombreuses commandes du shell, comme l'affichage du contenu du répertoire de travail, sont simples. Cependant, les shells peuvent rapidement devenir complexes lorsqu'on souhaite effectuer des traitements plus puissants.

L'exemple suivant affiche le contenu du répertoire de travail :

```
$ ls
apache2 bin    etc    include lib    libexec man    sbin   share  var
```

Très souvent, la simple présentation des noms de fichiers ne suffit pas et il faut donner un argument à la commande pour obtenir plus de détails.

INFO

Si ces commandes ne vous sont pas familières, pas de panique. Nous les donnons uniquement à des fins d'illustration, non pour vous enseigner les complexités du shell Bash.

L'argument passé à la commande suivante permet d'afficher des informations plus détaillées sur chaque fichier :

```
$ ls -l
total 8
drwxr-xr-x    13 root  admin   442 Sep 18 20:50 apache2
drwxrwxr-x    57 root  admin  1938 Sep 19 22:35 bin
drwxrwxr-x     5 root  admin   170 Sep 18 20:50 etc
drwxrwxr-x    30 root  admin  1020 Sep 19 22:30 include
drwxrwxr-x   102 root  admin  3468 Sep 19 22:30 lib
drwxrwxr-x     3 root  admin   102 Sep 18 20:11 libexec
lrwxr-xr-x     1 root  admin     9 Sep 18 20:12 man -> share/man
drwxrwxr-x     3 root  admin   102 Sep 18 20:11 sbin
drwxrwxr-x    13 root  admin   442 Sep 19 22:35 share
drwxrwxr-x     3 root  admin   102 Jul 30 21:05 var
```

Nous devons à présent décider de l'utilisation de ces informations. Comme vous pouvez le constater, les répertoires et les fichiers sont mélangés. Il est donc difficile de les distinguer. Pour n'afficher que les répertoires, nous pouvons filtrer les résultats en envoyant la sortie de la commande `ls` vers la commande `grep`. Dans l'exemple suivant, la sortie a été réduite afin de n'afficher que les lignes commençant par la lettre *d*, qui indique que le fichier est un répertoire (*directory*).

```
$ ls -l | grep '^d'
drwxr-xr-x    13 root  admin   442 Sep 18 20:50 apache2
drwxrwxr-x    57 root  admin  1938 Sep 19 22:35 bin
drwxrwxr-x     5 root  admin   170 Sep 18 20:50 etc
drwxrwxr-x    30 root  admin  1020 Sep 19 22:30 include
drwxrwxr-x   102 root  admin  3468 Sep 19 22:30 lib
drwxrwxr-x     3 root  admin   102 Sep 18 20:11 libexec
drwxrwxr-x     3 root  admin   102 Sep 18 20:11 sbin
drwxrwxr-x    13 root  admin   442 Sep 19 22:35 share
drwxrwxr-x     3 root  admin   102 Jul 30 21:05 var
```

Nous disposons ainsi d'une liste contenant uniquement les répertoires, mais les autres informations, comme la date, les autorisations, la taille, etc., sont superflues car seuls les noms des répertoires nous intéressent. Dans l'exemple suivant, nous utilisons la commande `awk` pour afficher seulement la dernière colonne de la sortie précédente.

```
$ ls -l | grep '^d' | awk '{ print $NF }'
apache2
bin
etc
include
lib
libexec
sbin
share
var
```

Nous obtenons alors une simple liste des sous-répertoires du répertoire de travail. Cette commande est assez directe, mais nous ne voulons pas la saisir chaque fois que nous souhaitons

obtenir une liste de répertoires. Nous créons donc un alias ou un raccourci de commande correspondant à celles que nous venons d'exécuter.

```
$ alias lsd="ls -l | grep '^d' | awk '{ print \$NF }'"
```

Ensuite, grâce à l'alias `lsd`, nous pouvons afficher la liste des sous-répertoires du répertoire en cours sans avoir à saisir l'intégralité de la commande employée dans les exemples précédents.

```
$ lsd
apache2
bin
etc
include
lib
libexec
sbin
share
var
```

Vous pouvez ainsi le deviner, un shell en ligne de commande ouvre de grandes possibilités pour l'automation de simples tâches répétitives.

Premiers scripts shell

L'utilisation d'un shell consiste à saisir chaque commande, à interpréter la sortie, à choisir l'usage de ces données et à combiner les commandes en un seul processus dépouillé. Quiconque a déjà examiné des dizaines de fichiers et ajouté manuellement une seule ligne à la fin de chacun d'eux conviendra que les scripts sont aussi vitaux que l'air que nous respirons.

Nous avons vu comment enchaîner des commandes dans un pipeline pour manipuler la sortie de la commande précédente et comment créer un alias pour réduire la saisie. Les alias sont les petits frères des scripts shell et apportent à la ligne de commande une part de la puissance des scripts, qui est sans commune mesure avec celle des alias.

Réunir des commandes d'une ligne et des pipelines dans des fichiers pour pouvoir les exécuter ultérieurement est une technique puissante. Placer une sortie dans des variables pour y faire référence ensuite dans le script et la soumettre à différentes manipulations nous fait passer au niveau de puissance supérieure. Embarquer des combinaisons de commandes dans des boucles récursives et des constructions de contrôle du flux fait de l'écriture de scripts une forme de programmation.

Certains diront que écriture de scripts et programmation sont deux choses différentes. Mais c'est faux, en particulier si l'on considère la diversité et la puissance des langages de scripts actuels. En ce sens, l'écriture de scripts n'est pas différente, tout comme la compilation du code ne signifie pas nécessairement que l'on programme. En gardant cela à l'esprit, essayons de développer notre commande d'une ligne de la section précédente en quelque chose de plus utile.

Nous disposons d'une liste des sous-répertoires du répertoire de travail. Supposons que nous voulions écrire un outil qui affiche l'espace disque occupé par chaque répertoire. La commande `bash`, qui affiche l'utilisation de l'espace disque, opère sur l'intégralité du contenu du répertoire indiqué ou sur la globalité d'un répertoire dans un récapitulatif ; elle donne également la quantité en octets, par défaut. Si nous souhaitons connaître l'espace disque occupé par chaque répertoire en tant qu'entité autonome, nous devons obtenir et afficher les informations pour chaque répertoire, un par un. Les exemples suivants montrent comment programmer ce comportement dans un script.

Rappelez-vous la commande écrite à la section précédente. La boucle `for` suivante prend chaque répertoire indiqué dans la liste retournée par cette commande, l'affecte à la variable `DIR` et exécute le code qui se trouve entre les mots clés `do` et `done`.

```
#!/bin/bash

for DIR in $(ls -l | grep '^d' | awk '{ print $NF }'); do
    du -sk ${DIR}
done
```

Nous enregistrons ce code dans le fichier de script `big_directory.sh`. L'exécution de ce script dans une session Bash produit la sortie suivante.

```
$ big_directory.sh
17988   apache2
5900    bin
72      etc
2652    include
82264   lib
0       libexec
0       sbin
35648   share
166768  var
```

Cette sortie ne semble pas particulièrement intéressante. En ajoutant quelques instructions, nous pouvons écrire un traitement utile, par exemple pour connaître les noms des répertoires qui occupent plus d'une certaine quantité d'espace disque. Pour cela, modifiez le fichier `big_directory.sh` de la manière suivante.

```
#!/bin/bash

PRINT_DIR_MIN=35000

for DIR in $(ls -l | grep ,^d' | awk ,{ print $NF }'); do
    DIR_SIZE=$(du -sk ${DIR} | cut -f 1)
    if [ ${DIR_SIZE} -ge ${PRINT_DIR_MIN} ];then
        echo ${DIR}
    fi
done
```

Nous avons ajouté des variables. `PRINT_DIR_MIN` précise le nombre minimal de kilo-octets qu'un répertoire doit occuper pour mériter d'être affiché. Puisque cette valeur peut être changée relativement souvent, elle doit pouvoir l'être facilement. De même, nous pouvons la réutiliser ailleurs dans le script afin d'éviter d'avoir à la modifier en de multiples endroits.

Vous pourriez penser que la commande find pourrait avantageusement remplacer ce script. Cependant, nous utilisons cette commande `ls` compliquée car, si find est parfaitement adaptée au parcours des structures hiérarchiques, elle est trop lourde pour le simple affichage du répertoire courant. Si vous recherchez des fichiers dans une arborescence, nous vous conseillons fortement la commande find. Mais, nous examinons simplement les sous-répertoires du répertoire de travail car, dans cet exemple, ils sont les seuls pertinents.

Voici un exemple de sortie obtenue par notre script.

```
$ big_directory.sh
lib
share
var
```

Nous pouvons l'exploiter de différentes manières. Par exemple, les administrateurs système pourraient se servir de ce script pour surveiller l'occupation disque des répertoires des utilisateurs et leur envoyer un message lorsqu'ils dépassent un certain seuil. Pour cela, il peut être intéressant de savoir lorsqu'un certain pourcentage des utilisateurs atteint ou dépasse le seuil.

INFO

Vous devez savoir qu'il existe aujourd'hui sur le marché de nombreux produits commerciaux qui avertissent les administrateurs lorsque des seuils d'occupation des disques sont dépassés. Par conséquent, même si vous pouvez économiser de l'argent en écrivant un script shell pour surveiller l'utilisation générale des disques, ce n'est pas une obligation. Déterminer les utilisateurs qui ont atteint un certain seuil est une tâche différente, car cela implique des mesures proactives afin de prévenir des problèmes avant qu'ils ne deviennent hors de contrôle. La solution consiste à avertir l'administrateur que certains utilisateurs doivent être déplacés sur de nouveaux disques en raison de l'espace qu'ils occupent sur les disques actuels. Cette méthode n'est pas à toute épreuve, mais elle permet d'ajouter facilement une couche de surveillance proactive qui évite que les utilisateurs soient confrontés à des problèmes sur leur machine. Les administrateurs système peuvent faire preuve d'imagination et modifier ce script en ajoutant des paramètres de commande afin d'indiquer la procédure à réaliser, comme afficher les utilisateurs les plus gourmands en espace disque, et signaler lorsqu'un certain pourcentage d'utilisateurs a atteint le seuil critique. Cependant, ce type d'extension sort du propos de ce chapitre.

Nous modifions ensuite le script pour afficher un message lorsqu'un certain pourcentage des répertoires est de la taille précisée.

```
#!/bin/bash

DIR_MIN_SIZE=35000
DIR_PERCENT_BIG_MAX=23

DIR_COUNTER=0
BIG_DIR_COUNTER=0

for DIR in $(ls -l | grep ,^d' | awk ,{ print $NF }'); do
    DIR_COUNTER=$(expr ${DIR_COUNTER} + 1)
    DIR_SIZE=$(du -sk ${DIR} | cut -f 1)
    if [ ${DIR_SIZE} -ge ${DIR_MIN_SIZE} ];then
        BIG_DIR_COUNTER=$(expr ${BIG_DIR_COUNTER} + 1)
    fi
done

if [ ${BIG_DIR_COUNTER} -gt 0 ]; then
    DIR_PERCENT_BIG=$(expr $(expr ${BIG_DIR_COUNTER} \* 100) / ${DIR_COUNTER})
    if [ ${DIR_PERCENT_BIG} -gt ${DIR_PERCENT_BIG_MAX} ]; then
        echo "${DIR_PERCENT_BIG} pourcent des répertoires occupent plus de
        ${DIR_MIN_SIZE} kilo-octets."
    fi
fi
```

L'exemple précédent ressemble peu à notre script initial. La variable `PRINT_DIR_MIN` est devenue `DIR_MIN_SIZE`, car nous n'affichons plus directement les répertoires qui atteignent la taille minimale. La variable `DIR_PERCENT_BIG_MAX` a été ajoutée. Elle indique le pourcentage maximal autorisé de répertoires dont la taille est égale ou supérieure à la taille minimale. Par ailleurs, deux compteurs sont utilisés : le premier (`DIR_COUNTER`) compte le nombre de répertoires et le second (`BIG_DIR_COUNTER`) le nombre de répertoires qui dépassent la taille minimale.

À l'intérieur de la boucle, `DIR_COUNTER` est incrémenté et l'instruction `if` de la boucle `for` incrémente à présent uniquement `BIG_DIR_COUNTER` au lieu d'afficher le nom du répertoire.

Une instruction `if` a été ajoutée après la boucle `for` pour procéder au calcul du pourcentage des répertoires dépassant la taille minimale, puis afficher le message si nécessaire. Suite à ces modifications, le script produit la sortie suivante :

```
$ big_directory.sh
33 pourcent des répertoires occupent plus de 35000 kilo-octets.
```

Elle signale que 33 % des répertoires ont une taille supérieure ou égale à 35 Mo. En modifiant la ligne `echo` du script de manière à alimenter un pipeline vers une commande d'envoi de courrier et en ajustant les seuils de taille et de pourcentage à leur environnement, les administrateurs système peuvent planifier l'exécution de ce script à différents moments et générer facilement des rapports d'occupation des répertoires. S'ils souhaitent aller plus loin, ils peuvent ajouter des paramètres au script afin de préciser les seuils de taille et de pourcentage.

Vous pouvez le constater, même un script shell de base peut être puissant. Avec une vingtaine de lignes de code, nous avons créé un script très utile. Certaines bizarreries pourraient sembler gênantes (par exemple, l'utilisation de la commande `expr` pour réaliser des calculs mathématiques simples), mais tout langage de programmation a ses forces et ses faiblesses. De manière générale, certaines des tâches sont plus complexes à accomplir que d'autres, quel que soit le langage choisi.

De cette introduction, vous devez retenir que l'écriture de scripts permet de vous simplifier la vie. Par exemple, supposons la fusion de deux sociétés. Au cours de cette opération, il est nécessaire de créer mille comptes d'utilisateurs dans Active Directory, ou n'importe quel autre système d'authentification. En général, un administrateur système prend la liste des utilisateurs, s'assoit devant son clavier avec une tasse de café et commence à cliquer ou à saisir les données. S'il dispose du budget nécessaire, il peut embaucher une personne pour effectuer le travail ou acheter un logiciel de migration. Mais pourquoi effectuer des tâches répétitives ou dépenser de l'argent qui pourrait être mieux utilisé (pour un meilleur salaire) ?

À la place de cette méthode, il est préférable d'automatiser ces tâches en utilisant des scripts. L'automation constitue la finalité de l'écriture des scripts. En tant qu'administrateur système, pour disposer des mêmes fonctionnalités que les développeurs lorsqu'ils codent les systèmes que vous administrez, vous devez exploiter les scripts avec des shells en ligne de commande ou des interpréteurs de commandes. Cependant, l'écriture de scripts est un domaine qui a tendance à être plus ouvert, plus flexible et plus ciblé sur les tâches que vous, en tant que professionnel des systèmes d'information, devez accomplir.

Historique des shells

Le premier interpréteur de commandes largement utilisé a été le shell Bourne. Il s'agit de l'interface utilisateur standard sur le système d'exploitation UNIX et ces systèmes en ont encore besoin pour la phase d'initialisation. Ce shell robuste dispose des pipelines et de l'exécution conditionnelle et récursive des commandes. Il a été développé par des programmeurs C pour des programmeurs C.

Cependant, bien qu'il ait été écrit par et pour des programmeurs C, le shell Bourne n'utilise pas un style d'écriture analogue à ce langage. Cette absence de similitude avec le langage C a conduit à la création du shell C, lequel dispose des structures de programmation plus conformes à C. Pendant qu'ils écrivaient un "meilleur" shell, les développeurs ont décidé d'ajouter l'édition de la ligne de commande et les alias de commandes (des raccourcis) pour répondre à la hantise de tout utilisateur d'UNIX, c'est-à-dire la saisie des commandes. Moins il a de caractères à saisir, plus l'utilisateur d'UNIX est heureux.

La majorité des utilisateurs d'UNIX a apprécié le shell C, mais, néanmoins, l'apprentissage d'un shell totalement nouveau s'est révélé un véritable défi pour certains. C'est pourquoi le shell Korn a été inventé. Il a ajouté certaines fonctionnalités du shell C au shell Bourne. Mais, puisque ce shell Korn était un produit commercial, la communauté des logiciels open source avait besoin d'un autre shell pour Linux et FreeBSD. C'est ainsi que le Bourne Again Shell, ou Bash, a été proposé par la FSF (*Free Software Foundation*).

Outre le développement d'UNIX et la naissance de Linux et de FreeBSD, d'autres systèmes d'exploitation sont arrivés, avec leur propre shell. DEC (*Digital Equipment Corporation*) a sorti VMS (*Virtual Memory System*) pour concurrencer UNIX sur ses systèmes VAX. Le shell de VMS se nommait DCL (*Digital Command Language*) et employait une syntaxe verbeuse, contrairement à ses homologues UNIX. Par ailleurs, il n'était pas sensible à la casse et ne disposait pas des pipelines.

Puis, à un moment donné, le PC est né. IBM a positionné cette machine sur le marché des entreprises et Apple a repris une technologie matérielle similaire avec les consommateurs pour cible. Microsoft a créé DOS pour l'IBM PC, en lui donnant le rôle de noyau et de shell et en le dotant de certaines fonctionnalités des autres shells (la syntaxe des pipelines vient d'UNIX).

Après DOS est venu Windows, avec un shell graphique qui a depuis servi de base aux shells Microsoft. Malheureusement, les shells graphiques ont la réputation d'être difficiles à scripter et Windows a donc proposé un environnement de type DOS. Il a été amélioré par un nouvel exécutable, cmd.exe à la place de command.com, et par des fonctionnalités d'édition

de la ligne de commande plus robustes. Cependant, cette évolution signifiait également que les scripts shell de Windows devaient être réécrits avec la syntaxe de DOSShell pour réunir et exécuter des groupes de commandes.

Microsoft a pris conscience de son erreur et a décidé que les administrateurs système devaient disposer d'outils plus élaborés pour gérer les systèmes Windows. WSH (*Windows Script Host*) est arrivé avec Windows 98 et a donné aux natifs un accès aux composants sous-jacents de Windows. Il s'agissait d'une bibliothèque permettant aux langages de scripts d'utiliser Windows de manière efficace et puissante. WSH n'étant pas un langage en soi, il fallait un langage de scripts compatible pour l'exploiter, comme par exemple JScript, VBScript, Perl, Python, Kixstart ou Object REXX. Certains de ces langages sont bien adaptés à la mise en œuvre de processus complexes et WSH était donc une bénédiction pour les administrateurs de systèmes Windows.

Pourtant, les réjouissances ont été de courte durée, car rien ne garantissait que le langage de scripts compatible WSH choisi par l'administrateur serait disponible ou une solution viable pour tout le monde. L'absence de langage et d'environnement standard pour l'écriture de scripts ne permettait pas aux utilisateurs et aux administrateurs d'inclure facilement une procédure d'automation avec WSH. La seule manière d'être certain que le langage de scripts ou la version de WSH était compatible avec le système administré consistait à utiliser un langage de scripts natif, c'est-à-dire DOSShell, et à accepter les problèmes y afférents. Par ailleurs, WSH ouvrait une porte aux attaques par code malveillant. Cette vulnérabilité a donné naissance à un flot de virus, de vers et autres programmes indélicats qui ont causé de nombreux dégâts sur les systèmes informatiques, et cela à cause de l'accent mis sur l'automation sans intervention de l'utilisateur.

Au final, les administrateurs système ont vu en WSH autant une bénédiction qu'une malédiction. Même si WSH arborait un bon modèle d'objets et donnait accès à de nombreuses interfaces d'automation, il ne s'agissait pas d'un shell. Il exigeait l'utilisation de `Wscript.exe` et de `Cscript.exe`, les scripts devaient être écrits dans un langage compatible et ses vulnérabilités ont représenté un défi de sécurité. Très clairement, une approche différente était nécessaire pour l'administration des systèmes. Microsoft est arrivé à la même conclusion.

Arrivée de PowerShell

Microsoft n'a pas consacré beaucoup d'efforts à un shell en ligne de commande. À la place, il s'est concentré sur un shell graphique, plus compatible avec ses systèmes d'exploitation graphiques. (Mac OS X n'a pas non plus beaucoup travaillé sur un shell en ligne de commande ;

il utilise le shell Bash.) Cependant, le DOSShell résultant présentait diverses limites, comme les structures de programmation conditionnelle et récursive peu documentées et une utilisation importante des instructions `goto`. Pendant des années, tous ces points ont gêné les auteurs de scripts shell, qui se sont tournés vers d'autres langages de scripts ou ont écrit des programmes compilés pour résoudre les problèmes classiques.

La standardisation de WSH dans les systèmes d'exploitation Windows a offert une alternative robuste aux scripts DOSShell. Malheureusement, cet outil présentait de nombreux défauts, comme l'a expliqué la section précédente. D'autre part, WSH ne permettait pas une utilisation en ligne de commande telle que la connaissaient depuis des années les administrateurs UNIX et Linux. À cause de l'absence d'un shell en ligne de commande et des avantages qu'il procure, les administrateurs Windows ont été raillés par leurs collègues du monde UNIX.

Heureusement, Jeffrey Snover (l'architecte de PowerShell) et les autres membres de l'équipe de développement de PowerShell ont compris que Windows avait besoin d'un shell en ligne de commande robuste et sûr pour l'administration des systèmes. PowerShell a été conçu comme un shell, avec un accès complet aux composants de Windows, par l'intermédiaire de .NET Framework, des objets COM (*Component Object Model*) et d'autres méthodes. Il offre également un environnement d'exécution familier, simple et sécurisé. Le nom PowerShell est parfaitement choisi, en ce sens qu'il augmente la puissance du shell de Windows. Pour les utilisateurs qui souhaitent automatiser leurs systèmes Windows, l'arrivée de PowerShell est très excitante car il combine "la puissance de WSH et la familiarité d'un shell".

Puisque PowerShell fournit un langage de scripts natif, les scripts peuvent être placés sur tous les systèmes Windows sans avoir à s'inquiéter de l'installation d'un interpréteur de langage particulier. Vous avez peut-être appris la syntaxe imposée par l'emploi de WSH dans Perl, Python, VBScript, JScript ou tout autre langage, pour vous apercevoir que le système sur lequel vous travaillez à présent ne dispose pas de l'interpréteur adéquat. Chez eux, les utilisateurs peuvent installer tout ce qu'ils souhaitent sur leur système et assurer sa maintenance comme bon leur semble. En revanche, dans une entreprise, cette situation n'est pas toujours viable. PowerShell résout ce problème en n'imposant pas la présence d'interpréteurs non natifs. Par ailleurs, il évite de parcourir les sites Web à la recherche d'équivalents en ligne de commande aux opérations simples du shell graphique et à les coder dans des fichiers `.cmd`. Enfin, PowerShell s'attaque au problème de sécurité de WSH, en fournissant une plate-forme d'écriture de scripts Windows sûrs. Il se concentre sur les fonctionnalités de sécurité comme la signature des scripts, l'absence d'extensions d'exécutables et les politiques d'exécution (restreintes par défaut).

Pour quiconque a besoin d'automatiser des tâches d'administration sur un système Windows, PowerShell apporte une puissance bienvenue. Sa nature orientée objet augmente également ses possibilités. Si vous êtes un administrateur de systèmes Windows ou un développeur de scripts, nous vous conseillons fortement de devenir un expert PowerShell.

PowerShell n'est pas un projet secondaire chez Microsoft. L'équipe de développement a réussi à créer un shell stupéfiant et à gagner un soutien au sein de Microsoft. Par exemple, l'équipe d'Exchange a adopté PowerShell comme support de l'interface de gestion d'Exchange Server 2007. Et ce n'est que le début. D'autres produits de Microsoft, comme System Center Operations Manager 2007, System Center Data Protection Manager V2 et System Center Virtual Machine Manager, vont bénéficier de tous les avantages de PowerShell.

En réalité, PowerShell constitue la solution que Microsoft recherchait pour une interface générale de gestion des systèmes Windows. Dans le futur, PowerShell pourrait remplacer les interfaces actuelles, comme `cmd.exe`, WSH, les outils en ligne de commande, etc., et être totalement intégré aux systèmes Windows. Avec la création de PowerShell, Microsoft a répondu aux besoins d'un shell Windows en ligne de commande. Les administrateurs de systèmes Windows et les développeurs de scripts n'ont plus que leur imagination comme seule limite.

En résumé

Ce chapitre a introduit la notion de shell, l'origine des shells, comment les utiliser et comment créer un script shell de base. En étudiant ces aspects, vous devez avoir compris pourquoi l'écriture de scripts est aussi importante pour les administrateurs système. Comme vous l'avez peut-être découvert, les scripts leur permettent d'automatiser les tâches répétitives. Ils peuvent ainsi travailler plus efficacement et passer plus de temps sur des tâches plus importantes.

D'autre part, ce chapitre a présenté PowerShell et les raisons de son existence. Ainsi, PowerShell est le remplaçant de WSH, qui, malgré sa puissance, affichait plusieurs défauts (les plus notables ayant trait à la sécurité et l'interopérabilité). PowerShell était également indispensable car il manquait à Windows un shell en ligne de commande digne de ce nom, en mesure d'effectuer facilement des tâches d'automation complexes. Pour remplacer WSH et améliorer l'interpréteur en ligne de commande de Windows, Microsoft a créé PowerShell. Il s'appuie sur .NET Framework et apporte le support tant attendu aux scripts et à l'automation sous Windows.

2

Les fondamentaux de PowerShell

Dans ce chapitre

- Introduction
- Avant de commencer
- Accéder à PowerShell
- Comprendre l'interface en ligne de commande
- Comprendre les applets de commande
- Quelques applets de commande utiles
- Expressions
- Comprendre les variables
- Comprendre les alias
- Séquences d'échappement
- Comprendre les portées
- Premier script

Introduction

Ce chapitre va nous emmener directement dans les bases techniques de PowerShell et de son utilisation. Nous téléchargerons et installerons PowerShell, travaillerons avec son interface en ligne de commande (CLI, *Command-line Interface*), utiliserons des applets de commande, définirons des variables, créerons des alias, examinerons les portées et écrirons un premier script. Il ne s'agit pas d'un guide de démarrage complet mais ce chapitre s'intéresse aux concepts importants que vous devez comprendre avant d'aborder les autres.

Avant de commencer

La meilleure manière de débuter avec PowerShell consiste à visiter sa page Web à l'adresse **www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx** (voir Figure 2.1).

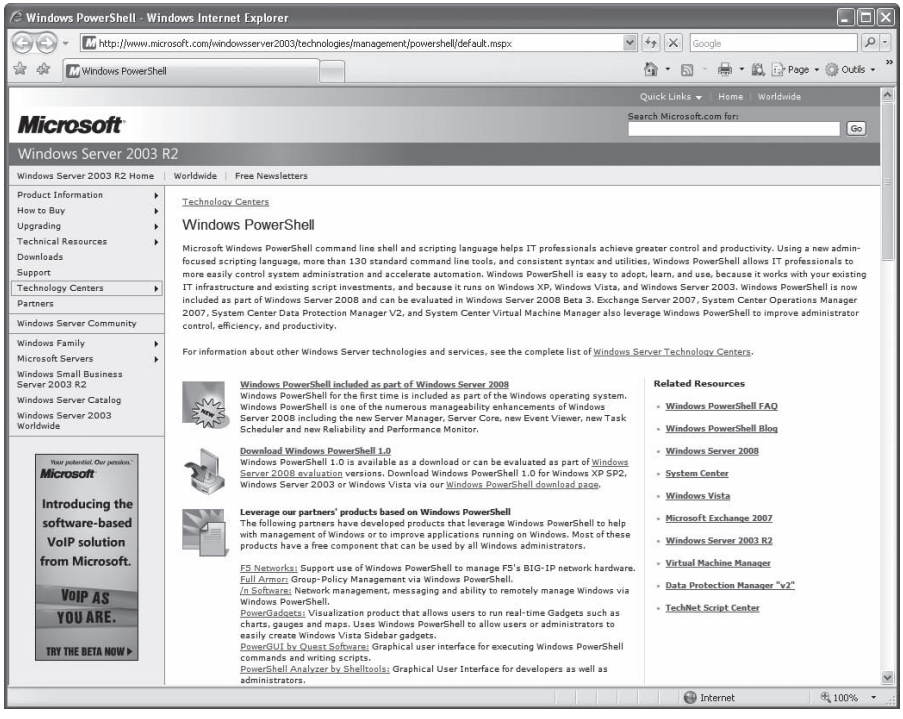


Figure 2.1

La page d'accueil de PowerShell sur le site de Microsoft.

À partir de cette page, nous pouvons obtenir des informations sur PowerShell, télécharger de la documentation et des outils, ainsi qu'obtenir les dernières nouvelles et les dernières versions de PowerShell. Nous allons télécharger et installer PowerShell, mais vous devez commencer par vérifier que la configuration de votre système respecte les contraintes suivantes :

- Windows XP Service Pack 2, Windows 2003 Service Pack 1 ou une version ultérieure de Windows ;
- Microsoft .NET Framework 2.0.

Si .NET Framework 2.0 n'est pas installé sur votre machine, vous pouvez télécharger son module d'installation depuis le centre de téléchargement de Microsoft à l'adresse **http://www.microsoft.com/downloads/Search.aspx?displaylang=fr** (voir Figure 2.2).

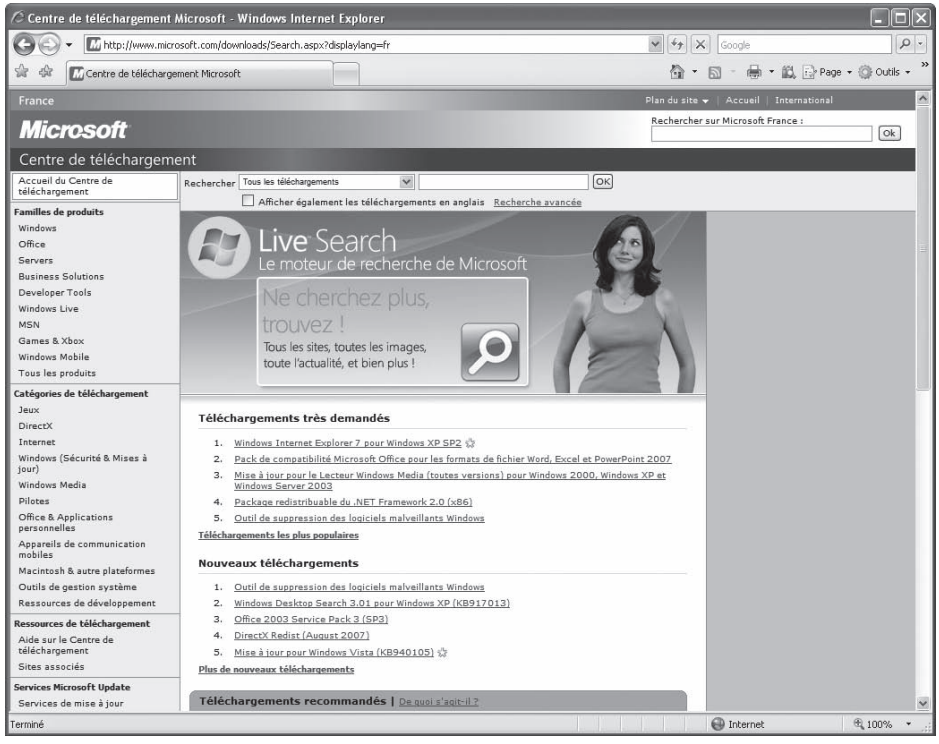


Figure 2.2

Le Centre de téléchargement de Microsoft.

Après avoir installé .NET Framework 2.0, vous devez télécharger le module d'installation de PowerShell à partir de **www.microsoft.com/windowsserver2003/technologies/management/powershell/download.mspx** (voir Figure 2.3).

Sur cette page, choisissez le module correspondant à votre version de Windows, en mode x86 ou x64. Cliquez sur le lien intitulé "Localized Package" pour sélectionner la version française de PowerShell. Ensuite, lancez l'installation de PowerShell en cliquant sur Ouvrir dans la boîte de téléchargement ou en double-cliquant sur le fichier téléchargé. (Le nom du fichier diffère selon la plate-forme, la version de Windows et la langue.) Après le démarrage de l'installation, suivez les instructions d'installation affichées.

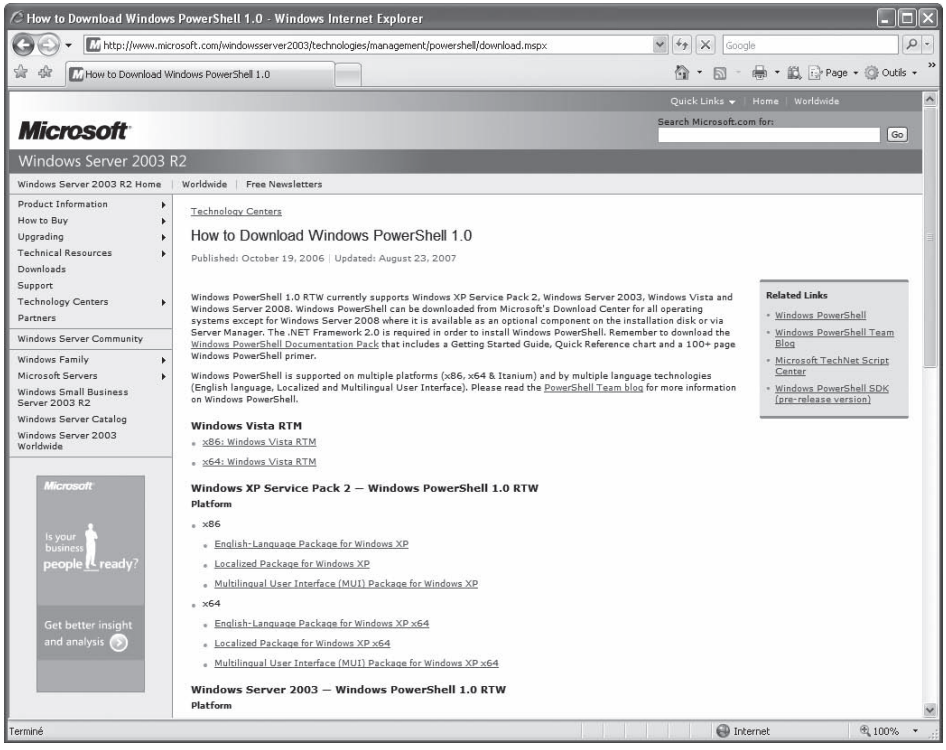


Figure 2.3
Télécharger Windows PowerShell 1.0.

Vous pouvez également installer PowerShell en mode silencieux depuis la ligne de commande, en passant l'option `/quiet` au fichier d'installation. Cette méthode peut être utile si vous prévoyez d'installer PowerShell sur plusieurs systèmes différents et souhaitez effectuer l'installation par un script d'ouverture de session, par SMS (*Systems Management Server*) ou tout autre logiciel d'administration. Pour effectuer une installation silencieuse, procédez comme suit :

1. Cliquez sur Démarrer > Exécuter.
2. Saisissez **cmd**, puis cliquez sur OK pour ouvrir une invite de commande.
3. Saisissez ***Nom-du-fichier-d'installation-de-PowerShell* /quiet** (en remplaçant le texte en italique par le nom du fichier d'installation que vous avez téléchargé) et appuyez sur Entrée.

Accéder à PowerShell

Après avoir installé PowerShell, vous disposez de trois méthodes pour y accéder. Pour employer la première, utilisez le menu Démarrer :

1. Cliquez sur Démarrer > Tous les programmes > Windows PowerShell 1.0.
2. Cliquez sur Windows PowerShell.

Voici les étapes de la deuxième méthode :

1. Cliquez sur Démarrer > Exécuter.
2. Saisissez **PowerShell** dans la boîte de dialogue Exécuter et cliquez sur OK.

Ces deux méthodes ouvrent la console PowerShell présentée à la Figure 2.4.

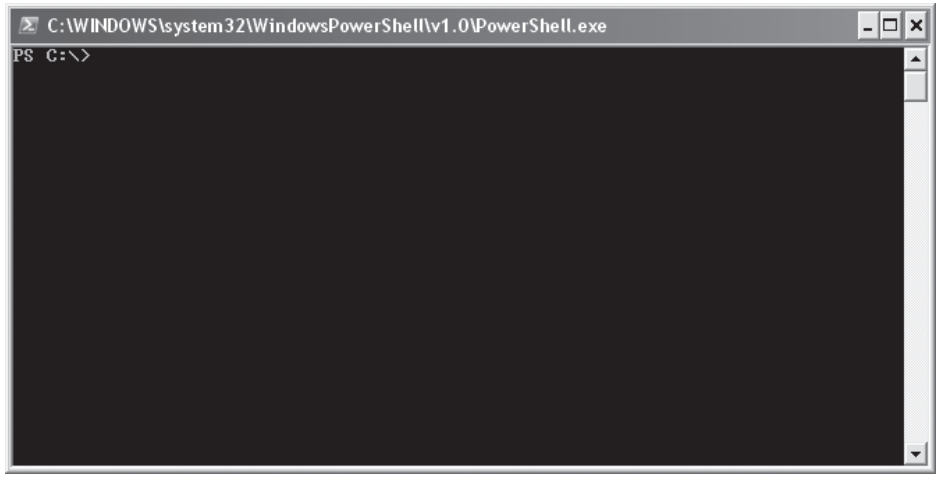


Figure 2.4
La console de PowerShell.

Les étapes de la troisième méthode répondent à une invite de commande :

1. Cliquez sur Démarrer > Exécuter.
2. Saisissez **cmd**, puis cliquez sur OK pour ouvrir une invite de commande.
3. À l'invite de commande, saisissez **powershell** (voir Figure 2.5) et appuyez sur Entrée.

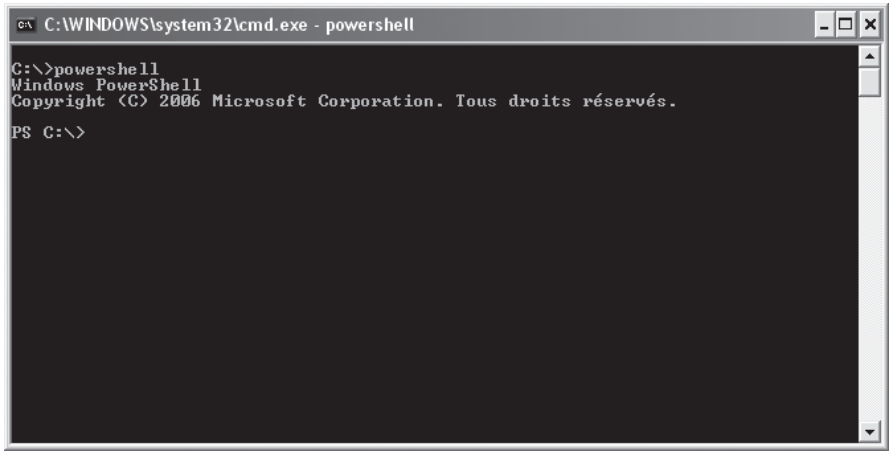


Figure 2.5
Ouverture de la console PowerShell depuis l'invite de commande.

Comprendre l'interface en ligne de commande

La syntaxe d'utilisation de PowerShell en ligne de commande est analogue à celle des autres shells dans le même mode. L'élément fondamental d'une commande PowerShell est, bien entendu, son nom. Par ailleurs, une commande peut être rendue plus spécifique en utilisant des paramètres et leurs arguments. Voici les formats d'une commande PowerShell :

```
[nom de commande]
[nom de commande] -[paramètre]
[nom de commande] -[paramètre] –[paramètre] [argument1]
[nom de commande] -[paramètre] –[paramètre] [argument1],[argument2]
```

INFO

Dans PowerShell, un paramètre est une variable acceptée par une commande, un script ou une fonction. Un argument est une valeur affectée à un paramètre. Bien que ces termes soient souvent employés de manière interchangeable, n'oubliez pas ces définitions dans le contexte PowerShell.

Voici un exemple d'utilisation d'une commande, d'un paramètre et d'un argument. Il s'agit de la commande `dir`, avec le paramètre `/w` (qui affiche la sortie de `dir` dans un format large) et l'argument `C:\temp\*.txt` :

```
C:\>dir /w C:\temp*.txt
 Le volume dans le lecteur C s'appelle OS
 Le numéro de série du volume est 1784-ADF9

 Répertoire de C:\temp

Mémo Eric.txt   mediapc.txt     note.txt        Version1.txt
               4 fichier(s)          953 octets
               0 Rép(s)  16 789 958 656 octets libres

C:\>
```

Le résultat de cette commande est une liste, dans un format large, de tous les fichiers `.txt` contenus dans le répertoire `C:\temp`. Si vous invoquez la commande `dir` sans paramètres ni arguments, le résultat est totalement différent. C'est également ainsi que fonctionne PowerShell. Par exemple, voici une commande PowerShell qui affiche des informations concernant le processus d'`explorer.exe` :

```
PS C:\> get-process -Name explorer

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    807      20    31672      14068   149    62,95   1280 explorer


PS C:\>
```

Dans cet exemple, `Get-Process` est la commande, `-Name` est le paramètre et `explorer` est l'argument. Cette commande affiche les informations concernant le processus d'`explorer.exe`. Si aucun paramètre ni argument n'est utilisé, elle affiche simplement les informations concernant tous les processus en cours et non un seul. Pour modifier le comportement d'une commande ou lui demander d'effectuer autre chose que son opération par défaut, il faut comprendre sa syntaxe. La commande `Get-Help`, décrite à la section "Quelques applets de commandes utiles", page 41, fournit des informations détaillées sur le rôle d'une commande et son utilisation.

Utiliser la ligne de commande

Comme pour tout shell en ligne de commande, nous devons apprendre à nous déplacer dans la ligne de commande de PowerShell pour l'utiliser efficacement. Le Tableau 2.1 donne la liste des opérations d'édition associées aux différentes touches disponibles dans la console PowerShell.

Tableau 2.1 Fonctions d'édition de la console de PowerShell

Touches	Opération d'édition
Flèches vers la gauche et vers la droite	Déplacent le curseur vers la gauche et vers la droite sur la ligne de commande en cours.
Flèches vers le haut et vers le bas	Parcourent la liste des dernières commandes saisies, vers le début et la fin.
Insér.	Bascule en mode insertion et remplacement du texte.
Suppr.	Supprime le caractère à la position courante du curseur.
Espace arrière	Supprime le caractère qui se trouve juste avant la position courante du curseur.
F7	Affiche la liste des dernières commandes saisies dans une fenêtre, par-dessus le shell de commande. Les touches Flèche vers le haut et Flèche vers le bas permettent de sélectionner l'une des commandes. La touche Entrée exécute la commande sélectionnée.
Tab	Complète automatiquement les éléments de la ligne de commande. La combinaison Maj+Tab permet de revenir en arrière dans la liste des correspondances possibles.

Heureusement, la plupart des opérations mentionnées au Tableau 2.1 existent déjà dans l'interpréteur de commande de Windows, ce qui permet aux administrateurs système d'adopter plus facilement PowerShell. La principale différence réside dans la touche Tab pour la complétion automatique. Elle a été étendue dans PowerShell.

Comme l'interpréteur de commande Windows, PowerShell complète automatiquement les noms de fichiers et de répertoires. Si nous saisissons un nom partiel et appuyons sur la touche Tab, PowerShell retourne le premier nom de fichier ou de répertoire correspondant dans le répertoire de travail. Si nous appuyons à nouveau sur Tab, nous obtenons la deuxième correspondance possible. Les appuis répétés sur Tab parcourent la liste des résultats.

La complétion automatique peut également tenir compte des caractères génériques, comme le montre l'exemple suivant :

```
PS C:\> cd C:\Doc*
<tab>
PS C:\> cd 'C:\Documents and Settings'
PS C:\Documents and Settings>
```

La complétion automatique de PowerShell concerne également les commandes. Par exemple, nous pouvons saisir un nom de commande partiel et appuyer sur la touche Tab afin que PowerShell parcoure la liste des commandes possibles :

```
PS C:\> get-pro
<tab>
PS C:\> get-process
```

PowerShell complète également les noms de paramètres associés à une commande. Il suffit de saisir une commande et un nom de paramètre partiel, puis d'appuyer sur la touche Tab. PowerShell itère alors sur les paramètres de la commande précisée. Cette méthode s'applique également aux variables associées à une commande. Par ailleurs, PowerShell effectue une complétion automatique des méthodes et des propriétés des variables et des objets. Prenons l'exemple d'une variable nommée `$Z` et fixée à la valeur `"Variable"` :

```
PS C:\> $Z = "Variable"
PS C:\> $Z.<tab>
```

Une fois que vous avez saisi `$Z` et appuyé sur la touche Tab, PowerShell propose toutes les opérations possibles sur la variable `$Z`. Par exemple, si nous choisissons la propriété `$Z.Length` et appuyons sur Entrée, il retourne la taille de la chaîne contenue dans la variable `$Z` :

```
PS C:\> $Z = "Variable"
PS C:\> $Z.
<tab>
PS C:\> $Z.Length
8
PS C:\>
```

La complétion automatique des variables distingue les **propriétés** et les **méthodes**. Les propriétés sont affichées sans parenthèse ouvrante (comme dans l'exemple `$Z.Length`) et les méthodes sont présentées avec cette parenthèse :

```
PS C:\> $Z = "Variable"
PS C:\> $Z.con
<tab>
PS C:\> $Z.Contains(
```

Lorsque l'invite `$Z.Contains(` apparaît, voici comment déterminer si la variable `$Z` contient le caractère `V` :

```
PS C:\> $Z = "Variable"
PS C:\> $Z.Contains("V")
True
PS C:\
```

PowerShell corrige automatiquement la casse des noms de méthodes ou de propriétés afin qu'elle corresponde à leur définition. Dans la plupart des cas, cette fonctionnalité n'est que cosmétique car, par défaut, PowerShell n'est pas sensible à la casse.

Types de commandes

Lorsqu'on exécute une commande dans PowerShell, l'interpréteur en examine le nom afin de définir la tâche à effectuer. Ce processus détermine le type de la commande et la manière de l'exécuter. Il existe quatre types de commandes PowerShell : applets de commande, fonctions shell, scripts et commandes natives.

Applets de commande

Les premières sortes de commandes sont appelées **applets de commande** (*cmdlet*). Elles sont équivalentes aux commandes internes des autres shells en ligne de commande. Cependant, elles sont implémentées à l'aide de classes .NET compilées dans une bibliothèque dynamique (DLL, *Dynamic Link Library*) et chargées par PowerShell au moment de l'exécution. Cette différence signifie que les applets de commande intégrées ne sont pas figées ; tout le monde peut utiliser le kit de développement (SDK, *Software Developers Kit*) de PowerShell pour écrire ses propres applets et étendre les fonctionnalités de PowerShell.

Le nom d'une applet de commande est toujours composé d'un verbe et d'un nom séparé par un tiret (-). Le verbe précise l'opération effectuée par l'applet et le nom indique l'objet concerné par l'opération. Pour plus d'informations sur les applets de commande et leur syntaxe, consultez la section "Comprendre les applets de commande", page 39.

Fonctions shell

Les **fonctions shell** représentent le deuxième type de commandes. Une fonction shell permet d'affecter un nom à une liste de commandes. Les fonctions sont comparables aux sous-routines et aux procédures dans d'autres langages de programmation. Un script diffère d'une fonction en cela qu'une nouvelle instance du shell est démarrée pour chaque script alors que les fonctions s'exécutent dans l'instance courante du shell. Voici un exemple de définition d'une fonction simple dans PowerShell :

```
PS C:\> function ma-fonction-dir {get-childitem | ft Mode,Name}
```

Après avoir défini `ma-fonction-dir`, nous pouvons l'exécuter pour afficher le contenu du répertoire de travail dans un format particulier :

```
PS C:\Travail> ma-fonction-dir

Mode                                Name
----                                ----
d----                               Livres
d----                               Dev
d----                               Outils
d----                               VM
-a---                               Mémo Éric.txt
-a---                               Configurer des certificats.doc
-a---                               mediapc.txt

PS C:\Travail>
```

Nous pouvons examiner le déroulement d'une fonction dans la console en activant le débogage. Pour cela, utilisons la commande suivante :

```
PS C:\Travail> set-psdebug -trace 2
```

Ensuite, exécutons la fonction :

```
PS D:\Travail> ma-fonction-dir
DÉBOGUER :    1+ ma-fonction-dir
DÉBOGUER :     ! CALL function 'ma-fonction-dir'
DÉBOGUER :    1+ function ma-fonction-dir {get-childitem | ft Mode,Name}
...
```

Lorsque la fonction `ma-fonction-dir` est placée sur la pile, PowerShell exécute l'applet de commande `Get-ChildItem` comme indiqué dans la fonction. Pour désactiver le débogage, saisissez la commande `Set-PSDebug -trace 0`.

INFO

Les fonctions définies sur la ligne de commande, comme `ma-fonction-dir`, n'existent que pour la durée de la session PowerShell en cours. Elles sont également locales et ne s'appliquent pas aux nouvelles sessions PowerShell. Pour plus d'informations, consultez la section "Comprendre les portées", page 59.

Même si une fonction définie au niveau de la ligne de commande est utile pour créer dynamiquement une suite de commandes dans l'environnement PowerShell, de telles fonctions résident uniquement en mémoire et sont effacées lorsque PowerShell est fermé. Par conséquent, l'écriture de ces fonctions sous forme de scripts peut être plus pratique.

Scripts

Les **scripts**, le troisième type de commandes, sont des commandes PowerShell enregistrées dans des fichiers `.ps1`. Contrairement aux fonctions shell, qui ne sont pas conservées entre les sessions, ils sont stockés sur le disque et peuvent être invoqués à tout moment.

Les scripts peuvent être exécutés dans une session PowerShell ou par l'interpréteur de commandes de Windows. Pour lancer un script dans PowerShell, saisissez son nom sans l'extension. Ce nom peut être suivi d'un nombre quelconque de paramètres. Le shell exécute alors le premier fichier `.ps1` qui correspond au nom indiqué et qui se trouve dans l'un des chemins mentionnés dans la variable PowerShell `$ENV:PATH`.

```
PS C:\> monscript arg1 arg2
```

La commande précédente exécute le script `monscript.ps1` en utilisant les arguments `arg1` et `arg2`, si le script se trouve dans l'un des chemins de la variable `$ENV:PATH`. Dans le cas contraire, nous devons préciser son emplacement par l'une des deux méthodes suivantes :

```
PS C:\> & 'C:\Mes Scripts\monscript.ps1' arg1 arg2
PS C:\Scripts> .\monscript.ps1 arg1 arg2
```

INFO

L'opérateur d'appel `&` a été utilisé dans l'exemple précédent car le chemin du script contient des espaces et son nom doit donc être placé entre des apostrophes. Cet opérateur demande au shell d'évaluer la chaîne comme une commande. Si le chemin ne contient pas d'espace, vous pouvez omettre l'opérateur `&` ainsi que les apostrophes autour du nom du script.

Pour exécuter un script PowerShell depuis l'interpréteur de commandes de Windows, nous pouvons tout d'abord invoquer la commande `cd` pour aller dans le répertoire qui contient le script, puis lancer l'exécutable de PowerShell avec le paramètre `-command`, en précisant le script à exécuter :

```
C:\Scripts>powershell -command .\monscript.ps1
```

Si nous ne souhaitons pas aller dans le répertoire du script avec la commande `cd`, nous pouvons l'exécuter en utilisant un chemin absolu :

```
C:\>powershell -command C:\Scripts\monscript.ps1
```

Dans PowerShell, il est important de prêter attention aux autorisations de sécurité des scripts. Par défaut, leur exécution est interdite. Il s'agit là d'une méthode de protection contre les scripts malveillants. Cette stratégie peut être modifiée à l'aide de l'applet de commande `Set-ExecutionPolicy`, que nous étudierons au Chapitre 3, "Présentation avancée de PowerShell".

Commandes natives

Le dernier type de commandes, les **commandes natives**, est constitué des programmes externes que le système d'exploitation peut exécuter. Puisque l'invocation d'une commande native déclenche la création d'un nouveau processus, elles sont moins efficaces que les autres types

de commandes PowerShell. Elles possèdent également leurs propres paramètres pour le traitement des commandes, qui sont généralement différents des paramètres de PowerShell.

La gestion du focus des commandes natives par PowerShell peut représenter un problème d'utilisation majeur. Lorsqu'une commande native s'exécute, PowerShell peut attendre qu'elle se termine ou poursuivre son traitement. Prenons l'exemple suivant :

```
PS C:\> .\monfichier.txt
PS C:\>
```

L'invite de PowerShell est réaffichée presque immédiatement et l'éditeur par défaut des fichiers ayant l'extension `.txt` démarre et affiche `C:\monfichier.txt`. Si l'éditeur de texte par défaut n'a pas été changé, `notepad.exe` est lancé et ouvre `C:\monfichier.txt`.

INFO

PowerShell dispose d'une fonction de sécurité unique. Pour exécuter ou ouvrir un fichier du répertoire de travail, vous devez préfixer la commande avec `.\` ou `./`. Cette caractéristique de sécurité évite aux utilisateurs de PowerShell de lancer par mégarde une commande native ou un script sans préciser explicitement son exécution.

Le même comportement se produit lorsqu'on indique explicitement une commande native :

```
PS C:\> notepad C:\monfichier.txt
PS C:\>
```

Dans cet exemple, le fichier `C:\monfichier.txt` est ouvert dans Notepad et l'invite de PowerShell revient immédiatement. Cependant, si nous exécutons une commande native au beau milieu d'un pipeline (voir le Chapitre 1, "Introduction aux shells et à PowerShell"), PowerShell attend la fin du processus externe avant de repasser le contrôle à la console :

```
PS C:\> ping monserveur | findstr "TTL"
Réponse de 10.0.0.2 : octets=32 temps<1ms TTL=126
Réponse de 10.0.0.2 : octets=32 temps<1ms TTL=126
Réponse de 10.0.0.2 : octets=32 temps<1ms TTL=126
Réponse de 10.0.0.2 : octets=32 temps<1ms TTL=126
PS C:\>
```

PowerShell attend la fin du processus `ping` avant de redonner le contrôle à la console et de terminer le pipeline. Lorsque cette commande est exécutée (en remplaçant `monserveur` par le nom d'un hôte sur votre réseau local), l'invite de PowerShell disparaît brièvement pendant que la sortie de la commande `ping` est envoyée à la commande `findstr` pour qu'elle recherche la chaîne "TTL". L'invite de PowerShell ne revient que lorsque la commande native est terminée.

Invoquer PowerShell depuis d'autres shells

Outre l'utilisation en ligne de commande que nous venons de présenter, il est possible d'appeler PowerShell depuis d'autres shells, par exemple depuis l'interpréteur de commandes de Windows. Lorsque PowerShell est invoqué comme une application externe, nous pouvons préciser différentes commandes, paramètres et arguments. La commande suivante affiche l'ensemble des commandes, paramètres et arguments disponibles lorsque PowerShell est utilisé depuis l'interpréteur de commandes de Windows :

```
C:\>powershell -?

powershell[.exe] [-PSConsoleFile <file> | -Version <version>]
    [-NoLogo] [-NoExit] [-NoProfile] [-NonInteractive]
    [-OutputFormat {Text | XML}] [-InputFormat {Text | XML}]
    [-Command { - | <bloc_script> [-args <tableau_arguments>]
                  | <chaîne> [<paramètres_commande>] } ]

powershell[.exe] -Help | -? | /?


-PSConsoleFile
    Charge le fichier console de Windows PowerShell spécifié. Pour créer
    un fichier console, utilisez Export-Console dans Windows PowerShell.

-Version
    Démarre la version de Windows PowerShell spécifiée.

-NoLogo
    Masque la bannière de copyright au démarrage.

-NoExit
    Ne quitte pas après exécution des commandes de démarrage.

-NoProfile
    N'utilise pas le profil utilisateur.

-Noninteractive
    Ne présente pas d'invite interactive à l'utilisateur.
```

```
-OutputFormat
    Indique comment la sortie de Windows PowerShell est mise en forme. Les
    valeurs valides sont "Text" (chaînes de texte) ou "XML" (format CLIXML
    sérialisé).

-InputFormat
    Décrit le format des données envoyées à Windows PowerShell. Les valeurs
    valides sont "Text" (chaînes de texte) ou "XML" (format CLIXML sérialisé).

-Command
    Exécute les commandes spécifiées (et tous paramètres) comme si elles avaient
    été tapées à l'invite de commande de Windows PowerShell, puis quitte sauf
    si NoExit est spécifié. La valeur de Command peut être "-", une chaîne ou
    un bloc de script.
    Si la valeur de Command est "-", le texte de la commande est lu à partir
    de l'entrée standard.

    Les blocs de script doivent être entre accolades ({}). Vous ne pouvez
    spécifier un bloc de script qu'en exécutant PowerShell.exe dans Windows
    PowerShell. Les résultants du script sont retournés à l'environnement
    parent en tant qu'objets XML désérialisés, et non en direct.

    Si la valeur de Command est une chaîne, Command doit être le dernier
    paramètre de la commande, car tous les caractères tapés après la commande
    sont interprétés comme des arguments de commande.
    Pour écrire une chaîne qui exécute une commande Windows PowerShell,
    utilisez le format :
        "& {<commande>}"
    dans lequel les guillemets indiquent une chaîne et l'opérateur d'appel (&)
    entraîne l'exécution de la commande.

-Help, -?, /?
    Affiche ce message. Si vous tapez une commande powershell.exe dans Windows
    PowerShell, faites précéder les paramètres de commande d'un trait d'union
    (-), et non d'une barre oblique (/). Vous pouvez utiliser un trait d'union
    ou une barre oblique dans Cmd.exe.

EXEMPLES
    powershell -psconsolefile sqlsnapin.psc1
    powershell -version 1.0 -nologo -inputformat text -outputformat XML
    powershell -command {get-eventlog -logname security}
    powershell -command "& {get-eventlog -logname security}"


C:\>
```

Grâce à cette possibilité, nous pouvons exécuter des commandes PowerShell à partir de l'interpréteur de commandes de Windows. Lorsque PowerShell est appelé avec le paramètre `-command`, des scripts PowerShell ou d'autres applets de commande et commandes peuvent être utilisés comme arguments de `-command`. L'exemple suivant montre l'invocation de PowerShell depuis l'interpréteur de commandes de Windows, pour exécuter l'applet de commande `Get-Service` en ne conservant que les services qui se trouvent dans l'état `Running`, puis pour trier les résultats en fonction du contenu de la colonne `DisplayName` du service. La chaîne complète de la commande est placée entre des guillemets afin d'empêcher l'interpréteur de traiter le tube.

```
C:\>powershell.exe -command "get-service | where-object {$_.Status -eq
'Running'} | sort DisplayName"

Status   Name               DisplayName
------   ----               -----------
Running  stisvc             Acquisition d'image Windows (WIA)
Running  helpsvc            Aide et support
Running  APC UPS Service    APC UPS Service
Running  RpcSs              Appel de procédure distante (RPC)
Running  LmHosts            Assistance TCP/IP NetBIOS
Running  AudioSrv           Audio Windows
Running  avast! Antivirus   avast! Antivirus
Running  aswUpdSv           avast! iAVS4 Control Service
Running  avast! Mail Sca... avast! Mail Scanner
Running  avast! Web Scanner avast! Web Scanner
Running  wscsvc             Centre de sécurité
Running  TrkWks             Client de suivi de lien distribué
Running  Dhcp               Client DHCP
Running  Dnscache           Client DNS
Running  FastUserSwitchi... Compatibilité avec le Changement ra...
Running  WZCSVC             Configuration automatique sans fil
Running  seclogon           Connexion secondaire
Running  Netman             Connexions réseau
Running  ShellHWDetection   Détection matériel noyau
Running  ProtectedStorage   Emplacement protégé
Running  Browser            Explorateur d'ordinateur
Running  SamSs              Gestionnaire de comptes de sécurité
Running  RasMan             Gestionnaire de connexions d'accès ...
Running  dmserver           Gestionnaire de disque logique
Running  W32Time            Horloge Windows
Running  winmgmt            Infrastructure de gestion Windows
```

```
Running  Eventlog           Journal des événements
Running  DcomLaunch         Lanceur de processus serveur DCOM
Running  wuauserv           Mises à jour automatiques
Running  Nla                NLA (Network Location Awareness)
Running  SENS               Notification d'événement système
Running  NVSvc              NVIDIA Display Driver Service
Running  SharedAccess       Pare-feu Windows / Partage de conne...
Running  Schedule           Planificateur de tâches
Running  PlugPlay           Plug-and-Play
Running  lanmanserver       Serveur
Running  SSDPSRV            Service de découvertes SSDP
Running  ALG                Service de la passerelle de la couc...
Running  BITS               Service de transfert intelligent en...
Running  CryptSvc           Services de cryptographie
Running  PolicyAgent        Services IPSEC
Running  TermService        Services Terminal Server
Running  Spooler            Spouleur d'impression
Running  lanmanworkstation  Station de travail
Running  EventSystem        Système d'événements de COM+
Running  TapiSrv            Téléphonie
Running  Themes             Thèmes
Running  WebClient          WebClient
Running  UMWdf              Windows User Mode Driver Framework
```

Comprendre les applets de commande

Les applets de commande (ou cmdlets) sont un élément essentiel de la puissance de PowerShell. Elles sont implémentées comme des classes gérées (développées avec .NET Framework), qui offrent un ensemble de méthodes parfaitement défini pour traiter les données. Le programmeur écrit le code qui s'exécute lors de l'appel de l'applet de commande et le compile dans une DLL chargée dans une instance de PowerShell au redémarrage du shell.

Les applets de commande sont toujours nommées en respectant le format `Verbe-Nom`, où le verbe définit l'opération et le nom précise l'objet concerné par l'opération. Comme vous l'avez certainement noté, la plupart des noms de PowerShell sont au singulier afin de rendre PowerShell plus universel. En effet, une commande peut fournir une valeur ou un ensemble de valeurs et il est impossible de savoir à l'avance si le nom d'une applet de commande doit être pluriel. Par ailleurs, l'anglais n'est pas très cohérent dans son utilisation du pluriel.

Par exemple, le mot "fish" est au singulier ou au pluriel, selon le contexte. Puisque l'anglais n'est sans doute pas votre langue maternelle, il vous sera difficile de savoir ce qui doit être au pluriel ou d'imaginer la forme plurielle correcte.

INFO

Le verbe par défaut de PowerShell est `Get`. Si aucun autre verbe n'est donné, il est donc supposé. Ce fonctionnement par défaut signifie que la commande `Process` produit le même résultat que `Get-Process`.

Pour connaître les paramètres acceptés par une applet de commande, nous pouvons consulter ses informations d'aide avec l'une des commandes suivantes :

```
PS C:\> nom_cmdlet -?
PS C:\> get-help nom_cmdlet
```

D'autre part, l'applet de commande `Get-Command` permet de déterminer les paramètres disponibles et leur utilisation. Voici un exemple de sa syntaxe :

```
PS C:\> get-command nom_cmdlet
```

En redirigeant la sortie de `Get-Command` vers l'applet de commande `Format-List`, nous obtenons une liste plus concise de l'utilisation d'une applet de commande. Par exemple, voici comment afficher uniquement la définition de `Get-Process` :

```
PS C:\> get-command get-process | format-list Definition


Definition : Get-Process [[-Name] <String[]>] [-Verbose] [-Debug][-ErrorAction
               <ActionPreference>] [-ErrorVariable <String>][-OutVariable <Stri
             ng>] [-OutBuffer <Int32>]
             Get-Process -Id <Int32[]> [-Verbose] [-Debug][-ErrorAction <Actio
             nPreference>] [-ErrorVariable <String>][-OutVariable <String>] [-
             OutBuffer <Int32>]
             Get-Process -InputObject <Process[]> [-Verbose] [-Debug] [-ErrorAc
             tion <ActionPreference>] [-ErrorVariable <String>][-OutVariable <
             String>] [-OutBuffer <Int32>]



PS C:\>
```

Paramètres communs

Puisque les applets de commande dérivent d'une classe de base, quelques paramètres communs à toutes les applets permettent d'offrir une interface plus cohérente aux applets de commande de PowerShell. Ces paramètres sont décrits au Tableau 2.2.

Tableau 2.2 Paramètres communs de PowerShell

Paramètre	Type de données	Description
`Verbose`	`Boolean`	Génère une information détaillée concernant l'opération, de manière analogue à un journal de trace ou de transaction. Ce paramètre n'est valide qu'avec les applets de commande qui génèrent des données verbeuses.
`Debug`	`Boolean`	Génère des détails de l'opération destinés au programmeur. Ce paramètre n'est valide qu'avec les applets de commande qui génèrent des données de débogage.
`ErrorAction`	`Enum`	Détermine la réponse de l'applet de commande aux erreurs. Les valeurs acceptées sont `Continue` (par défaut), `Stop`, `SilentlyContinue` et `Inquire`.
`ErrorVariable`	`String`	Désigne une variable qui stocke les erreurs de la commande pendant son exécution. Cette variable est modifiée tout comme `$error`.
`OutVariable`	`String`	Désigne une variable qui stocke la sortie de la commande pendant son exécution.
`OutBuffer`	`Int32`	Détermine le nombre d'objets à placer dans le tampon avant d'invoquer l'applet de commande suivante du pipeline.
`WhatIf`	`Boolean`	Affiche le déroulement de l'exécution de la commande, mais sans réellement l'exécuter.
`Confirm`	`Boolean`	Demande une autorisation à l'utilisateur avant d'effectuer toute action qui modifie le système.

INFO

Les deux derniers paramètres du Tableau 2.2, `WhatIf` et `Confirm`, sont particuliers, car ils exigent que l'applet de commande prenne en charge la méthode .NET `ShouldProcess`, ce qui n'est peut-être pas le cas de toutes les applets de commande. La méthode `ShouldProcess` demande une confirmation de l'opération à l'utilisateur, en lui indiquant le nom de la ressource concernée par la modification.

Quelques applets de commande utiles

Lorsqu'on débute avec PowerShell, les applets de commande `Get-Help` et `Get-Command` se révèlent extrêmement utiles. Décrites dans les sections suivantes, elles vous aident à explorer le fonctionnement de PowerShell et à comprendre les commandes exécutées.

Get-Help

Comme vous devez le deviner, l'applet `Get-Help` affiche une aide sur les applets de commande et d'autres thèmes. Pour afficher la liste de tous les sujets d'aide, saisissons `Get-Help *` à l'invite de commande de PowerShell :

```
PS C:\> get-help *

Name                      Category                  Synopsis

----                      --------                  --------
ac                        Alias                     Add-Content
asnp                      Alias                     Add-PSSnapin
clc                       Alias                     Clear-Content
cli                       Alias                     Clear-Item
clp                       Alias                     Clear-ItemProperty
clv                       Alias                     Clear-Variable
cpi                       Alias                     Copy-Item
cpp                       Alias                     Copy-ItemProperty
cvpa                      Alias                     Convert-Path
diff                      Alias                     Compare-Object
epal                      Alias                     Export-Alias
epcsv                     Alias                     Export-Csv
fc                        Alias                     Format-Custom
fl                        Alias                     Format-List
```

```
foreach                 Alias                   ForEach-Object
...
Get-Command             Cmdlet                  Obtient des informatio...
Get-Help                Cmdlet                  Affiche des informatio...
Get-History             Cmdlet                  Obtient la liste des c...
Invoke-History          Cmdlet                  Exécute les commandes ...
Add-History             Cmdlet                  Ajoute des entrées à l...
ForEach-Object          Cmdlet                  Exécute une opération ...
Where-Object            Cmdlet                  Crée un filtre qui con...
Set-PSDebug             Cmdlet                  Active et désactive le...
Add-PSSnapin            Cmdlet                  Ajoute un ou plusieurs...
Remove-PSSnapin         Cmdlet                  Supprime les composant...
Get-PSSnapin            Cmdlet                  Obtient les composants...
Export-Console          Cmdlet                  Exporte la configurati...
Start-Transcript        Cmdlet                  Crée un enregistrement...
Stop-Transcript         Cmdlet                  Arrête une transcription.
Add-Content             Cmdlet                  Ajoute le contenu aux ...
Clear-Content           Cmdlet                  Supprime le contenu d'...
Clear-ItemProperty      Cmdlet                  Supprime la valeur d'u...
Join-Path               Cmdlet                  Combine un chemin d'ac...
Convert-Path            Cmdlet                  Convertit un chemin d'...
Copy-ItemProperty       Cmdlet                  Copie une propriété et...
Get-EventLog            Cmdlet                  Obtient des informatio...
Get-ChildItem           Cmdlet                  Obtient les éléments e...
Get-Content             Cmdlet                  Obtient le contenu de ...
Get-ItemProperty        Cmdlet                  Récupère les propriété...
Get-WmiObject           Cmdlet                  Obtient des instances ...
Move-ItemProperty       Cmdlet                  Déplace une propriété ...
Get-Location            Cmdlet                  Obtient des informatio...
Set-Location            Cmdlet                  Définit l'emplacement ...
Push-Location           Cmdlet                  Exécute une opération ...
Pop-Location            Cmdlet                  Définit l'emplacement ...
New-PSDrive             Cmdlet                  Installe un nouveau le...
Remove-PSDrive          Cmdlet                  Supprime un lecteur Wi...
Get-PSDrive             Cmdlet                  Obtient des informatio...
...
Alias                   Provider                Donne accès aux alias ...
Environment             Provider                Donne accès aux variab...
FileSystem              Provider                Fournisseur de PowerSh...
Function                Provider                Donne accès aux foncti...
Registry                Provider                Donne accès aux clés e...
Variable                Provider                Donne accès aux variab...
Certificate             Provider                Donne accès aux magasi...
```

```
about_alias                HelpFile                    Utilisation d'autres n...
about_arithmetic_operators HelpFile                    Opérateurs pouvant êtr...
about_array                HelpFile                    Structure de données c...
about_assignment_operators HelpFile                    Opérateurs pouvant êtr...
about_associative_array    HelpFile                    Structure de données c...
about_automatic_variables  HelpFile                    Variables définies aut...
about_break                HelpFile                    Instruction permettant...
about_command_search       HelpFile                    Explique comment Windo...
about_command_syntax       HelpFile                    Format de commande dan...
about_commonparameters     HelpFile                    Paramètres que chaque ...
about_comparison_operators HelpFile                    Opérateurs qui peuvent...
about_continue             HelpFile                    Revient immédiatement ...
about_core_commands        HelpFile                    Applets de commande pr...
about_display.xml          HelpFile                    Contrôle de l'affichag...
about_environment_variable HelpFile                    Comment accéder aux va...
...

PS C:\>
```

Si cette liste semble trop longue pour être utile, nous pouvons la raccourcir en appliquant un filtre sur le nom du thème et la catégorie. Par exemple, voici comment obtenir une liste de toutes les applets de commande dont le verbe commence par `Get` :

```
PS C:\> get-help -Name get-* -Category cmdlet

Name                       Category                Synopsis
----                       --------                --------
Get-Command                Cmdlet                  Obtient des informatio...
Get-Help                   Cmdlet                  Affiche des informatio...
Get-History                Cmdlet                  Obtient la liste des c...
Get-PSSnapin               Cmdlet                  Obtient les composants...
Get-EventLog               Cmdlet                  Obtient des informatio...
Get-ChildItem              Cmdlet                  Obtient les éléments e...
Get-Content                Cmdlet                  Obtient le contenu de ...
...

PS C:\>
```

Après avoir choisi un thème, nous pouvons obtenir l'aide en indiquant le nom du thème en paramètre à l'applet de commande `Get-Help`. Par exemple, la commande suivante fournit de l'aide sur `Get-Content` :

```
PS C:\> get-help get-content
```

INFO

Dans Windows PowerShell RC2, deux paramètres supplémentaires ont été ajoutés à `get-help` : `-detailed` et `-full`. Le paramètre `-detailed` affiche des informations complémentaires sur une applet de commande, y compris la description des paramètres et des exemples d'utilisation. Le paramètre `-full` affiche l'intégralité de l'aide d'une applet de commande, y compris des informations techniques concernant ses paramètres.

Contenu de l'aide sur une applet de commande

L'aide fournie par PowerShell est divisée en rubriques. Le Tableau 2.3 récapitule les détails des informations d'aide sur chaque applet de commande.

Tableau 2.3 Rubriques de l'aide de PowerShell

Rubrique	Description
Nom	Le nom de l'applet de commande
Résumé	Courte description du rôle de l'applet de commande
Description détaillée	Description détaillée du comportement de l'applet de commande, généralement avec des exemples d'utilisation
Syntaxe	Détails d'invocation de l'applet de commande
Paramètres	Paramètres reconnus par l'applet de commande
Type d'entrée	Type de l'entrée acceptée par l'applet de commande
Type de retour	Type des données retournées par l'applet de commande
Erreurs fatales	Si cette rubrique est présente, elle identifie les erreurs qui conduisent à l'arrêt prématuré de l'applet de commande

Tableau 2.3 Rubriques de l'aide de PowerShell *(suite)*

Rubrique	Description
Erreurs non fatales	Identifie les erreurs non critiques qui peuvent se produire pendant l'exécution de l'applet de commande sans pour cela la terminer
Remarques	Informations détaillées complémentaires sur l'utilisation de l'applet de commande, y compris des scénarios particuliers et des limitations possibles ou des curiosités
Exemples	Exemple d'utilisation classique de l'applet de commande
Liens connexes	Références à d'autres cmdlets qui réalisent des tâches similaires

Get-Command

`Get-Command` est également très utile, car elle affiche la liste de toutes les applets de commande disponibles dans une session PowerShell :

```
PS C:\> get-command

CommandType     Name                    Definition
-----------     ----                    ----------
Cmdlet          Add-Content             Add-Content [-Path] <String[...
Cmdlet          Add-History             Add-History [[-InputObject] ...
Cmdlet          Add-Member              Add-Member [-MemberType] <PS...
Cmdlet          Add-PSSnapin            Add-PSSnapin [-Name] <String...
Cmdlet          Clear-Content           Clear-Content [-Path] <Strin...
Cmdlet          Clear-Item              Clear-Item [-Path] <String[]...
Cmdlet          Clear-ItemProperty      Clear-ItemProperty [-Path] <...
Cmdlet          Clear-Variable          Clear-Variable [-Name] <Stri...
Cmdlet          Compare-Object          Compare-Object [-ReferenceOb...
...

PS C:\>
```

`Get-Command` est plus puissante que `Get-Help` car elle présente toutes les commandes disponibles (applets de commande, scripts, alias, fonctions et applications natives) dans une session PowerShell.

Par exemple :

```
PS C:\> get-command note*

CommandType     Name                    Definition
-----------     ----                    ----------
Application     NOTEPAD.EXE             C:\WINDOWS\NOTEPAD.EXE
Application     notepad.exe             C:\WINDOWS\system32\notepad.exe


PS C:\>
```

Lorsque `Get-Command` est invoquée avec des éléments autres que des applets de commande, les informations retournées sont légèrement différentes de celles retenues pour une applet de commande. Par exemple, pour une application existante, la valeur de la propriété `Definition` est le chemin de l'application. Cependant, d'autres informations concernant l'application sont également disponibles :

```
PS C:\> get-command ipconfig | format-list *
FileVersionInfo : File:             C:\WINDOWS\system32\ipconfig.exe
                  InternalName:     ipconfig.exe
                  OriginalFilename: ipconfig.exe
                  FileVersion:      5.1.2600.2180 (xpsp_sp2_rtm.040803-2158)
                  FileDescription:  Utilitaire de configuration IP
                  Product:          Système d'exploitation Microsoft®
Windows®
                  ProductVersion:   5.1.2600.2180
                  Debug:            False
                  Patched:          False
                  PreRelease:       False
                  PrivateBuild:     False
                  SpecialBuild:     False
                  Language:         Français (France)

Path            : C:\WINDOWS\system32\ipconfig.exe
Extension       : .exe
Definition      : C:\WINDOWS\system32\ipconfig.exe
Name            : ipconfig.exe
CommandType     : Application
```

Avec une fonction, la propriété `Definition` donne le corps de la fonction :

```
PS C:\> get-command Prompt

CommandType     Name                    Definition
-----------     ----                    ----------
Function        prompt                  Write-Host ("PS " + $(Get-Lo...


PS C:\>
```

Avec un alias, nous obtenons la commande cible de l'alias :

```
PS C:\> get-command write

CommandType     Name                            Definition
-----------     ----                            ----------
Alias           write                           Write-Output


PS C:\>
```

Avec un script, la propriété `Definition` contient le chemin du script. Si le script n'est pas de type PowerShell (comme un fichier `.bat` ou `.vbs`), l'information retournée est identique à celle des applications natives.

Expressions

PowerShell permet également d'évaluer des expressions. Dans l'exemple suivant, il retourne le résultat d'une expression mathématique simple :

```
PS C:\> (100 / 2) * 3
150
PS C:\>
```

INFO

Il est important de noter que, dans cet exemple, PowerShell calcule et affiche immédiatement le résultat de l'expression. Ce fonctionnement est différent des autres shells et langages de scripts, où le résultat de l'expression doit être affecté à une variable ou passé à une commande d'affichage avant de pouvoir être présenté à l'écran.

Même si PowerShell affiche immédiatement les résultats des expressions, rien ne nous empêche de les stocker dans des variables ou dans des fichiers texte pour une utilisation ultérieure. L'exemple suivant enregistre la sortie de l'expression dans la variable `$Calc` :

```
PS C:\> $Calc = (100 / 2) * 3
PS C:\> $Calc
150
PS C:\>
```

Cette technique peut également être étendue aux applets de commande. Dans l'exemple suivant, la sortie de `Get-Process` est affectée à la variable `$Procinfo` avec le paramètre `-Name` :

```
PS C:\> $Procinfo = get-process -Name explorer
PS C:\> $Procinfo

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  ProcessName
-------  ------    -----      ----- -----   ------     --  -----------
    494      12    14248      24804    83   107,45   2964  explorer


PS C:\> $Procinfo

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  ProcessName
-------  ------    -----      ----- -----   ------     --  -----------
    494      12    14248      24804    83   107,51   2964  explorer


PS C:\>
```

La variable `$Procinfo` contient le résultat de la commande `get-process -Name explorer`. Nous demandons ensuite à PowerShell de retrouver la valeur de `$Procinfo`. Il affiche le

résultat pour le processus `explorer`. Lorsque `$Procinfo` est affichée une seconde fois, la valeur de `CPU(s)` est différente. Cet exemple montre que le contenu de la variable `$Procinfo` est dynamique. Autrement dit, nous obtenons des informations en temps réel sur le processus `explorer`.

Comprendre les variables

Une **variable** est un emplacement permettant de stocker des données. Dans la majorité des shells, les variables ne peuvent contenir que des données de type texte. Dans les shells élaborés et les langages de programmation, les données placées dans des variables peuvent être quasiment de n'importe quel type, de la chaîne de caractères à des ensembles d'objets. De la même manière, les variables de PowerShell acceptent de stocker des données quelconques.

Pour définir une variable PowerShell, nous devons la nommer avec le préfixe `$`, lequel permet de différencier les variables des alias, des applets de commande, des noms de fichiers et des autres éléments du shell. Ce nom est sensible à la casse et peut contenir toute combinaison de caractères alphanumériques (A–Z et 0–9) et le caractère de soulignement (_). Même s'il n'existe aucune convention de nommage des variables, il est conseillé de leur donner un nom qui reflète le type des données contenues :

```
PS C:\> $MSProcesses = get-process | where {$_.company -match
".*Microsoft*"}
PS C:\> $MSProcesses

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     68       4     1712       6496    30     0.19   2420 ctfmon
    715      21    27024      40180   126    58.03   3620 explorer
    647      19    23160      36924   109    18.69   1508 iexplore
    522      11    31364      30876   151     6.59   3268 powershell
    354      17    28172      47612   482    36.22   2464 WINWORD


PS C:\>
```

Dans cet exemple, la variable `$MSProcesses` contient une liste des processus Microsoft actuellement en cours d'exécution sur le système.

INFO

Un nom de variable peut inclure n'importe quel caractère, y compris les espaces, à condition qu'il soit placé entre des accolades (les symboles { et }). Cependant, si vous définissez un nom de variable non standard, PowerShell vous signale que cette pratique est déconseillée.

Variables internes

Lorsqu'une session PowerShell démarre, un certain nombre de variables sont automatiquement définies :

```
PS C:\> set-location variable:
PS Variable:\> get-childitem

Name                  Value
----                  -----
Error                           {CommandNotFoundException}
DebugPreference                 SilentlyContinue
PROFILE                         \\bob'shosting.com\homes\tyson\My Documents\P...
HOME                            U:\
Host                            System.Management.Automation.Internal.Host.In...
MaximumHistoryCount             64
MaximumAliasCount               4096
input                           System.Array+SZArrayEnumerator
StackTrace                         à System.Management.Automation.CommandDis...
ReportErrorShowSource           1
ExecutionContext                System.Management.Automation.EngineIntrinsics
true                            True
VerbosePreference               SilentlyContinue
ShellId                         Microsoft.PowerShell
false                           False
null
MaximumFunctionCount            4096
ConsoleFileName
ReportErrorShowStackTrace       0
FormatEnumerationLimit          4
?                               True
PSHOME                          C:\Program Files\Windows PowerShell\v1.0
MyInvocation                    System.Management.Automation.InvocationInfo
PWD                             Variable:\
```

```
^                              set-location

_
ReportErrorShowExceptionClass  0
ProgressPreference             Continue
ErrorActionPreference          Continue
args                           {}
MaximumErrorCount              256
NestedPromptLevel              0
WhatIfPreference               0
$                              variable:
ReportErrorShowInnerException  0
ErrorView                      NormalView
WarningPreference              Continue
PID                            3124
ConfirmPreference             High
MaximumDriveCount              4096
MaximumVariableCount           4096


PS C:\>
```

Ces variables internes se répartissent en deux catégories. La première a une signification particulière dans PowerShell car elle enregistre des informations de configuration pour la session en cours. Parmi ces variables spéciales, les suivantes sont à retenir car elles sont souvent employées dans ce livre :

- `$_` contient l'objet en cours dans le pipeline.
- `$Error` contient les objets d'erreur de la session PowerShell en cours.

```
PS C:\> get-service | where-object {$_.Name -match "W32Time"}

Status   Name               DisplayName
------   ----               -----------
Running  W32Time            Horloge Windows


PS C:\>
```

```
PS C:\> $Error
Vous devez fournir une expression de valeur à droite de l'opérateur « * ».
PS C:\>
```

La seconde catégorie de variables internes comprend les paramètres de préférence qui contrôlent le comportement de PowerShell. Le Tableau 2.4 les décrit.

INFO

Une stratégie de commande peut être l'une des chaînes suivantes : `SilentlyContinue`, `NotifyContinue`, `NotifyStop` ou `Inquire`.

Tableau 2.4 Paramètres de préférence de PowerShell

Nom	Valeur acceptée	Description
`$DebugPreference`	Stratégie de commande	Action à effectuer lorsque des données sont écrites à l'aide de `Write-Debug` dans un script ou de `WriteDebug()` dans une applet de commande ou un fournisseur
`$ErrorActionPreference`	Stratégie de commande	Action à effectuer lorsque des données sont écrites à l'aide de `Write-Error` dans un script ou de `WriteError()` dans une applet de commande ou un fournisseur
`$MaximumAliasCount`	Entier	Nombre maximal d'alias
`$MaximumDriveCount`	Entier	Nombre maximal de lecteurs autorisés
`$MaximumErrorCount`	Entier	Nombre maximal d'erreurs placées dans `$Error`
`$MaximumFunctionCount`	Entier	Nombre maximal de fonctions pouvant être définies
`$MaximumVariableCount`	Entier	Nombre maximal de variables pouvant être définies
`$MaximumHistoryCount`	Entier	Nombre maximal d'entrées enregistrées dans l'historique des commandes
`$ShouldProcessPreference`	Stratégie de commande	Action à effectuer lorsque `ShouldProcess` est utilisé dans une applet de commande
`$ProcessReturnPreference`	Booléen	Valeur retournée par `ShouldProcess`
`$ProgressPreference`	Stratégie de commande	Action à effectuer lorsque des données sont écrites à l'aide de `Write-Progress` dans un script ou de `WriteProgress()` dans une applet de commande ou un fournisseur
`$VerbosePreference`	Stratégie de commande	Action à effectuer lorsque des données sont écrites à l'aide de `Write-Verbose` dans un script ou de `WriteVerbose()` dans une applet de commande ou un fournisseur

Comprendre les alias

Vous le remarquerez très rapidement, l'utilisation de PowerShell demande une saisie importante, à moins que vous n'exécutiez un script. Par exemple, ouvrons une console PowerShell et tapons la commande suivante :

```
PS C:\> get-process | where-object {$_.Company -match ".*Microsoft*"} |
format-table Name, ID, Path –Autosize
```

Elle est plutôt longue à saisir. Heureusement, comme la plupart des shells, PowerShell prend en charge les alias d'applets de commande et d'exécutables. Ainsi, pour éviter de saisir une commande aussi longue, nous pouvons utiliser les alias par défaut de PowerShell. Dans ce cas, l'exemple `Get-Process` devient :

```
PS C:\> gps | ? {$_.Company -match ".*Microsoft*"} | ft Name, ID, Path –Autosize
```

L'économie n'est pas énorme, mais les alias peuvent faire gagner du temps et éviter les fautes de frappe. Pour obtenir la liste des alias définis dans la session PowerShell en cours, invoquons `Get-Alias` :

```
PS C:\> get-alias

CommandType     Name                               Definition
-----------     ----                               ----------
Alias           ac                                 Add-Content
Alias           asnp                               Add-PSSnapin
Alias           clc                                Clear-Content
Alias           cli                                Clear-Item
Alias           clp                                Clear-ItemProperty
Alias           clv                                Clear-Variable
Alias           cpi                                Copy-Item
Alias           cpp                                Copy-ItemProperty
Alias           cvpa                               Convert-Path
Alias           diff                               Compare-Object
Alias           epal                               Export-Alias
Alias           epcsv                              Export-Csv
Alias           fc                                 Format-Custom
Alias           fl                                 Format-List
Alias           foreach                            ForEach-Object
```

```
Alias           %                               ForEach-Object
Alias           ft                              Format-Table
Alias           fw                              Format-Wide
Alias           gal                             Get-Alias
Alias           gc                              Get-Content
Alias           gci                             Get-ChildItem
Alias           gcm                             Get-Command
Alias           gdr                             Get-PSDrive
Alias           ghy                             Get-History
Alias           gi                              Get-Item
Alias           gl                              Get-Location
Alias           gm                              Get-Member
Alias           gp                              Get-ItemProperty
Alias           gps                             Get-Process
Alias           group                           Group-Object
Alias           gsv                             Get-Service
Alias           gsnp                            Get-PSSnapin
Alias           gu                              Get-Unique
Alias           gv                              Get-Variable
Alias           gwmi                            Get-WmiObject
Alias           iex                             Invoke-Expression
Alias           ihy                             Invoke-History
Alias           ii                              Invoke-Item
Alias           ipal                            Import-Alias
Alias           ipcsv                           Import-Csv
Alias           mi                              Move-Item
Alias           mp                              Move-ItemProperty
Alias           nal                             New-Alias
Alias           ndr                             New-PSDrive
Alias           ni                              New-Item
Alias           nv                              New-Variable
Alias           oh                              Out-Host
Alias           rdr                             Remove-PSDrive
Alias           ri                              Remove-Item
Alias           rni                             Rename-Item
Alias           rnp                             Rename-ItemProperty
Alias           rp                              Remove-ItemProperty
Alias           rsnp                            Remove-PSSnapin
Alias           rv                              Remove-Variable
Alias           rvpa                            Resolve-Path
Alias           sal                             Set-Alias
Alias           sasv                            Start-Service
Alias           sc                              Set-Content
Alias           select                          Select-Object
Alias           si                              Set-Item
```

```
Alias           sl                              Set-Location
Alias           sleep                           Start-Sleep
Alias           sort                            Sort-Object
Alias           sp                              Set-ItemProperty
Alias           spps                            Stop-Process
Alias           spsv                            Stop-Service
Alias           sv                              Set-Variable
Alias           tee                             Tee-Object
Alias           where                           Where-Object
Alias           ?                               Where-Object
Alias           write                           Write-Output
Alias           cat                             Get-Content
Alias           cd                              Set-Location
Alias           clear                           Clear-Host
Alias           cp                              Copy-Item
Alias           h                               Get-History
Alias           history                         Get-History
Alias           kill                            Stop-Process
Alias           lp                              Out-Printer
Alias           ls                              Get-ChildItem
Alias           mount                           New-PSDrive
Alias           mv                              Move-Item
Alias           popd                            Pop-Location
Alias           ps                              Get-Process
Alias           pushd                           Push-Location
Alias           pwd                             Get-Location
Alias           r                               Invoke-History
Alias           rm                              Remove-Item
Alias           rmdir                           Remove-Item
Alias           echo                            Write-Output
Alias           cls                             Clear-Host
Alias           chdir                           Set-Location
Alias           copy                            Copy-Item
Alias           del                             Remove-Item
Alias           dir                             Get-ChildItem
Alias           erase                           Remove-Item
Alias           move                            Move-Item
Alias           rd                              Remove-Item
Alias           ren                             Rename-Item
Alias           set                             Set-Variable
Alias           type                            Get-Content


PS C:\>
```

Découvrir les applets de commande relatives aux alias

Plusieurs applets de commande permettent de définir de nouveaux alias, d'exporter, d'importer et d'afficher les alias existants. Grâce à la commande suivante, nous pouvons obtenir toutes les cmdlets relatives aux alias :

```
PS C:\> get-command *-Alias

CommandType     Name                    Definition
-----------     ----                    ----------
Cmdlet          Export-Alias            Export-Alias [-Path] <String...
Cmdlet          Get-Alias               Get-Alias [[-Name] <String[]...
Cmdlet          Import-Alias            Import-Alias [-Path] <String...
Cmdlet          New-Alias               New-Alias [-Name] <String> [...
Cmdlet          Set-Alias               Set-Alias [-Name] <String> [...
```

Nous avons déjà vu comment utiliser `Get-Alias` pour obtenir la liste des alias définis dans la session PowerShell en cours. Les applets de commande `Export-Alias` et `Import-Alias` permettent d'exporter et d'importer des listes d'alias entre des sessions PowerShell. Quant à `New-Alias` et à `Set-Alias`, elles permettent de définir de nouveaux alias dans la session PowerShell en cours.

INFO

L'implémentation des alias dans PowerShell est limitée. Comme nous l'avons mentionné précédemment, un alias ne fonctionne que pour les applets de commande et les exécutables, sans paramètres. Cependant, certaines méthodes permettent de contourner cette contrainte. L'une d'elles consiste à définir la commande dans une variable, puis à appeler la variable depuis d'autres commandes. Mais la variable ne peut être appelée que dans la session PowerShell en cours, sauf si elle est définie dans le fichier `profile.ps1`. Une autre méthode, conseillée, consiste à placer la commande dans une fonction.

Créer des alias persistants

Les alias créés avec `New-Alias` et `Set-Alias` ne sont valides que dans la session PowerShell en cours. Lorsque vous fermez cette session, les alias existants sont effacés. Pour que des alias persistent entre des sessions PowerShell, nous devons les définir dans le fichier `profile.ps1` :

```
set-alias new new-object
set-alias time get-date
...
```

Bien que l'économie de saisie soit attirante, nous vous déconseillons d'abuser des alias. En effet, ils ne sont pas très portables. Par exemple, si nous employons de nombreux alias dans un script, nous devons inclure une suite de commandes `Set-Alias` au début du script pour être certain de l'existence de ces alias, quel que soit le profil de machine ou de session, au moment de l'exécution du script.

Cependant, leur principal problème n'est pas la portabilité mais la confusion ou le masquage de la réelle signification des commandes ou des scripts. Si les alias que nous définissons ont un sens pour nous, tout le monde ne partage pas notre logique. Par conséquent, si nous voulons que les autres utilisateurs comprennent nos scripts, nous devons éviter de trop en employer. À la place, il est préférable de créer des fonctions réutilisables.

INFO

Lorsque vous créez des alias pour des scripts, utilisez des noms compréhensibles par les autres personnes. Par exemple, il n'y a aucune raison, autre que celle de vouloir chiffrer vos scripts, de créer des alias constitués uniquement de deux lettres.

Séquences d'échappement

Le caractère accent grave ou apostrophe inverse (`) joue le rôle de caractère d'échappement dans PowerShell. Selon son contexte d'utilisation, PowerShell interprète les caractères qui le suivent immédiatement de différentes manières.

Si l'apostrophe inverse est utilisée à la fin d'une ligne dans un script, elle sert de caractère de continuation. Autrement dit, le rôle de ` est équivalent à celui de `&` en VBScript. Elle permet de découper les longues lignes de code en morceaux plus petits :

```
$Reg = get-wmiobject -Namespace Root\Default -computerName `
    $Computer -List | where-object `
    {$_.Name -eq "StdRegProv"}
```

Si l'apostrophe inverse précède une variable PowerShell, les caractères venant immédiatement après ne sont pas soumis à la substitution et ne sont pas interprétés :

```
PS C:\> $Chaîne = "Cela fonctionne-t-il ?"
PS C:\> write-host "La question est : $Chaîne"
La question est : Cela fonctionne-t-il ?
PS C:\> write-host "La question est : `$Chaîne"
La question est : $Chaîne
PS C:\>
```

Si l'apostrophe inverse est employée dans une chaîne ou interprétée comme une partie d'une chaîne, cela signifie que le caractère suivant doit être considéré comme un caractère spécial. Par exemple, pour placer un caractère de tabulation dans une chaîne, nous utilisons la séquence d'échappement `` `t `` :

```
PS C:\> $Chaîne = "Voyez la tabulation :`t [TAB]"
PS C:\> write-host $Chaîne
Voyez la tabulation :         [TAB]
PS C:\>
```

Le Tableau 2.5 récapitule les séquences d'échappement reconnues par PowerShell.

Tableau 2.5 Séquences d'échappement de PowerShell

Caractère	Signification
`` `' ``	Apostrophe
`` `" ``	Guillemets
`` `0 ``	Caractère nul
`` `a ``	Alarme (émission d'un bip)
`` `b ``	Espace arrière
`` `f ``	Saut de page (pour les impressions)
`` `n ``	Saut de ligne
`` `r ``	Retour chariot
`` `t ``	Tabulation horizontale (8 espaces)
`` `v ``	Tabulation verticale (pour les impressions)

Comprendre les portées

Une **portée** est une frontière logique dans PowerShell qui isole l'utilisation des fonctions et des variables. Les portées peuvent être globales, locales, de scripts et privées. Elles fonctionnent comme une hiérarchie, dans laquelle les informations sont héritées vers le bas. Par exemple, la portée locale a accès à la portée globale, mais pas l'inverse. Les portées et leurs utilisations sont décrites au fil des sections suivantes.

Portée globale

Comme l'implique son nom, une **portée globale** s'applique à l'intégralité d'une instance PowerShell. Les données de la portée globale sont héritées par toutes les portées enfants. Par conséquent, n'importe quel script, commande ou fonction a accès aux variables définies dans la portée globale. Cependant, les portées globales ne sont pas partagées entre les différentes instances de PowerShell.

L'exemple suivant montre la définition de la variable globale `$Processus` dans la fonction `AfficherProcessus`. Puisque la variable `$Processus` est définie globalement, nous pouvons consulter la valeur de `$Processus.Count` après la terminaison de `AfficherProcessus`. Nous obtenons le nombre de processus actifs au moment de l'exécution d'`AfficherProcessus`.

```
PS C:\> function AfficherProcessus {$Global:Processus = get-process}
PS C:\> AfficherProcessus
PS C:\> $Processus.Count
37
```

INFO

Dans PowerShell, vous pouvez utiliser un indicateur de portée explicite pour fixer la portée d'une variable. Par exemple, si vous souhaitez qu'une variable réside dans la portée globale, définissez-la avec `$Global:nomDeLaVariable`. Lorsque l'indicateur explicite n'est pas utilisé, la variable réside dans la portée courante.

Portée locale

Une **portée locale** est créée dynamiquement chaque fois qu'une fonction, un filtre ou un script s'exécute. Une fois la portée locale terminée, les informations qu'elle contenait sont effacées. Une portée locale peut lire les informations d'une portée globale, mais elle ne peut pas les modifier.

L'exemple suivant montre la variable locale `$Processus` définie dans la fonction `AfficherProcessus`. Après la fin d'`AfficherProcessus`, la variable `$Processus` ne contient plus aucune donnée car elle a été définie uniquement dans la fonction `AfficherProcessus`. Comme vous pouvez le constater, l'affichage de la valeur de `$Processus.Count`, après l'exécution de la fonction `AfficherProcessus`, ne produit aucun résultat.

```
PS C:\> function AfficherProcessus {$Processus = get-process}
PS C:\> AfficherProcessus
PS C:\> $Processus.Count
PS C:\>
```

Portée de script

Une **portée de script** est créée dès qu'un fichier de script s'exécute et elle est détruite une fois le script terminé. Pour illustrer ce fonctionnement, créons le script suivant et enregistrons-le sous le nom `AfficherProcessus.ps1` :

```
$Processus = get-process
write-host "Voici le premier processus :" -Foregroundcolor Yellow
$Processus[0]
```

Exécutons-le ensuite dans une session PowerShell. La sortie doit être comparable à la suivante :

```
PS C:\> .\AfficherProcessus.ps1
Voici le premier processus :

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    105       5     1992       4128    32     0,05    916 alg


PS C:\> $Processus[0]
Impossible d'indexer dans un tableau Null.
Au niveau de ligne : 1 Caractère : 12
+ $Processus[0 <<<< ]
PS C:\>
```

Lorsque le script `AfficherProcessus.ps1` s'exécute, les informations concernant le premier objet de processus dans la variable `$Processus` sont affichées sur la console. En revanche, lorsque nous essayons d'accéder aux informations contenues dans la variable `$Processus` depuis la console, une erreur est affichée car cette variable n'est valide que dans la portée du script. Lorsque le script se termine, cette portée et tout son contenu disparaissent.

Que se passe-t-il si nous essayons d'utiliser un script dans un pipeline ou d'y accéder comme à une bibliothèque de fonctions communes ? Normalement, ce fonctionnement n'est pas possible car PowerShell détruit une portée de script dès la fin de son exécution. Cela dit, PowerShell offre la commande "point", issue du monde UNIX. La commande point demande à PowerShell de charger une portée de script dans la portée de l'appelant.

Pour utiliser cette fonctionnalité, il suffit de préfixer le nom de script par un point (`.`) lors de son invocation :

```
PS C:\> . .\monscript.ps1
```

Portée privée

Une **portée privée** est analogue à une portée locale, à une différence près : ses définitions ne sont pas héritées par les portées enfants.

L'exemple suivant montre la définition de la variable privée `$Processus` dans la fonction `AfficherProcessus`. Pendant l'exécution de cette fonction, la variable `$Processus` n'est pas disponible à la portée enfant représentée par le bloc de script placé entre les caractères { et } (lignes 6 à 9).

```
PS C:\> function AfficherProcessus {$Private:Processus = get-process
>>     write-host "Voici le premier processus :" -Foregroundcolor Yellow
>>     $Processus[0]
>>     write-host
>>
>>     &{
>>         write-host "Le voici à nouveau :" -Foregroundcolor Yellow
>>         $Processus[0]
>>     }
>> }
>>
PS C:\> AfficherProcessus
```

```
Voici le premier processus :

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    105       5     1992       4128    32     0,05    916 alg

Le voici à nouveau :
Impossible d'indexer dans un tableau Null.
Au niveau de ligne : 7 Caractère : 20
+         $Processus[0 <<<< ]


PS C:\>
```

Cet exemple fonctionne car il utilise l'opérateur d'invocation `&`. Il est ainsi possible d'exécuter des fragments de code dans une portée locale isolée. Cette technique permet d'isoler un bloc de script et ses variables de la portée parente ou, comme dans cet exemple, d'empêcher un bloc de script d'accéder à une variable privée.

Premier script

La plupart des commandes décrites dans ce chapitre sont interactives. Autrement dit, nous saisissons des commandes à l'invite de PowerShell et la sortie est affichée. Même si l'utilisation interactive de PowerShell est pratique pour des tâches qui ne doivent être effectuées qu'une seule fois, cette méthode n'est pas efficace pour reproduire des tâches d'automation. Heureusement, PowerShell est capable de lire des fichiers contenant des commandes. Nous pouvons ainsi écrire, enregistrer, puis rappeler une suite de commandes selon les besoins. L'ensemble de ces commandes enregistrées est généralement appelé **script**.

Les scripts PowerShell sont de simples fichiers texte enregistrés avec l'extension `.ps1`. Nous pouvons utiliser n'importe quel éditeur de texte, comme Bloc-notes, pour créer un fichier qui contient les commandes de notre script PowerShell. Par exemple, ouvrons Bloc-notes et saisissons les commandes suivantes :

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

Enregistrons ce fichier sous le nom `AfficherServicesArretes.ps1` dans un répertoire (`C:\Scripts` dans notre exemple).

Avant de pouvoir exécuter ce script, nous devons ajuster la stratégie d'exécution de PowerShell car la configuration par défaut interdit l'exécution des scripts, pour des raisons de protection contre le code malveillant. Pour cela, invoquons `Set-ExecutionPolicy` comme le montre l'exemple suivant. Nous pouvons également utiliser `Get-ExecutionPolicy` pour connaître la stratégie d'exécution en cours. (Le Chapitre 3 détaillera la sécurité PowerShell et les pratiques conseillées.)

```
PS C:\> set-executionpolicy RemoteSigned
PS C:\> get-executionpolicy
RemoteSigned
PS C:\>
```

La stratégie `RemoteSigned` permet d'exécuter les scripts créés localement sans qu'ils soient signés numériquement (un concept qui sera expliqué au Chapitre 4, "Signer du code"). En revanche, les scripts téléchargés sur Internet doivent être signés. Cette configuration nous donne toute liberté d'exécuter des scripts non signés provenant de la machine locale, tout en offrant une certaine protection contre les scripts externes non signés.

Après avoir fixé la stratégie d'exécution de PowerShell à `RemoteSigned`, nous pouvons exécuter le script dans une session PowerShell en entrant simplement son chemin de répertoire complet et son nom de fichier. Avec la commande `C:\Scripts\AfficherServicesArretes.ps1`, nous obtenons la sortie suivante :

```
PS C:\> C:\Scripts\AfficherServicesArretes.ps1

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Avertissement
Stopped  AppMgmt            Gestion d'applications
Stopped  aspnet_state       Service d'état ASP.NET
Stopped  CiSvc              Service d'indexation
Stopped  ClipSrv            Gestionnaire de l'Album
Stopped  clr_optimizatio... .NET Runtime Optimization Service v...
Stopped  COMSysApp          Application système COM+
Stopped  dmadmin            Service d'administration du Gestion...
Stopped  HidServ            Accès du périphérique d'interface u...
Stopped  HTTPFilter         HTTP SSL
Stopped  IDriverT           InstallDriver Table Manager
Stopped  ImapiService       Service COM de gravage de CD IMAPI
Stopped  mnmsrvc            Partage de Bureau à distance NetMee...
```

```
Stopped  MSDTC              Distributed Transaction Coordinator
Stopped  MSIServer          Windows Installer
Stopped  NetDDE             DDE réseau
Stopped  NetDDEdsdm         DSDM DDE réseau
Stopped  NtLmSsp            Fournisseur de la prise en charge d...
Stopped  NtmsSvc            Stockage amovible
Stopped  ose                Office Source Engine
Stopped  RasAuto            Gestionnaire de connexion automatiq...
Stopped  RDSessMgr          Gestionnaire de session d'aide sur ...
Stopped  RemoteAccess       Routage et accès distant
Stopped  RemoteRegistry     Accès à distance au Registre
Stopped  RpcLocator         Localisateur d'appels de procédure ...
Stopped  RSVP               QoS RSVP
Stopped  SCardSvr           Carte à puce
Stopped  SwPrv              MS Software Shadow Copy Provider
Stopped  SysmonLog          Journaux et alertes de performance
Stopped  TlntSvr            Telnet
Stopped  upnphost           Hôte de périphérique universel Plug...
Stopped  UPS                Onduleur
Stopped  VSS                Cliché instantané de volume
Stopped  WmdmPmSN           Service de numéro de série du lecte...
Stopped  Wmi                Extensions du pilote WMI
Stopped  WmiApSrv           Carte de performance WMI
Stopped  xmlprov            Service d'approvisionnement réseau


PS C:\>
```

Bien que ce script d'une ligne soit simple, il illustre parfaitement l'écriture et l'utilisation d'un script dans PowerShell. Si nécessaire, nous pouvons inclure d'autres commandes pour qu'il effectue une tâche d'automation. En voici un exemple :

```
param ([string] $DemarrerAvec)

$ServicesArretes = get-service | where-object {$_.Status -eq "Stopped"}

write-host "Les services $DemarrerAvec suivants sont arrêtés sur" `
    "$Env:COMPUTERNAME :" -Foregroundcolor Yellow

$ServicesArretes | where-object {$_.Name -like $DemarrerAvec} | `
    format-table Name, DisplayName
```

Ce script affiche les résultats suivants :

```
PS C:\> C:\Scripts\AfficherServicesArretes.ps1 N*
Les services N* suivants sont arrêtés sur PLANX :

Name                                    DisplayName
----                                    -----------
NetDDE                                  DDE réseau
NetDDEdsdm                              DSDM DDE réseau
NtLmSsp                                 Fournisseur de la prise en charge
de...
NtmsSvc                                 Stockage amovible


PS C:\>
```

Ce script est un peu plus complexe car il peut filtrer les services arrêtés en fonction de la chaîne fournie. Il ne s'agit pas d'un élément d'automation très compliqué, mais il permet d'illustrer quelques possibilités de PowerShell. Pour en bénéficier, vous devez simplement mieux comprendre les fonctionnalités de PowerShell afin d'écrire des scripts plus complexes et plus intéressants.

En résumé

Vous venez de découvrir les bases de PowerShell. Au cours de cet apprentissage, vous avez fait connaissance avec certains concepts, comme les différents types de commandes PowerShell, les applets de commandes (cmdlets), les alias, les variables, l'interface en ligne de commande et les portées. Après cette présentation, vous avez abordé l'écriture de scripts PowerShell et développé votre premier script. Cependant, l'élément le plus important de ce chapitre est que vous avez téléchargé et installé PowerShell avant de commencer à le manipuler.

En utilisant simplement PowerShell, vous avez franchi la première des nombreuses étapes menant à sa maîtrise complète. Le premier pas est toujours le plus difficile et, une fois fait, la route devient de plus en plus facile. En lisant les chapitres suivants, vous remarquerez que vos compétences augmentent avec l'acquisition de nouveaux concepts et l'utilisation de PowerShell dans la réalisation de tâches d'automation.

3

Présentation avancée de PowerShell

Dans ce chapitre

- Introduction
- Orientation objet
- Comprendre les fournisseurs
- Comprendre les erreurs
- Gérer les erreurs
- Profils
- Comprendre la sécurité
- Langage

Introduction

Ce chapitre détaille certaines particularités du fonctionnement de PowerShell que vous devez comprendre avant de passer aux chapitres traitant de l'écriture de scripts. Ne vous attachez pas trop aux détails. L'objectif est de comprendre les concepts. Puisque PowerShell apporte son lot de changements par rapport à l'ancienne écriture des scripts pour Windows, vous devez également modifier vos méthodes de développement. Avec un peu de pratique, l'écriture de scripts PowerShell vous sera aussi familière que le développement de scripts en VBScript ou JScript, qui constituent les méthodes standard pour mettre en œuvre des tâches d'automation sous Windows.

Orientation objet

La plupart des shells opèrent dans un environnement de type texte. Cela signifie généralement que la sortie doit être traitée pour effectuer des tâches d'automation. Par exemple, si les données d'une commande doivent être envoyées à une autre commande, la sortie de la première doit généralement être remise en forme afin de répondre aux exigences d'entrée de la seconde. Même si cette méthode fonctionne depuis des années, le traitement de données dans un format textuel peut être difficile et frustrant.

Le plus souvent, un travail important est nécessaire pour transformer les données textuelles en un format utilisable. Dans PowerShell, Microsoft a décidé de modifier les standards. Au lieu de transporter les données sous forme de texte brut, PowerShell le fait sous forme d'objets .NET Framework, ce qui permet aux applets de commande d'accéder directement aux propriétés et aux méthodes d'un objet. Cette évolution simplifie également l'usage du shell. Plutôt que de modifier des données textuelles, nous pouvons simplement faire référence aux données requises par leur nom. De même, au lieu d'écrire du code pour convertir les données dans un format utilisable, nous pouvons simplement faire référence aux objets et les manipuler comme bon nous semble.

Comprendre le pipeline

Les objets nous apportent une méthode de traitement des données plus robuste. Par le passé, les données étaient transférées d'une commande à la suivante en utilisant un pipeline (*tube*). Il est ainsi possible d'enchaîner une suite de commandes afin de réunir des informations concernant un système. Cependant, comme nous l'avons mentionné précédemment, la plupart des shells présentent un inconvénient majeur : les informations fournies par les commandes sont du texte. Ce texte brut doit être transformé en un format compatible avec la commande suivante avant d'être placé dans le pipeline. Pour comprendre ce fonctionnement, examinons l'exemple Bash suivant :

```
$ ps -ef | grep "bash" | cut -f2
```

L'objectif est de trouver l'identifiant du processus (PID, *Process ID*) `bash`. La liste des processus en cours d'exécution est obtenue à l'aide de la commande `ps`. Ensuite, elle est envoyée, *via* un tube (|), à la commande `grep`, qui applique un filtre avec la chaîne "bash". Les informations restantes sont envoyées à la commande `cut`, qui retourne le second champ, dans lequel se trouve le PID (le délimiteur de champ est un caractère de tabulation).

INFO

Un **délimiteur** est un caractère qui sert à séparer les champs de données. Le délimiteur par défaut de la commande `cut` est un caractère de tabulation. Pour choisir un autre délimiteur, vous devez utiliser le paramètre `-d`.

D'après les informations des pages de manuel des commandes `grep` et `cut`, la commande `ps` devrait fonctionner. Cependant, le PID n'est pas retourné ni affiché dans le bon format.

La commande ne fonctionne pas car le shell Bash nous oblige à manipuler les données textuelles pour afficher le PID. La commande `ps` envoie sa sortie dans un format textuel et sa conversion en un format utilisable nécessite d'autres commandes, comme `grep` et `cut`. La manipulation des données textuelles complique notre tâche. Par exemple, pour obtenir le PID à partir des données renvoyées par la commande `grep`, nous devons indiquer l'emplacement du champ et le délimiteur pour que la commande `cut` produise l'information recherchée. Pour cela, exécutons la première partie de la commande `ps` :

```
$ ps -ef | grep "bash"
   bob    3628       1 con  16:52:46 /usr/bin/bash
```

Le champ recherché est le second (3628). Vous remarquerez que la commande `ps` ne sépare pas les colonnes de sa sortie par un caractère de tabulation. À la place, elle utilise un nombre variable d'espaces ou un délimiteur espace blanc entre les champs.

INFO

Un **délimiteur espace blanc** est constitué de caractères, comme les espaces ou les tabulations, qui équivalent à un espace vide.

La commande `cut` ne peut savoir que les espaces doivent être employés comme séparateurs de champ. Il s'agit de la raison du dysfonctionnement de la commande. Pour obtenir le PID, nous devons nous servir du langage awk. La commande et la sortie deviennent alors les suivantes :

```
$ ps -ef | grep "bash" | awk '{print $2}'
3628
```

Le point important est que, malgré la puissance des commandes des shells UNIX et Linux, ils peuvent être compliqués et frustrants. Puisque ces shells utilisent un format textuel, les commandes manquent souvent de fonctionnalités ou nécessitent des commandes ou des outils supplémentaires pour effectuer certaines tâches. Pour prendre en charge les différences dans les sorties textuelles des commandes du shell, de nombreux utilitaires et langages de scripts ont été développés.

Toutes ces conversions ont pour résultat une arborescence de commandes et d'outils qui rendent les shells difficiles à manier et gourmands en temps. C'est l'une des raisons de la prolifération des interfaces de gestion graphiques. Cette tendance se retrouve également parmi les outils employés par les administrateurs de systèmes Windows. En effet, Microsoft s'est attaché à améliorer les interfaces de gestion graphiques aux dépens des interfaces en ligne de commande.

Aujourd'hui, les administrateurs Windows ont accès aux mêmes possibilités d'automation que leurs collègues UNIX et Linux. Cependant, PowerShell et son orientation objet permettent de satisfaire les besoins d'automation que les administrateurs Windows ont exprimés depuis les premiers jours des traitements par lots et de WSH : une plus grande facilité d'utilisation et moins de conversions. L'exemple suivant montre comment fonctionne le pipeline de commandes PowerShell :

```
PS C:\> get-process bash | format-table id -autosize

  Id
  --
3628


PS C:\>
```

À l'instar de l'exemple Bash, l'objectif est d'afficher le PID du processus `bash`. Tout d'abord, les informations concernant ce processus sont obtenues à l'aide de `Get-Process`. Ensuite, elles sont envoyées à l'applet de commande `Format-Table`, qui retourne un tableau contenant uniquement le PID du processus Bash.

L'exemple Bash exige l'écriture d'une commande shell complexe, contrairement à l'exemple PowerShell qui nécessite simplement la mise en forme d'un tableau. Comme vous pouvez le constater, la structure des applets de commandes PowerShell est beaucoup plus facile à comprendre et à utiliser.

Puisque nous disposons du PID du processus `bash`, voyons comment nous pouvons tuer (arrêter) ce processus :

```
PS C:\> get-process bash | stop-process
PS C:\>
```

Conseils .NET Framework

Avant de poursuivre, vous devez connaître quelques détails sur l'interaction entre PowerShell et .NET Framework. Ces informations sont essentielles à la compréhension des scripts qui seront présentés au cours des chapitres suivants.

Applet de commande New-Object

`New-Object` permet de créer une instance d'un objet .NET. Pour cela, nous précisons simplement le nom complet de la classe .NET de l'objet :

```
PS C:\> $Ping = new-object Net.NetworkInformation.Ping
PS C:\>
```

Grâce à `New-Object`, nous disposons d'une instance de la classe `Ping` qui permet de déterminer si un ordinateur distant peut être contacté *via* ICMP (*Internet Control Message Protocol*). Autrement dit, nous disposons d'une version objet de l'outil en ligne de commande `Ping.exe`.

Vous vous demandez peut-être quel est le remplaçant de la méthode VBScript `CreateObject` : il s'agit de l'applet de commande `New-Object`. Nous pouvons également employer l'option `comObject` de cette applet de commande pour créer un objet COM, simplement en précisant l'identificateur de programmes (ProgID) de l'objet :

```
PS C:\> $IE = new-object -comObject InternetExplorer.Application
PS C:\> $IE.Visible=$True
PS C:\> $IE.Navigate("www.france3.fr")
PS C:\>
```

Crochets

Dans ce livre, vous remarquerez l'usage des crochets ([et]), qui indiquent que le terme inclus est une référence .NET Framework. Voici les références valides :

- *un nom de classe complet*, par exemple `[System.DirectoryServices.ActiveDirectory.Forest]` ;
- *une classe de l'espace de noms* `System`, par exemple `[string]`, `[int]`, `[boolean]`, etc. ;
- un *type abrégé*, par exemple `[ADSI]`, `[WMI]`, `[Regex]`, etc.

INFO

Le Chapitre 8, "PowerShell et WMI", reviendra en détail sur les types abrégés.

La définition d'une variable est un bon exemple d'utilisation d'une référence .NET Framework. Dans le cas suivant, une énumération est affectée à une variable en utilisant une conversion explicite en une classe .NET :

```
PS C:\> $UnNombre = [int]1
PS C:\> $Identite = [System.Security.Principal.NTAccount]"Administrateur"
PS C:\>
```

Si une énumération ne peut être constituée que d'un ensemble figé de constantes, que nous ne connaissons pas, nous pouvons utiliser la méthode `GetNames` de la classe `System.Enum` pour obtenir cette information :

```
PS C:\> [enum]::GetNames([System.Security.AccessControl.FileSystemRights])
ListDirectory
ReadData
WriteData
CreateFiles
CreateDirectories
AppendData
ReadExtendedAttributes
WriteExtendedAttributes
Traverse
ExecuteFile
DeleteSubdirectoriesAndFiles
```

```
ReadAttributes
WriteAttributes
Write
Delete
ReadPermissions
Read
ReadAndExecute
Modify
ChangePermissions
TakeOwnership
Synchronize
FullControl
PS C:\>
```

Classes et méthodes statiques

Les crochets ne servent pas uniquement à définir des variables, mais également à utiliser ou à invoquer des membres statiques d'une classe .NET. Pour cela, il suffit de placer deux caractères deux-points (`::`) entre le nom de la classe et la méthode ou la propriété statique :

```
PS C:\> [System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest()


Name                   : taosage.internal
Sites                  : {HOME}
Domains                : {taosage.internal}
GlobalCatalogs         : {sol.taosage.internal}
ApplicationPartitions  : {DC=DomainDnsZones,DC=taosage,DC=internal, DC=ForestDns
                         Zones,DC=taosage,DC=internal}
ForestMode             : Windows2003Forest
RootDomain             : taosage.internal
Schema                 : CN=Schema,CN=Configuration,DC=taosage,DC=internal
SchemaRoleOwner        : sol.taosage.internal
NamingRoleOwner        : sol.taosage.internal


PS C:\>
```

Réflexion

La **réflexion** est une fonctionnalité .NET Framework qui permet aux développeurs d'examiner des objets et de retrouver leurs méthodes, leurs propriétés, leurs champs, etc. Puisque PowerShell s'appuie sur .NET Framework, il offre également cette possibilité, grâce à l'applet de commande `Get-Member`. Elle analyse l'objet ou l'ensemble d'objets que nous lui passons *via* le pipeline. Par exemple, la commande suivante analyse les objets retournés par `Get-Process` et affiche leurs propriétés et leurs méthodes :

```
PS C:\> get-process | get-member
```

Les développeurs désignent souvent ce processus sous le terme "interroger" un objet. Cette solution permet d'obtenir plus rapidement des informations sur des objets que d'invoquer l'applet de commande `Get-Help` (qui, au moment de l'écriture de ces lignes, fournit des informations limitées), de lire la documentation MSDN ou de faire une recherche sur Internet.

```
PS C:\> get-process | get-member


   TypeName: System.Diagnostics.Process

Name                        MemberType     Definition
----                        ----------     ----------
Handles                     AliasProperty  Handles = Handlecount
Name                        AliasProperty  Name = ProcessName
NPM                         AliasProperty  NPM = NonpagedSystemMemorySize
PM                          AliasProperty  PM = PagedMemorySize
VM                          AliasProperty  VM = VirtualMemorySize
WS                          AliasProperty  WS = WorkingSet
add_Disposed                Method         System.Void add_Disposed(Event...
add_ErrorDataReceived       Method         System.Void add_ErrorDataRecei...
add_Exited                  Method         System.Void add_Exited(EventHa...
add_OutputDataReceived      Method         System.Void add_OutputDataRece...
BeginErrorReadLine          Method         System.Void BeginErrorReadLine()
BeginOutputReadLine         Method         System.Void BeginOutputReadLine()
CancelErrorRead             Method         System.Void CancelErrorRead()
CancelOutputRead            Method         System.Void CancelOutputRead()
Close                       Method         System.Void Close()
CloseMainWindow             Method         System.Boolean CloseMainWindow()
CreateObjRef                Method         System.Runtime.Remoting.ObjRef...
Dispose                     Method         System.Void Dispose()
Equals                      Method         System.Boolean Equals(Object obj)
```

```
get_BasePriority              Method         System.Int32 get_BasePriority()
get_Container                 Method         System.ComponentModel.IContain...
get_EnableRaisingEvents       Method         System.Boolean get_EnableRaisi...
...
__NounName                    NoteProperty   System.String __NounName=Process
BasePriority                  Property       System.Int32 BasePriority {get;}
Container                     Property       System.ComponentModel.IContain...
EnableRaisingEvents           Property       System.Boolean EnableRaisingEv...
ExitCode                      Property       System.Int32 ExitCode {get;}
ExitTime                      Property       System.DateTime ExitTime {get;}
Handle                        Property       System.IntPtr Handle {get;}
HandleCount                   Property       System.Int32 HandleCount {get;}
HasExited                     Property       System.Boolean HasExited {get;}
Id                            Property       System.Int32 Id {get;}
MachineName                   Property       System.String MachineName {get;}
MainModule                    Property       System.Diagnostics.ProcessModu...
MainWindowHandle              Property       System.IntPtr MainWindowHandle...
MainWindowTitle               Property       System.String MainWindowTitle ...
MaxWorkingSet                 Property       System.IntPtr MaxWorkingSet {g...
MinWorkingSet                 Property       System.IntPtr MinWorkingSet {g...
...
Company                       ScriptProperty System.Object Company {get=$th...
CPU                           ScriptProperty System.Object CPU {get=$this.T...
Description                   ScriptProperty System.Object Description {get...
FileVersion                   ScriptProperty System.Object FileVersion {get...
Path                          ScriptProperty System.Object Path {get=$this....
Product                       ScriptProperty System.Object Product {get=$th...
ProductVersion                ScriptProperty System.Object ProductVersion {...

PS C:\>
```

Cet exemple montre que les objets retournés par `Get-Process` possèdent des propriétés que nous ne connaissions pas. L'exemple suivant utilise ces informations pour générer un rapport sur les processus appartenant à Microsoft, ainsi que leur emplacement :

```
PS C:\> get-process | where-object {$_.Company -match ".*Microsoft*"} |
format-table Name, ID, Path -Autosize

Name          Id Path
----          -- ----
ctfmon      4052 C:\WINDOWS\system32\ctfmon.exe
```

```
explorer   3024 C:\WINDOWS\Explorer.EXE
iexplore   2468 C:\Program Files\Internet Explorer\iexplore.exe
iexplore   3936 C:\Program Files\Internet Explorer\iexplore.exe
mobsync     280 C:\WINDOWS\system32\mobsync.exe
notepad    1600 C:\WINDOWS\system32\notepad.exe
notepad    2308 C:\WINDOWS\system32\notepad.exe
notepad    2476 C:\WINDOWS\system32\NOTEPAD.EXE
notepad    2584 C:\WINDOWS\system32\notepad.exe
OUTLOOK    3600 C:\Program Files\Microsoft Office\OFFICE11\OUTLOOK.EXE
powershell 3804 C:\Program Files\Windows PowerShell\v1.0\powershell.exe
WINWORD    2924 C:\Program Files\Microsoft Office\OFFICE11\WINWORD.EXE


PS C:\>
```

Une seule ligne de code WSH serait bien incapable d'obtenir ces informations sur le processus.

`Get-Member` n'est pas réservée aux objets générés par les applets de commande PowerShell. Nous pouvons également l'employer sur des objets initialisés à partir de classe .NET, par exemple :

```
PS C:\> new-object System.DirectoryServices.DirectorySearcher
```

La classe `DirectorySearcher` a pour fonction d'obtenir des informations sur un utilisateur depuis Active Directory, mais nous ne connaissons pas les méthodes prises en charge par les objets retournés. Pour connaître cette information, exécutons `Get-Member` sur une variable contenant les objets mystérieux :

```
PS C:\> $Recherche = new-object System.DirectoryServices.DirectorySearcher
PS C:\> $Recherche | get-member


   TypeName: System.DirectoryServices.DirectorySearcher

Name                        MemberType Definition
----                        ---------- ----------
add_Disposed                Method     System.Void add_Disposed(EventHandle...
CreateObjRef                Method     System.Runtime.Remoting.ObjRef Creat...
Dispose                     Method     System.Void Dispose()
Equals                      Method     System.Boolean Equals(Object obj)
FindAll                     Method     System.DirectoryServices.SearchResul...
```

```
FindOne                     Method     System.DirectoryServices.SearchResul...
...
Asynchronous                Property   System.Boolean Asynchronous {get;set;}
AttributeScopeQuery         Property   System.String AttributeScopeQuery {g...
CacheResults                Property   System.Boolean CacheResults {get;set;}
ClientTimeout               Property   System.TimeSpan ClientTimeout {get;s...
Container                   Property   System.ComponentModel.IContainer Con...
DerefAlias                  Property   System.DirectoryServices.Dereference...
DirectorySynchronization    Property   System.DirectoryServices.DirectorySy...
ExtendedDN                  Property   System.DirectoryServices.ExtendedDN ...
Filter                      Property   System.String Filter {get;set;}
PageSize                    Property   System.Int32 PageSize {get;set;}
PropertiesToLoad            Property   System.Collections.Specialized.Strin...
PropertyNamesOnly           Property   System.Boolean PropertyNamesOnly {ge...
ReferralChasing             Property   System.DirectoryServices.ReferralCha...
SearchRoot                  Property   System.DirectoryServices.DirectoryEn...
SearchScope                 Property   System.DirectoryServices.SearchScope...
SecurityMasks               Property   System.DirectoryServices.SecurityMas...
ServerPageTimeLimit         Property   System.TimeSpan ServerPageTimeLimit ...
ServerTimeLimit             Property   System.TimeSpan ServerTimeLimit {get...
Site                        Property   System.ComponentModel.ISite Site {ge...
SizeLimit                   Property   System.Int32 SizeLimit {get;set;}
Sort                        Property   System.DirectoryServices.SortOption ...
Tombstone                   Property   System.Boolean Tombstone {get;set;}
VirtualListView             Property   System.DirectoryServices.DirectoryVi...

PS C:\>
```

Vous noterez la présence de la méthode `FindAll` et de la propriété `Filter`. Il s'agit d'attributs d'objets qui peuvent servir à rechercher des informations concernant des utilisateurs dans un domaine Active Directory. La première étape, pour utiliser ces attributs, consiste à filtrer les informations renvoyées par `DirectorySearcher` grâce à la propriété `Filter`, qui attend une instruction de filtre analogue à celles employées avec LDAP (*Lightweight Directory Access Protocol*) :

```
PS C:\> $Recherche.Filter = ("(objectCategory=user)")
```

Ensuite, nous récupérons tous les utilisateurs du domaine Active Directory à l'aide de la méthode `FindAll` :

```
PS C:\> $Utilisateurs = $Recherche.FindAll()
```

À ce stade, la variable `$Utilisateurs` englobe une collection d'objets contenant les identifiants uniques (DN, *Distinguished Name*) de tous les utilisateurs du domaine Active Directory :

```
PS C:\> $Utilisateurs

Path                                      Properties
----                                      ----------
LDAP://CN=Administrator,CN=Users,DC=... {homemdb, samaccounttype, countrycod...
LDAP://CN=Guest,CN=Users,DC=taosage,... {samaccounttype, objectsid, whencrea...
LDAP://CN=krbtgt,CN=Users,DC=taosage... {samaccounttype, objectsid, whencrea...
LDAP://CN=admintyson,OU=Admin Accoun... {countrycode, cn, lastlogoff, usncre...
LDAP://CN=servmom,OU=Service Account... {samaccounttype, lastlogontimestamp,...
LDAP://CN=SUPPORT_388945a0,CN=Users,... {samaccounttype, objectsid, whencrea...
LDAP://CN=Tyson,OU=Acc... {msmqsigncertificates, distinguished...
LDAP://CN=Maiko,OU=Acc... {homemdb, msexchhomeservername, coun...
LDAP://CN=servftp,OU=Service Account... {samaccounttype, lastlogontimestamp,...
LDAP://CN=Erica,OU=Accounts,OU... {samaccounttype, lastlogontimestamp,...
LDAP://CN=Garett,OU=Accou... {samaccounttype, lastlogontimestamp,...
LDAP://CN=Fujio,OU=Accounts,O... {samaccounttype, givenname, sn, when...
LDAP://CN=Kiyomi,OU=Accounts,... {samaccounttype, givenname, sn, when...
LDAP://CN=servsql,OU=Service Account... {samaccounttype, lastlogon, lastlogo...
LDAP://CN=servdhcp,OU=Service Accoun... {samaccounttype, lastlogon, lastlogo...
LDAP://CN=servrms,OU=Service Account... {lastlogon, lastlogontimestamp, msmq...


PS C:\>
```

INFO

Les commandes de ces exemples utilisent les paramètres de connexion par défaut de la classe `DirectorySearcher`. Autrement dit, la connexion à Active Directory emploie le contexte de nommage par défaut. Si vous souhaitez établir une connexion à un domaine autre que celui par défaut, vous devez fixer les paramètres de connexion appropriés.

Nous disposons à présent d'un objet pour chaque utilisateur. Nous pouvons utiliser l'applet de commande `Get-Member` pour en savoir plus sur ces objets :

```
PS C:\> $Utilisateurs | get-member


   TypeName: System.DirectoryServices.SearchResult

Name              MemberType Definition
----              ---------- ----------
Equals            Method     System.Boolean Equals(Object obj)
get_Path          Method     System.String get_Path()
get_Properties    Method     System.DirectoryServices.ResultPropertyCollecti...
GetDirectoryEntry Method     System.DirectoryServices.DirectoryEntry GetDire...
GetHashCode       Method     System.Int32 GetHashCode()
GetType           Method     System.Type GetType()
ToString          Method     System.String ToString()
Path              Property   System.String Path {get;}
Properties        Property   System.DirectoryServices.ResultPropertyCollecti...


PS C:\>
```

Pour obtenir les informations concernant ces objets d'utilisateurs, il semblerait que nous devions les prendre tour à tour et invoquer leur méthode `GetDirectoryEntry`. Pour connaître les données que nous allons obtenir, appelons à nouveau `Get-Member` :

```
PS C:\> $Utilisateurs[0].GetDirectoryEntry() | get-member -MemberType Property


   TypeName: System.DirectoryServices.DirectoryEntry

Name                           MemberType Definition
----                           ---------- ----------
accountExpires                 Property   System.DirectoryServices.Property...
adminCount                     Property   System.DirectoryServices.Property...
badPasswordTime                Property   System.DirectoryServices.Property...
badPwdCount                    Property   System.DirectoryServices.Property...
cn                             Property   System.DirectoryServices.Property...
codePage                       Property   System.DirectoryServices.Property...
countryCode                    Property   System.DirectoryServices.Property...
description                    Property   System.DirectoryServices.Property...
displayName                    Property   System.DirectoryServices.Property...
distinguishedName              Property   System.DirectoryServices.Property...
```

```
homeMDB                          Property   System.DirectoryServices.Property...
homeMTA                          Property   System.DirectoryServices.Property...
instanceType                     Property   System.DirectoryServices.Property...
isCriticalSystemObject           Property   System.DirectoryServices.Property...
lastLogon                        Property   System.DirectoryServices.Property...
lastLogonTimestamp               Property   System.DirectoryServices.Property...
legacyExchangeDN                 Property   System.DirectoryServices.Property...
logonCount                       Property   System.DirectoryServices.Property...
mail                             Property   System.DirectoryServices.Property...
mailNickname                     Property   System.DirectoryServices.Property...
mDBUseDefaults                   Property   System.DirectoryServices.Property...
memberOf                         Property   System.DirectoryServices.Property...
msExchALObjectVersion            Property   System.DirectoryServices.Property...
msExchHomeServerName             Property   System.DirectoryServices.Property...
msExchMailboxGuid                Property   System.DirectoryServices.Property...
msExchMailboxSecurityDescriptor  Property   System.DirectoryServices.Property...
msExchPoliciesIncluded           Property   System.DirectoryServices.Property...
msExchUserAccountControl         Property   System.DirectoryServices.Property...
mSMQDigests                      Property   System.DirectoryServices.Property...
mSMQSignCertificates             Property   System.DirectoryServices.Property...
name                             Property   System.DirectoryServices.Property...
nTSecurityDescriptor             Property   System.DirectoryServices.Property...
objectCategory                   Property   System.DirectoryServices.Property...
objectClass                      Property   System.DirectoryServices.Property...
objectGUID                       Property   System.DirectoryServices.Property...
objectSid                        Property   System.DirectoryServices.Property...
primaryGroupID                   Property   System.DirectoryServices.Property...
proxyAddresses                   Property   System.DirectoryServices.Property...
pwdLastSet                       Property   System.DirectoryServices.Property...
sAMAccountName                   Property   System.DirectoryServices.Property...
sAMAccountType                   Property   System.DirectoryServices.Property...
showInAddressBook                Property   System.DirectoryServices.Property...
textEncodedORAddress             Property   System.DirectoryServices.Property...
userAccountControl               Property   System.DirectoryServices.Property...
uSNChanged                       Property   System.DirectoryServices.Property...
uSNCreated                       Property   System.DirectoryServices.Property...
whenChanged                      Property   System.DirectoryServices.Property...
whenCreated                      Property   System.DirectoryServices.Property...

PS C:\>
```

INFO

Le paramètre `MemberType` demande à `Get-Member` de retrouver un type de membre particulier. Par exemple, pour afficher les méthodes associées à un objet, utilisez la commande `get-member –MemberType Méthode`.

Si vous voulez réellement utiliser PowerShell, vous devez vous familiariser avec `Get-Member`. Si vous ne comprenez pas son fonctionnement, il vous sera parfois difficile de déterminer les possibilités d'un objet.

Nous savons à présent comment extraire des informations depuis Active Directory. Nous pouvons donc réunir toutes les commandes précédentes :

```
PS C:\> $Recherche = new-object System.DirectoryServices.DirectorySearcher
PS C:\> $Recherche.Filter = ("(objectCategory=user)")
PS C:\> $Utilisateurs = $Recherche.FindAll()
PS C:\> foreach ($Utilisateur in $Utilisateurs){$Utilisateur.GetDirectoryEntry()
.sAMAccountName}
Administrator
Guest
krbtgt
admintyson
servmom
SUPPORT_388945a0
Tyson
Maiko
servftp
Erica
Garett
Fujio
Kiyomi
servsql
servdhcp
servrms
PS C:\>
```

Bien que la liste des utilisateurs de ce domaine ne soit pas très longue, elle montre parfaitement que nous pouvons interroger un ensemble d'objets pour en comprendre les possibilités.

Nous pouvons faire de même pour les classes statiques. Mais, si l'on tente d'utiliser `Get-Member` comme nous l'avons fait précédemment, nous obtenons l'erreur suivante :

```
PS C:\> new-object System.Net.Dns
New-Object : Constructeur introuvable. Impossible de trouver un constructeur
approprié pour le type System.Net.Dns.
Au niveau de ligne : 1 Caractère : 11
+ new-object  <<<< System.Net.Dns
PS C:\>
```

Ainsi que vous le constatez, la classe `System.Net.Dns` ne possède pas de constructeur, ce qui ne nous facilite pas la tâche. Cependant, `Get-Member` sait également comment traiter ce cas. Avec le paramètre `Static`, nous pouvons obtenir les informations concernant les classes statiques :

```
PS C:\> [System.Net.Dns] | get-member -Static


   TypeName: System.Net.Dns

Name                   MemberType Definition
----                   ---------- ----------
BeginGetHostAddresses Method     static System.IAsyncResult BeginGetHostAddr...
BeginGetHostByName    Method     static System.IAsyncResult BeginGetHostByNa...
BeginGetHostEntry     Method     static System.IAsyncResult BeginGetHostEntr...
BeginResolve          Method     static System.IAsyncResult BeginResolve(Str...
EndGetHostAddresses   Method     static System.Net.IPAddress[] EndGetHostAdd...
EndGetHostByName      Method     static System.Net.IPHostEntry EndGetHostByN...
EndGetHostEntry       Method     static System.Net.IPHostEntry EndGetHostEnt...
EndResolve            Method     static System.Net.IPHostEntry EndResolve(IA...
Equals                Method     static System.Boolean Equals(Object objA, O...
GetHostAddresses      Method     static System.Net.IPAddress[] GetHostAddres...
GetHostByAddress      Method     static System.Net.IPHostEntry GetHostByAddr...
GetHostByName         Method     static System.Net.IPHostEntry GetHostByName...
GetHostEntry          Method     static System.Net.IPHostEntry GetHostEntry(...
GetHostName           Method     static System.String GetHostName()
ReferenceEquals       Method     static System.Boolean ReferenceEquals(Objec...
Resolve               Method     static System.Net.IPHostEntry Resolve(Strin...


PS C:\
```

Nous savons tout de la classe `System.Net.Dns` et nous pouvons donc la mettre en œuvre. Comme exemple, utilisons la méthode `GetHostAddress` afin de connaître l'adresse IP du site Web **www.digg.com** :

```
PS C:\> [System.Net.Dns]::GetHostAddresses("www.digg.com")


IPAddressToString : 64.191.203.30
Address           : 516669248
AddressFamily     : Internetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False


PS C:\>
```

INFO

Nous venons de le voir, l'applet de commande `Get-Member` peut être un outil très puissant. Elle peut également être pernicieuse car il est facile de passer des heures à explorer les possibilités des différentes applets de commande et des classes. Pour vous empêcher d'être victime du syndrome de l'utilisateur stressé de `Get-Member`, essayez de limiter vos sessions de découverte à deux heures par jour.

Système de types étendu

Vous pourriez penser que les scripts PowerShell n'utilisent aucun système de type car il est rarement nécessaire de préciser le type d'une variable. Mais c'est faux. En effet, PowerShell s'interface avec différents types d'objets issus de .NET, de WMI (*Windows Management Instrumentation*), de COM (*Component Object Model*), d'ADO (*ActiveX Data Objects*), d'ADSI (*Active Directory Service Interfaces*), de XML (*Extensible Markup Language*) et même d'objets personnels. En revanche, les types ne nous concernent généralement pas car PowerShell s'adapte aux différents types d'objets et affiche son interprétation d'un objet à notre place.

En quelque sorte, PowerShell tente de fournir une couche d'abstraction commune qui offre des interactions cohérentes avec les objets, malgré l'existence des types. Cette couche d'abstraction est `PSObject`. Il s'agit d'un objet commun employé pour tous les accès aux objets.

Il peut encapsuler n'importe quel objet de base (.NET, personnel, etc.), n'importe quel membre d'une instance et un accès implicite ou explicite aux membres étendus adaptés et du type, selon le type de l'objet de base.

Par ailleurs, il peut établir son type et ajouter des membres dynamiquement. Pour cela, PowerShell utilise le **système de types étendu** (ETS, *Extended Type System*), qui fournit une interface permettant aux développeurs d'applets de commande et de scripts de manipuler et de modifier les objets selon les besoins.

INFO

Lorsque vous utilisez `Get-Member`, les informations obtenues proviennent de `PSObject`. Il arrive que `PSObject` cache des membres, des méthodes et des propriétés de l'objet originel. Si vous souhaitez voir les informations bloquées, utilisez la propriété `BaseObject` dont le nom standard est `PSBase`. Par exemple, la commande `$Procs.PSBase | get-member` affiche les informations bloquées de la collection `$Procs`.

Bien entendu, cet aspect fait partie des sujets avancés, car `PSBase` n'est pas mentionnée. Vous devrez l'utiliser lorsque `PSObject` n'interprète pas correctement un objet ou lorsque vous voudrez étudier les aspects cachés de PowerShell.

Par conséquent, grâce au système de types étendu, nous pouvons modifier des objets en adaptant leur structure à nos besoins ou créer de nouveaux objets. Pour manipuler des objets, une solution consiste à adapter (étendre) des types existants ou à créer de nouveaux types d'objets. Pour cela, il faut définir les types personnalisés dans un fichier de type, dont la structure repose sur le fichier des types par défaut, `Types.ps1xml`.

Dans ce fichier, tous les types se trouvent dans un nœud `<Type></Type>` et chaque type peut contenir des membres standard, des membres de données et des méthodes d'objet. En nous appuyant sur cette structure, nous pouvons créer notre propre fichier de types personnalisés et le charger dans une session PowerShell grâce à l'applet de commande `Update-TypeData` :

```
PS C:\> Update-TypeData D:\PS\Mes.Types.Ps1xml
```

Cette commande doit être exécutée manuellement pour chaque session PowerShell ou ajoutée au fichier `profile.ps1`.

ATTENTION

Le fichier `Types.ps1xml` définit des comportements par défaut pour tous les objets de PowerShell. Vous ne devez en aucun cas le modifier, sinon vous pourriez empêcher le fonctionnement de PowerShell.

La seconde façon de manipuler la structure d'un objet passe par l'applet de commande `Add-Member`. Elle permet d'ajouter un membre défini par l'utilisateur à une instance d'objet existante :

```
PS C:\> $Procs = get-process
PS C:\> $Procs | add-member -Type scriptProperty "TotalDays" {
>> $Date = get-date
>> $Date.Subtract($This.StartTime).TotalDays}
>>
PS C:\>
```

Ce code crée un membre `scriptProperty` nommé `TotalDays` pour la collection d'objets contenue dans la variable `$Procs`. Le membre `scriptProperty` peut ensuite être appelé comme n'importe quel autre membre de ces objets :

INFO

Lorsque vous créez une méthode de script, la variable `$This` représente l'objet courant.

```
PS C:\> $Procs | where {$_.name -Match "WINWORD"} | ft Name,TotalDays -
AutoSize

Name             TotalDays
----             ---------
WINWORD 5,1238899696898148


PS C:\>
```

Même si le membre `scriptProperty` n'est pas particulièrement utile, il montre bien comment étendre un objet. Cette capacité d'extension des objets est extrêmement utile, que ce soit pour l'écriture d'un script ou le développement d'une applet de commande.

Comprendre les fournisseurs

La plupart des systèmes informatiques servent à stocker des données, généralement dans une structure comme un système de fichiers. Étant donné le volume de données enregistré dans ces structures, le traitement et la recherche d'informations peuvent être complexes. Les shells offrent généralement des interfaces, ou **fournisseurs**, pour interagir avec les magasins de données de manière prédéfinie. PowerShell dispose également d'un ensemble de fournisseurs pour présenter le contenu des magasins de données, par le biais d'un jeu d'applets de commande principales. Nous pouvons nous en servir pour parcourir et manipuler les données enregistrées au travers d'une interface commune. La commande suivante affiche la liste des applets principales :

```
PS C:\> help about_core_commands
…
    APPLETS DE COMMANDE ChildItem
    Get-ChildItem

    APPLETS DE COMMANDE CONTENT
    Add-Content
    Clear-Content
    Get-Content
    Set-Content

    APPLETS DE COMMANDE DRIVE
    Get-PSDrive
    New-PSDrive
    Remove-PSDrive

    APPLETS DE COMMANDE ITEM
    Clear-Item
    Copy-Item
    Get-Item
    Invoke-Item
    Move-Item
    New-Item
    Remove-Item
    Rename-Item
    Set-Item

    APPLETS DE COMMANDE LOCATION
    Get-Location
```

```
    Pop-Location
    Push-Location
    Set-Location

    APPLETS DE COMMANDE PATH
    Join-Path
    Convert-Path
    Split-Path
    Resolve-Path
    Test-Path

    APPLETS DE COMMANDE PROPERTY
    Clear-Itemproperty
    Copy-ItemProperty
    Get-ItemProperty
    Move-ItemProperty
    New-ItemProperty
    Remove-ItemProperty
    Rename-ItemProperty
    Set-ItemProperty

    APPLETS DE COMMANDE PROVIDER
    Get-PSProvider


PS C:\>
```

La commande suivante affiche les fournisseurs PowerShell prédéfinis :

```
PS C:\> get-psprovider

Name                Capabilities                Drives
----                ------------                ------
Alias               ShouldProcess               {Alias}
Environment         ShouldProcess               {Env}
FileSystem          Filter, ShouldProcess       {C, D, E, F...}
Function            ShouldProcess               {Function}
Registry            ShouldProcess               {HKLM, HKCU}
Variable            ShouldProcess               {Variable}
Certificate         ShouldProcess               {cert}


PS C:\>
```

Cette liste présente non seulement les fournisseurs intégrés, mais également les lecteurs reconnus par chacun d'eux. Un **lecteur** est une entité utilisée par un fournisseur pour représenter un magasin de données et au travers duquel ces données sont rendues disponibles à la session PowerShell. Par exemple, le fournisseur Registry crée un lecteur PowerShell pour les clés `HKEY_LOCAL_MACHINE` et `HKEY_CURRENT_USER`.

La commande suivante affiche tous les lecteurs PowerShell reconnus :

```
PS C:\> get-psdrive

Name        Provider      Root
----        --------      ----
Alias       Alias
C           FileSystem    C:\
cert        Certificate   \
D           FileSystem    D:\
E           FileSystem    E:\
Env         Environment
F           FileSystem    F:\
Function    Function
G           FileSystem    G:\
HKCU        Registry      HKEY_CURRENT_USER
HKLM        Registry      HKEY_LOCAL_MACHINE
U           FileSystem    U
Variable    Variable


PS C:\>
```

Accéder aux lecteurs et aux données

Pour accéder aux lecteurs PowerShell et à leurs données, une solution consiste à utiliser l'applet de commande `Set-Location`. Elle modifie l'emplacement de travail en sélectionnant celui qui est indiqué, lequel peut être un répertoire, un sous-répertoire, une pile d'emplacements ou un emplacement dans le Registre :

```
PS C:\> set-location hklm:
PS HKLM:\> set-location software\microsoft\windows
PS HKLM:\software\microsoft\windows>
```

`Get-ChildItem` permet ensuite d'afficher les sous-clés de la clé `Windows` :

```
PS HKLM:\software\microsoft\windows> get-childitem


   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software\micros
oft\windows

SKC  VC Name                           Property
---  -- ----                           --------
 55  13 CurrentVersion                 {DevicePath, MediaPathUnexpanded, SM_...
  0  16 Help                           {PINTLPAD.HLP, PINTLPAE.HLP, IMEPADEN...
  0  36 Html Help                      {PINTLGNE.CHM, PINTLGNT.CHM, PINTLPAD...
  1   0 ITStorage                      {}
  0   0 Shell                          {}


PS HKLM:\software\microsoft\windows>
```

Vous remarquerez qu'avec le fournisseur Registry, `Get-ChildItem` donne uniquement la liste des sous-clés d'une clé, sans les valeurs du Registre. En effet, les valeurs du Registre sont considérées comme des propriétés d'une clé, non comme un élément valide.

Pour obtenir ces valeurs, nous devons utiliser l'applet de commande `Get-ItemProperty` :

```
PS HKLM:\software\microsoft\windows> get-itemproperty currentversion


PSPath                    : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHI
                            NE\software\microsoft\windows\currentversion
PSParentPath              : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHI
                            NE\software\microsoft\windows
PSChildName               : currentversion
PSDrive                   : HKLM
PSProvider                : Microsoft.PowerShell.Core\Registry
DevicePath                : C:\WINDOWS\inf
MediaPathUnexpanded       : C:\WINDOWS\Media
SM_GamesName              : Jeux
SM_ConfigureProgramsName  : Configurer les programmes par défaut
ProgramFilesDir           : C:\Program Files
```

```
CommonFilesDir           : C:\Program Files\Fichiers communs
ProductId                : 76487-OEM-0011903-00101
WallPaperDir             : C:\WINDOWS\Web\Wallpaper
MediaPath                : C:\WINDOWS\Media
ProgramFilesPath         : C:\Program Files
SM_AccessoriesName       : Accessoires
PF_AccessoriesName       : Accessoires


PS HKLM:\software\microsoft\windows>
```

Comme pour `Get-Process`, les données obtenues sont une collection d'objets que nous pouvons modifier afin d'obtenir la sortie souhaitée :

```
PS HKLM:\software\microsoft\windows> get-itemproperty currentversion |
select ProductId

ProductId
---------
76487-OEM-XXXXXXX-XXXXX


PS HKLM:\software\microsoft\windows>
```

L'accès aux données d'un fournisseur `FileSystem` est tout aussi simple. La même logique de commandes change l'emplacement et affiche la structure :

```
PS HKLM:\software\microsoft\windows> set-location c:
PS C:\> set-location "C:\WINDOWS\system32\windowspowershell\v1.0\fr"
PS C:\WINDOWS\system32\windowspowershell\v1.0\fr> get-childitem about_a*

    Répertoire : Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\system32\
    windowspowershell\v1.0\fr

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-----        19/09/2006     09:03       6397 about_alias.help.txt
-----        19/09/2006     09:03       3774 about_arithmetic_operators.help.txt
-----        19/09/2006     09:03       9403 about_array.help.txt
```

```
-----        19/09/2006     09:03      17090 about_assignment_operators.help.txt
-----        19/09/2006     09:03       6227 about_associative_array.help.txt
-----        19/09/2006     09:03       4597 about_automatic_variables.help.txt

PS C:\WINDOWS\system32\windowspowershell\v1.0\fr>
```

Dans ce cas, les données sont stockées dans un élément au lieu d'en être des propriétés. Pour obtenir les données d'un élément, nous devons invoquer l'applet `Get-Content` :

```
PS C:\WINDOWS\system32\windowspowershell\v1.0\fr> get-content about_Alias.help.txt
RUBRIQUE
    Alias

DESCRIPTION COURTE
    Utilisation d'autres noms pour les applets de commande et les commandes
    dans Windows PowerShell

DESCRIPTION LONGUE
    Un alias est un autre nom ou surnom utilisé pour une applet de commande ou un
    élément de commande, tel qu'une fonction, un script, un fichier ou un fichier
    exécutable. Vous pouvez utiliser l'alias au lieu du nom de la commande. Par
    exemple, si vous établissez l'alias " gas " pour Get-AuthenticodeSignature,
    vous pouvez taper :

        gas c:\scripts\sqlscript.ps1
...

PS C:\WINDOWS\system32\windowspowershell\v1.0\fr>
```

INFO

Tous les lecteurs ne s'appuient pas sur un magasin de données hiérarchique. Par exemple, les fournisseurs Environment, Function et Variable ne sont pas hiérarchiques. L'accès aux données par le biais de ces fournisseurs se fait dans l'emplacement racine du lecteur associé.

Monter un lecteur

Il est possible de créer et de supprimer des lecteurs PowerShell, ce qui est très pratique lorsqu'un emplacement ou un ensemble d'emplacements est souvent utilisé. Au lieu de changer l'emplacement ou d'employer un chemin absolu, nous pouvons créer de nouveaux lecteurs (ou "monter un lecteur" dans le jargon PowerShell) qui sont des raccourcis vers ces emplacements.

Pour cela, nous utilisons `New-PSDrive` :

```
PS C:\> new-psdrive -name PSScripts -root D:\Dev\Scripts -psp FileSystem

Name       Provider      Root                           CurrentLocation
----       --------      ----                           ---------------
PSScripts  FileSystem    D:\Dev\Scripts


PS C:\> get-psdrive

Name       Provider      Root                           CurrentLocation
----       --------      ----                           ---------------
Alias      Alias
C          FileSystem    C:\
cert       Certificate   \
D          FileSystem    D:\
E          FileSystem    E:\
Env        Environment
F          FileSystem    F:\
Function   Function
G          FileSystem    G:\
HKCU       Registry      HKEY_CURRENT_USER              software
HKLM       Registry      HKEY_LOCAL_MACHINE             ...crosoft\windows
PSScripts  FileSystem    D:\Dev\Scripts
U          FileSystem    U:\
Variable   Variable


PS C:\>
```

La suppression d'un lecteur se fait avec l'applet de commande `Remove-PSDrive` :

```
PS C:\> remove-psdrive -name PSScripts
PS C:\> get-psdrive

Name       Provider      Root                           CurrentLocation
----       --------      ----                           ---------------
Alias      Alias
C          FileSystem    C:\
cert       Certificate   \
D          FileSystem    D:\
E          FileSystem    E:\
Env        Environment
F          FileSystem    F:\
```

```
Function   Function
G          FileSystem    G:\
HKCU       Registry      HKEY_CURRENT_USER              software
HKLM       Registry      HKEY_LOCAL_MACHINE             ...crosoft\windows
U          FileSystem    U:\
Variable   Variable


PS C:\>
```

Comprendre les erreurs

Les erreurs PowerShell se répartissent en deux catégories : fatales et non fatales. Comme leur nom le sous-entend, les **erreurs fatales** stoppent l'exécution d'une commande. Les **erreurs non fatales** sont généralement signalées sans que la commande soit arrêtée. Lorsque des erreurs se produisent, quel que soit leur type, elles sont ajoutées dans la variable `$Error`. Cette collection contient les erreurs générées pendant la session PowerShell en cours. L'erreur la plus récente se trouve dans `$Error[0]` et le nombre maximal d'erreurs est défini par `$MaximumErrorCount`, qui vaut par défaut 256.

Les erreurs contenues dans la variable `$Error` variable peuvent être représentées par l'objet `ErrorRecord`. Il détient les informations d'exception, ainsi que d'autres propriétés permettant de comprendre l'origine d'une erreur.

L'exemple suivant montre les informations qui se trouvent dans la propriété `InvocationInfo` d'un objet `ErrorRecord` :

```
PS C:\> $Error[0].InvocationInfo


MyCommand        : Get-ChildItem
ScriptLineNumber : 1
OffsetInLine     : -2147483648
ScriptName       :
Line             : dir z:
PositionMessage  :
                   Au niveau de ligne : 1 Caractère : 4
                   + dir  <<<< z:
InvocationName   : dir
PipelineLength   : 1
PipelinePosition : 1


PS C:\>
```

Grâce à ces informations, nous pouvons déterminer certains détails concernant `$Error[0]`, notamment la commande qui a provoqué l'erreur. Ces informations sont essentielles pour comprendre les erreurs et les gérer efficacement.

La commande suivante affiche la liste complète des propriétés d'`ErrorRecord` :

```
PS C:\> $Error[0] | get-member -MemberType Property


   TypeName: System.Management.Automation.ErrorRecord

Name                   MemberType Definition
----                   ---------- ----------
CategoryInfo           Property   System.Management.Automation.ErrorCategoryI...
ErrorDetails           Property   System.Management.Automation.ErrorDetails E...
Exception              Property   System.Exception Exception {get;}
FullyQualifiedErrorId  Property   System.String FullyQualifiedErrorId {get;}
InvocationInfo         Property   System.Management.Automation.InvocationInfo...
TargetObject           Property   System.Object TargetObject {get;}


PS C:\>
```

Le Tableau 3.1 récapitule les définitions des propriétés d'`ErrorRecord` affichées par l'exemple précédent.

Tableau 3.1 Définitions des propriétés d'`ErrorRecord`

Propriété	Définition
`CategoryInfo`	Indique la catégorie de l'erreur
`ErrorDetails`	Lorsqu'elle n'est pas nulle, elle fournit des informations supplémentaires concernant l'erreur
`Exception`	L'erreur qui s'est produite
`FullyQualifiedErrorId`	Identifie plus précisément une condition d'erreur
`InvocationInfo`	Lorsqu'elle n'est pas nulle, elle décrit le contexte dans lequel l'erreur s'est produite
`TargetObject`	Lorsqu'elle n'est pas nulle, elle indique l'objet cible de l'opération

Gérer les erreurs

Les méthodes de gestion des erreurs dans PowerShell vont de la plus simple à la plus complexe. La plus simple consiste à laisser PowerShell traiter l'erreur. Selon le type de l'erreur, la commande ou le script peut se terminer ou se poursuivre. Cependant, si le gestionnaire d'erreurs par défaut ne répond pas aux besoins, nous pouvons mettre en place une gestion d'erreurs plus complexe en employant les méthodes décrites dans les sections suivantes.

Méthode 1 : préférences d'une applet de commande

Dans PowerShell, certains paramètres sont disponibles pour toutes les applets de commande. En particulier, les paramètres `ErrorAction` et `ErrorVariable` fixent le traitement des erreurs *non fatales* :

```
PS C:\> get-childitem z: -ErrorVariable Err -ErrorAction SilentlyContinue
PS C:\> if ($Err){write-host $Err -Foregroundcolor Red}
Lecteur introuvable. Il n'existe aucun lecteur nommé « z ».
PS C:\>
```

Le paramètre `ErrorAction` définit le comportement d'une applet de commande lorsqu'elle rencontre une erreur *non fatale*. Dans l'exemple précédent, ce paramètre est fixé à `SilentlyContinue`. Autrement dit, l'applet de commande poursuit son exécution sans afficher les erreurs *non fatales* qui peuvent se produire. Voici les autres options d'`ErrorAction` :

- `Continue`. Afficher l'erreur et poursuivre l'exécution (action par défaut).
- `Inquire`. Demander à l'utilisateur s'il souhaite poursuivre, arrêter ou suspendre l'exécution.
- `Stop`. Stopper l'exécution de la commande ou du script.

INFO

L'expression non fatale est mise en exergue dans cette section car une erreur fatale ne tient pas compte du paramètre `ErrorAction` et est passée au gestionnaire d'erreurs par défaut ou personnalisé.

Le paramètre `ErrorVariable` définit le nom de la variable pour l'objet d'erreur généré par une erreur *non fatale*. Dans l'exemple précédent, `ErrorVariable` est fixé à `Err`. Vous remarquerez que le nom de la variable n'inclut pas le préfixe `$`. Cependant, pour accéder à

`ErrorVariable` en dehors d'une applet de commande, le préfixe `$` est indispensable (`$Err`). Par ailleurs, après que `ErrorVariable` a été définie, la variable résultante est valide dans la session PowerShell en cours ou le bloc de script associé. Autrement dit, d'autres applets de commande peuvent ajouter des objets d'erreur à une variable d'erreur existante en utilisant le préfixe `+` :

```
PS C:\> get-childitem z: -ErrorVariable Err -ErrorAction SilentlyContinue
PS C:\> get-childitem y: -ErrorVariable +Err -ErrorAction SilentlyContinue
PS C:\> write-host $Err[0] -Foregroundcolor Red
Lecteur introuvable. Il n'existe aucun lecteur nommé « z ».
PS C:\> write-host $Err[1] -Foregroundcolor Red
Lecteur introuvable. Il n'existe aucun lecteur nommé « y ».
PS C:\>
```

Méthode 2 : interception des erreurs

En cas d'erreur fatale, PowerShell affiche par défaut l'erreur et stoppe l'exécution de la commande ou du script. Pour mettre en place une gestion personnalisée des erreurs fatales, il faut définir un gestionnaire d'exception qui empêche que l'erreur fatale (`ErrorRecord`) ne soit envoyée au mécanisme par défaut. Cette procédure s'applique également aux erreurs *non fatales*, car PowerShell affiche par défaut l'erreur et poursuit la commande ou le script.

Pour définir une interception, nous utilisons la syntaxe suivante :

```
trap ExceptionType {code; mot-clé}
```

La première partie, *`ExceptionType`*, précise le type d'erreur accepté par l'interception. S'il n'est pas précisé, toutes les erreurs sont interceptées. La partie *`code`*, facultative, peut être une commande ou un ensemble de commandes qui sont exécutées une fois l'erreur envoyée au gestionnaire. La dernière partie, *`mot-clé`*, précise si l'interception autorise la poursuite de l'exécution du bloc de code où l'erreur s'est produite ou bien si elle doit être stoppée.

Voici les mots clés reconnus :

- `Break`. L'exception est à nouveau levée et l'exécution de la portée en cours s'arrête.
- `Continue`. L'exécution de la portée courante peut se poursuivre à partir de la ligne qui se trouve après celle où s'est produite l'exception.
- `Return [argument]`. L'exécution de la portée courante est arrêtée et l'argument est retourné, s'il est précisé.

Si aucun mot clé n'est donné, le gestionnaire utilise `Return [argument]` ; *argument* étant l'objet `ErrorRecord` initialement passé au gestionnaire.

Exemples d'interception

Les deux exemples suivants montrent comment définir des gestionnaires d'erreurs. Le premier illustre une interception d'erreur *non fatale* produite lorsqu'un nom de DNS invalide est passé à la classe `System.Net.Dns`. Le second exemple présente à nouveau l'interception d'une erreur *non fatale* générée par l'applet de commande `Get-Item`. Cependant, dans ce cas, puisque le paramètre `ErrorAction` a été fixé à `Stop`, l'erreur est en réalité une erreur fatale qui est traitée par le gestionnaire.

Exemple 1 : `traperreur1.ps1`

```
$NomDNS = "www.-nomdnsinvalide-.com"

trap [System.Management.Automation.MethodInvocationException]{
    write-host ("ERREUR : " + $_) -Foregroundcolor Red; Continue}

write-host "Obtenir l'adresse IP de" $NomDNS
write-host ([System.Net.Dns]::GetHostAddresses("www.$nomdnsinvalide$.com"))
write-host "Terminé"
```

Dans cet exemple, le paramètre `$_` représente l'objet `ErrorRecord` qui a été passé au gestionnaire.

Voici la sortie produite par cet exemple :

```
PS C:\> .\traperreur1.ps1
Obtenir l'adresse IP de www.-nomdnsinvalide-.com
ERREUR : Exception lors de l'appel de « GetHostAddresses » avec « 1 »
argument(s) : « Hôte inconnu »
Terminé
PS C:\>
```

INFO

Une applet de commande ne génère pas d'erreur fatale sauf en cas d'erreur de syntaxe. Autrement dit, un gestionnaire n'intercepte aucune erreur non fatale produite par une applet de commande, sauf si l'erreur est transformée en une erreur fatale après que le paramètre `ErrorAction` de l'applet a été fixé à `Stop`.

Exemple 2 : `traperreur2.ps1`

```
write-host "Changer le lecteur pour z:"

trap {write-host("[ERREUR] " + $_) -Foregroundcolor Red; Continue}

get-item z: -ErrorAction Stop
$FichiersTXT = get-childitem *.txt -ErrorAction Stop

write-host "Terminé"
```

Voici la sortie produite par cet exemple :

```
PS C:\> .\traperreur2.ps1
Changer le lecteur pour z:
[ERREUR] L'exécution de la commande s'est arrêtée, car la variable
d'environnement « ErrorActionPreference » a la valeur Stop : Lecteur
introuvable. Il n'existe aucun lecteur nommé « z ».
Terminé
PS C:\>
```

Portées d'interception

Comme nous l'avons expliqué au Chapitre 2, "Les fondamentaux de PowerShell", dans PowerShell, une portée détermine l'exécution des interceptions. En général, un gestionnaire d'erreurs est défini et exécuté au sein de la même portée. Par exemple, nous pouvons définir un gestionnaire dans une certaine portée et, lorsqu'une erreur fatale se produit dans cette portée, il est exécuté. Si la portée en cours ne contient aucun gestionnaire d'erreurs mais s'il en existe un dans une portée extérieure, les erreurs fatales rencontrées sortent de la portée en cours et sont passées au gestionnaire dans la portée externe.

Méthode 3 : mot clé throw

Dans PowerShell, nous pouvons générer nos propres erreurs fatales. Cela ne signifie pas provoquer des erreurs en utilisant une syntaxe invalide, mais générer exprès une erreur fatale en utilisant le mot clé `throw`. Ainsi, l'exemple suivant génère une erreur lorsqu'un utilisateur exécute le script `MonParam.ps1` sans définir le paramètre `MonParam`. Cette possibilité est très utile lorsque des données provenant de fonctions, d'applets de commande, de sources de données, d'applications, etc., ne sont pas celles attendues et peuvent donc empêcher l'exécution correcte d'un script ou d'un ensemble de commandes.

Voici le script :

```
param([string]$MonParam = $(throw write-host "Le paramètre MonParam n'a pas
été défini" -Foregroundcolor Red))

write-host $MonParam
```

Et sa sortie :

```
PS C:\> .\MonParam.ps1
Le paramètre MonParam n'a pas été défini
ScriptHalted
Au niveau de C:\MonParam.ps1 : 1 Caractère : 34
+ param([string]$MonParam = $(throw  <<<< write-host "Le paramètre MonParam
n'a pas été défini" -Foregroundcolor Red))
PS C:\>
```

Profils

Un **profil** est un ensemble enregistré de paramètres qui personnalisent l'environnement PowerShell. Il existe quatre types de profils, chargés dans un ordre précis à chaque démarrage de PowerShell. La section suivante explique ces types de profils, où ils doivent être placés et l'ordre de leur chargement.

Le profil Tous les utilisateurs

Ce profil se trouve dans le fichier `%windir%\system32\windowspowershell\v1.0\profile.ps1`. Les paramètres qui y sont définis sont appliqués à tous les utilisateurs PowerShell sur la machine courante. Si vous voulez configurer PowerShell pour l'ensemble des utilisateurs d'une machine, vous devez modifier ce profil.

Le profil Tous les utilisateurs pour un hôte spécifique

Ce profil se trouve dans le fichier `%windir%\system32\windowspowershell\v1.0\`*`IdShell`*`_profile.ps1`. Les paramètres qui y sont définis sont appliqués à tous les utilisateurs du shell en cours (par défaut la console PowerShell). PowerShell reconnaît le concept de shells, ou hôtes, multiples. Par exemple, la console PowerShell est un hôte, que la majorité des utilisateurs emploient exclusivement. Cependant, d'autres applications peuvent démarrer une instance

de PowerShell afin d'exécuter des commandes et des scripts PowerShell. Il s'agit alors d'une **application hôte**, qui utilise un profil particulier pour contrôler la configuration de PowerShell. Le nom du profil d'hôte spécifique inclut l'identifiant du shell. Dans la console PowerShell, cet identifiant est le suivant :

```
PS C:\> $ShellId
Microsoft.PowerShell
PS C:\>
```

En réunissant ces noms, le profil Tous les utilisateurs pour un hôte spécifique de la console PowerShell s'appelle donc `Microsoft.PowerShell_profile.ps1`. Pour les autres hôtes, l'identifiant du shell et les noms des profils sont différents. Par exemple, l'outil PowerShell Analyzer (**www.powershellanalyzer.com**) est un hôte qui offre une interface graphique élaborée pour PowerShell. Son identifiant de shell est `PowerShellAnalyzer.PSA` et son profil Tous les utilisateurs pour un hôte spécifique est `PowerShellAnalyzer.PSA_profile.ps1`.

Le profil Utilisateur courant

Ce profil se trouve dans le fichier `%userprofile%\Mes Documents\WindowsPowerShell\profile.ps1`. L'utilisateur qui souhaite définir ses propres paramètres de profil peut le faire dans ce fichier. Les paramètres sont appliqués uniquement à sa session PowerShell courante et n'affectent pas les autres utilisateurs.

Le profil Utilisateur courant pour un hôte spécifique

Ce profil se trouve dans le fichier `%userprofile%\My Documents\WindowsPowerShell\IdShell_profile.ps1`. Comme pour le profil Tous les utilisateurs pour un hôte spécifique, ce type de profil charge les paramètres uniquement pour le shell en cours. Cependant, ils sont propres à l'utilisateur.

INFO

Lorsque vous démarrez le shell pour la première fois, un message signalant que les scripts sont désactivés et qu'aucun profil n'est chargé peut s'afficher. Vous pouvez changer ce comportement en modifiant la stratégie d'exécution de PowerShell (voir la section suivante).

Comprendre la sécurité

Lorsque WSH est arrivé avec Windows 98, il a été accueilli comme une aubaine par les administrateurs Windows qui souhaitaient disposer des mêmes possibilités d'automation que leurs homologues UNIX. Dans le même temps, les créateurs de virus ont rapidement découvert que WSH constituait également un vecteur d'attaque des systèmes Windows.

Il est possible d'automatiser et de contrôler pratiquement l'intégralité d'un système Windows à l'aide de WSH, ce qui constitue un grand avantage pour les administrateurs. En revanche, WSH n'apporte aucune sécurité dans l'exécution des scripts. Lorsqu'on lui désigne un script, WSH l'exécute. L'origine du script et son rôle n'ont aucune importance. C'est pourquoi WSH s'est rapidement fait connaître comme un trou de sécurité et non comme un outil d'automation.

Stratégies d'exécution

À cause des critiques concernant la sécurité de WSH, l'équipe de développement de PowerShell a décidé d'inclure une stratégie d'exécution qui réduit les menaces potentielles d'un code malveillant. Une stratégie d'exécution contraint l'exécution des scripts et le chargement des fichiers de configuration dans PowerShell. Il en existe quatre, détaillées au cours des sections suivantes : `Restricted`, `AllSigned`, `RemoteSigned` et `Unrestricted`.

Stratégie Restricted

Par défaut, PowerShell utilise la stratégie d'exécution `Restricted`. Elle est la plus sécurisée car PowerShell ne peut alors fonctionner qu'en mode interactif. Autrement dit, aucun script ne peut être lancé et seuls les fichiers de configuration signés numériquement par un éditeur de confiance peuvent être exécutés ou chargés.

Stratégie AllSigned

La stratégie d'exécution `AllSigned` est moins contraignante que `Restricted`. Lorsqu'elle est activée, seuls les scripts et les fichiers de configuration signés numériquement par un éditeur de confiance peuvent être exécutés ou chargés. Voici un exemple de sortie obtenue lorsque la stratégie `AllSigned` est active :

```
PS C:\Scripts> .\MonScript.ps1
Impossible de charger le fichier C:\Scripts\MonScript.ps1. Le fichier C:\Scripts\
MonScript.ps1 n'est pas signé numériquement. Le script ne sera pas exécuté sur
le système. Pour plus d'informations, consultez « get-help about_signing »..
Au niveau de ligne : 1 Caractère : 24
+ .\MonScript.ps1 <<<<
PS C:\Scripts>
```

La signature d'un script ou d'un fichier de configuration exige un certificat de signature de code. Ce certificat peut provenir d'une autorité de certification (CA, *Certificate Authority*) ou nous pouvons en générer un avec l'outil de création d'un certificat (`Makecert.exe`). Cependant, il est généralement préférable d'obtenir un certificat de signature de code auprès d'une autorité de certification reconnue, comme Verisign, Thawte ou l'infrastructure à clés publiques (PKI, *Public Key Infrastructure*) de votre entreprise. Dans le cas contraire, le partage de vos scripts ou de vos fichiers de configuration risque d'être plus difficile car votre ordinateur n'est pas, par défaut, une autorité de certification approuvée.

INFO

Le Chapitre 4, "Signer du code", explique comment obtenir un certificat de signature du code valide et digne de confiance. Nous conseillons fortement de le lire car la signature numérique des scripts et des fichiers de configuration est un sujet extrêmement important.

Stratégie RemoteSigned

La stratégie d'exécution `RemoteSigned` est conçue de manière à empêcher l'exécution ou le chargement automatique des scripts et des fichiers de configuration PowerShell distants qui n'ont pas été signés numériquement par un éditeur de confiance. Les scripts et les fichiers de configuration qui ont été créés localement peuvent être chargés et exécutés, même s'ils n'ont pas été signés.

Un script ou un fichier de configuration distant peuvent être obtenus à partir d'une application de communication, comme Microsoft Outlook, Internet Explorer, Outlook Express ou Windows Messenger. L'exécution ou le chargement d'un fichier fourni par l'une de ces applications produisent le message d'erreur suivant :

```
PS C:\Scripts> .\SonScript.ps1
Impossible de charger le fichier C:\Scripts\SonScript.ps1. Le fichier
C:\Scripts\SonScript.ps1 n'est pas signé numériquement. Le script ne sera
pas exécuté sur le système. Pour plus d'informations, consultez « get-help
about_signing »..
Au niveau de ligne : 1 Caractère : 24
+ .\SonScript.ps1 <<<<
PS C:\Scripts>
```

Pour exécuter ou charger un script ou un fichier de configuration distant non signé, il faut indiquer que le fichier est digne de confiance. Pour cela, cliquez du bouton droit sur le fichier dans l'explorateur Windows et choisissez Propriétés. Dans l'onglet Général, cliquez sur le bouton Débloquer (voir Figure 3.1).

Après que le fichier a été approuvé, le script ou le fichier de configuration peut être exécuté ou chargé. S'il est signé numériquement, mais si l'éditeur n'est pas de confiance, PowerShell affiche l'invite suivante :

```
PS C:\Scripts> .\ScriptSigne.ps1

Voulez-vous exécuter le logiciel de cet éditeur non approuvé ?
Le fichier C:\Scripts\ScriptSigne.ps1 est publié par CN=companyabc.com, OU=IT,
O=companyabc.com, L=Oakland, S=California, C=US et n'est pas approuvé sur
votre système. N'exécutez que des scripts provenant d'éditeurs approuvés.
[M] Ne jamais exécuter  [N] Ne pas exécuter  [O] Exécuter une fois
[T] Toujours exécuter[?] Aide (la valeur par défaut est « N ») :
```

Dans ce cas, nous devons décider si nous pouvons faire confiance ou non au contenu du fichier.

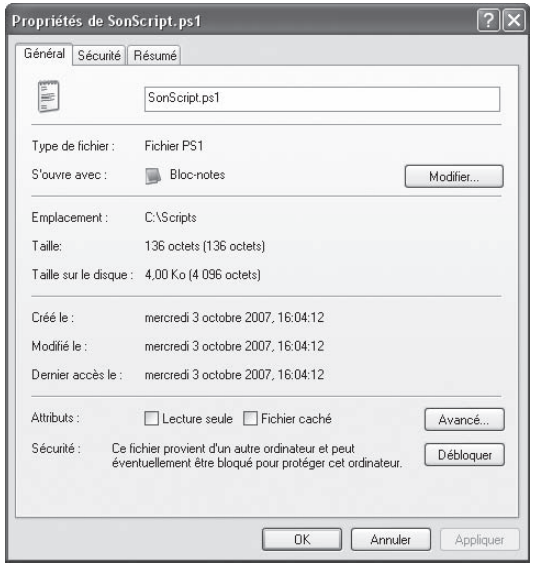


Figure 3.1

Faire confiance à un script ou un fichier de configuration distant.

INFO

Le Chapitre 4 reviendra en détail sur les options de cette invite.

Stratégie Unrestricted

Comme son nom le suggère, la stratégie d'exécution `Unrestricted` retire pratiquement toutes les restrictions d'exécution des scripts ou de chargement des fichiers de configuration. Tous les fichiers locaux ou signés peuvent être exécutés ou chargés, mais, pour les fichiers distants, PowerShell vous demande de choisir ce qu'il doit faire :

```
PS C:\Scripts> .\ScriptDistant.ps1

Avertissement de sécurité
N'exécutez que des scripts que vous approuvez. Bien que les scripts en
provenance d'Internet puissent être utiles, ce script est susceptible
d'endommager votre ordinateur. Voulez-vous exécuter
C:\Scripts\ScriptDistant.ps1 ?
[N] Ne pas exécuter  [O] Exécuter une fois  [S] Suspendre  [?] Aide
(la valeur par défaut est « N ») :
```

Fixer la stratégie d'exécution

Pour changer de stratégie d'exécution, nous devons employer l'applet de commande `Set-ExecutionPolicy` :

```
PS C:\> set-executionpolicy AllSigned
PS C:\>
```

L'applet de commande `Get-ExecutionPolicy` affiche la stratégie d'exécution en place :

```
PS C:\> get-executionpolicy
AllSigned
PS C:\>
```

Lors de l'installation de PowerShell, la stratégie d'exécution est fixée par défaut à `Restricted`. Comme vous le savez, les configurations par défaut ne sont pas conservées très longtemps. Par ailleurs, si PowerShell est installé sur de nombreuses machines, la probabilité que la stratégie d'exécution passe à `Unrestricted` augmente rapidement.

Par chance, il est possible de contrôler la stratégie d'exécution de PowerShell par le biais du Registre. Ce paramètre est une valeur de type `REG_SZ` baptisée `ExecutionPolicy`, qui se trouve dans la clé `HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell`. Ce contrôle de la stratégie d'exécution par le biais du Registre signifie qu'il est possible d'imposer une stratégie sur les machines gérées par une stratégie de groupe (GPO, *Group Policy Object*).

Par le passé, la création d'un GPO pour contrôler la stratégie d'exécution était simple car l'installation de PowerShell incluait un modèle d'administration (ADM, *Administrative Template*). Cependant, depuis la version PowerShell RC2, le modèle d'administration ne fait plus partie de l'installation et n'est pas forcément disponible dans un téléchargement séparé. Si Microsoft ne fournit aucun ADM pour contrôler la stratégie d'exécution, vous pouvez toujours créer le vôtre, comme le montre l'exemple suivant :

```
CLASS MACHINE

CATEGORY !!PowerShell
    POLICY !!Security
        KEYNAME "SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell"
        EXPLAIN !!PowerShell_ExecutionPolicy

        PART !!ExecutionPolicy EDITTEXT REQUIRED
            VALUENAME "ExecutionPolicy"
        END PART
    END POLICY
END CATEGORY

[strings]
PowerShell=PowerShell
Security=Paramètres de sécurité
PowerShell_ExecutionPolicy=Si elle est activée, cette stratégie définira la
stratégie d'exécution de PowerShell sur une machine à la valeur indiquée.
Les valeurs de stratégie d'exécution sont Restricted, AllSigned, RemoteSigned
ou Unrestricted.
Executionpolicy=Stratégie d'exécution
```

Une version opérationnelle de cet ADM est disponible sur la page consacrée à *Windows PowerShell*, à l'adresse **www.pearsoneducation.fr**. Même si le fichier `PowerShellExecutionPolicy.adm` a été testé et doit fonctionner dans votre environnement, sachez que les paramètres de stratégie d'exécution qu'il contient sont considérés comme des préférences. Les paramètres de préférence sont des GPO qui sont des valeurs du Registre placées en dehors

des arborescences de stratégies de groupe approuvées. Lorsqu'un GPO qui contient des paramètres de préférence quitte son étendue, les paramètres de préférence ne sont pas retirés sur Registre.

INFO

Comme pour tous les fichiers disponibles sur la page Web dédiée à Windows PowerShell, essayez le modèle d'administration dans un environnement de test avant de déployer un GPO qui l'utilise.

Pour configurer le fichier `PowerShellExecutionPolicy.adm`, procédez comme suit :

1. Connectez-vous sur une machine de gestion de la stratégie de groupe en tant qu'administrateur GPO.
2. À l'aide de la console MMC de stratégie de groupe (GPMC), créez un GPO nommé PowerShell.
3. Dans l'arborescence de la console, ouvrez Configuration ordinateur et ensuite Modèles d'administration.
4. Cliquez du bouton droit sur Modèles d'administration et choisissez Ajout/Suppression de modèles dans le menu contextuel.
5. Allez dans le dossier qui contient le fichier `PowerShellExecutionPolicy.adm`. Sélectionnez ce fichier, cliquez sur Ouvrir, puis sur Fermer. Le nœud PowerShell s'affiche alors sous le nœud Modèles d'administration.
6. Cliquez sur le nœud Modèles d'administration, puis sur Affichage > Filtrage dans le menu de la MMC stratégie de groupe. Décochez la case Afficher uniquement les paramètres de stratégie pouvant être entièrement gérés. Vous pourrez ainsi administrer les paramètres de préférences.
7. Cliquez ensuite sur le nœud PowerShell sous Modèles d'administration.
8. Dans le volet de détails, cliquez du bouton droit sur Paramètres de sécurité et sélectionnez Propriétés dans le menu contextuel.
9. Cliquez sur Activé.
10. Fixez la stratégie d'exécution à l'une des valeurs suivantes : Restricted, AllSigned, RemoteSigned ou Unrestricted.
11. Fermez le GPO, ainsi que la MMS stratégie de groupe.

Contrôler la stratégie d'exécution par le biais d'un paramètre de préférence de GPO ne semble sans doute pas une solution parfaite. En effet, un paramètre de préférence n'offre pas le même niveau de sécurité qu'un paramètre de stratégie d'exécution et les utilisateurs disposant des droits adéquats peuvent le modifier facilement. Ce manque de sécurité est probablement la raison du retrait du fichier ADM originel de PowerShell. Une version future de PowerShell pourrait autoriser le contrôle de la stratégie d'exécution par le biais d'un paramètre de stratégie de GPO valide.

Mesures de sécurité complémentaires

Les stratégies d'exécution ne sont pas la seule mesure de sécurité implémentée par Microsoft dans PowerShell. Les fichiers de scripts PowerShell qui ont l'extension `.ps1` ne peuvent pas être exécutés à partir de l'Explorateur Windows car ils sont associés à Bloc-notes. Autrement dit, nous ne pouvons pas simplement double-cliquer sur un fichier `.ps1` pour l'exécuter. Les scripts PowerShell doivent être lancés depuis une session PowerShell en utilisant un chemin relatif ou absolu ou depuis l'invite de commandes Windows en utilisant l'exécutable de PowerShell.

Par ailleurs, comme nous l'avons expliqué au Chapitre 2, pour exécuter ou ouvrir un fichier du répertoire courant depuis la console PowerShell, il est nécessaire de préfixer la commande par `.\` ou `./`. Cela évite que des utilisateurs PowerShell lancent accidentellement une commande ou un script sans avoir précisé explicitement son exécution.

Enfin, par défaut, il n'existe aucune méthode pour se connecter à PowerShell ou l'invoquer à distance. Cependant, cela ne signifie pas qu'il est impossible d'écrire une application qui accepte les connexions distantes à PowerShell. En réalité, cela existe déjà. Si vous souhaitez apprendre comment procéder, téléchargez PowerShell Remoting depuis le site **www.codeplex.com/powershellremoting**.

Langage

Sur ce point, ce livre diffère des autres ouvrages sur les langages de scripts, qui tentent d'expliquer les concepts de l'écriture des scripts au lieu de montrer des scripts opérationnels. Nous voulons mettre l'accent sur les applications pratiques de PowerShell.

Nous supposons que l'écriture de scripts ne vous est pas totalement étrangère. Par ailleurs, puisque le langage de PowerShell est analogue à Perl, C# et même VBScript, il n'y a aucune raison de perdre du temps à présenter les boucles `for`, les instructions `if...then` et les autres aspects fondamentaux de la programmation.

Il est vrai que certains points sont spécifiques au langage de PowerShell, mais vous pouvez alors consulter sa documentation. Ce livre n'est pas un ouvrage de référence du langage, mais s'attache à illustrer l'application de PowerShell à des cas réels. Pour une information détaillée sur le langage de PowerShell, vous pouvez consulter le Guide de l'utilisateur disponible *via* le menu Démarrer.

En résumé

Vous venez de faire connaissance de manière plus approfondie avec PowerShell et son fonctionnement. Vous avez abordé des sujets comme les fournisseurs, le traitement des erreurs, les profils et les stratégies d'exécution. Cependant, de tous les points étudiés, le concept le plus important à retenir est que PowerShell s'appuie sur .NET Framework. Par conséquent, il ne ressemble pas aux autres shells car il est orienté objet et il tente de présenter tous les objets sous une forme commune qui peut être utilisée sans autre modification dans les commandes et les scripts. À partir de là et avec les connaissances acquises aux Chapitres 2 et 3, vous allez explorer l'écriture de scripts PowerShell. Les scripts des chapitres suivants vont devenir de plus en plus complexes, au fur et à mesure de la présentation des différents aspects de l'automation de Windows avec PowerShell.

4

Signer du code

Dans ce chapitre

- Introduction
- Qu'est-ce que la signature du code ?
- Obtenir un certificat de signature du code
- Signer des scripts
- Vérifier des signatures numériques
- Distribuer du code signé

Introduction

Pour apprendre à signer les scripts et les fichiers de configuration de PowerShell, vous avez passé du temps à faire des recherches sur Internet, lu plusieurs blogs qui traitent de ce sujet, consulté la documentation PowerShell et même parcouru plusieurs livres sur PowerShell. Plus vous vous renseignez sur la signature du code, moins cela vous semble clair. Pour finir, vous ouvrez votre console PowerShell et saisissez la commande suivante :

```
set-executionpolicy unrestricted
```

Avant d'entrer cette commande, n'oubliez pas ce que vous avez appris à propos des stratégies d'exécution au Chapitre 3, "Présentation avancée de PowerShell". Le paramètre `Unrestricted` annihile une couche de sécurité importante conçue pour empêcher l'exécution du code malveillant sur votre système. La signature du code est une autre composante essentielle de la sécurité de PowerShell, mais nombreux sont ceux à la croire trop complexe et à utiliser une stratégie d'exécution `Unrestricted` pour ne pas avoir à l'employer. En réponse à un billet

concernant la signature de code sur le blog de Scott Hanselman (**www.hanselman.com/blog**), un internaute a fait le commentaire suivant : "La gestion des certificats de signature du code dépasse les capacités de nombreux utilisateurs, y compris des développeurs et des administrateurs normaux." Cette réponse montre bien un réel besoin d'explications. Ce chapitre se consacre donc à la signature du code. En surface, cela semble complexe, mais, avec des explications claires, la procédure est facile à comprendre. Les créateurs de scripts, les développeurs et les administrateurs doivent se familiariser avec elle, car il s'agit d'un élément essentiel de la mise en place d'une sécurité.

Qu'est-ce que la signature du code ?

En bref, la **signature du code** consiste à signer numériquement des scripts, des fichiers exécutables, des DLL, etc., afin de donner un certain niveau de confiance au code, laquelle repose sur deux hypothèses. Premièrement, un code signé garantit qu'il n'a pas été modifié ou corrompu depuis sa signature. Deuxièmement, la signature numérique sert à prouver l'identité de l'auteur du code, ce qui doit aider l'utilisateur à savoir s'il peut exécuter le code en toute sécurité.

Ces deux hypothèses sont un moyen de garantir l'intégrité et l'authenticité du code. Cependant, à elles seules, elles n'assurent pas la fiabilité du code signé. Pour qu'elles soient valides, nous avons besoin de la signature numérique et de l'infrastructure qui définit un mécanisme d'identification de l'origine de la signature.

Les signatures numériques reposent sur une cryptographie à clés publiques, avec des algorithmes de chiffrement et de déchiffrement. Ces algorithmes génèrent une paire de clés constituée d'une clé privée et d'une clé publique. La clé privée reste secrète et seul son propriétaire y a accès. La clé publique peut être donnée à d'autres entités, au travers d'une forme d'interaction sécurisée. Selon le type d'interaction, une clé sert à verrouiller (chiffrer) la communication et l'autre sert à la débloquer (déchiffrer). Dans le cas des signatures numériques, la clé privée est utilisée pour générer la signature, tandis que la clé publique permet de valider la signature générée. Voici la procédure mise en œuvre :

1. Un code de hachage à sens unique du contenu (document, code, etc.) à signer est généré en utilisant un condensat (*digest*) cryptographique.
2. Le code de hachage est ensuite chiffré à l'aide de la clé privée, afin d'obtenir la signature numérique.
3. Le contenu est ensuite envoyé au destinataire.

4. Le destinataire crée un autre code de hachage à sens unique du contenu et déchiffre le code de hachage à l'aide de la clé publique de l'émetteur.
5. Le destinataire compare les deux codes de hachage. S'ils sont identiques, la signature numérique est valide et le contenu n'a pas été modifié.

INFO

Une fonction de hachage à sens unique (également appelée condensat de message, empreinte ou somme de contrôle) est un algorithme cryptographique qui convertit des données en une suite binaire de longueur fixe. Le terme "à sens unique" vient du fait qu'il est difficile de retrouver les données originelles à partir de la suite obtenue.

Pour associer une entité, comme un organisme, une personne ou un ordinateur, à une signature numérique, un certificat numérique est utilisé. Il est constitué de la clé publique et des informations d'identification du propriétaire de la paire de clés. Pour en garantir l'intégrité, il est également signé numériquement. Un certificat numérique peut être signé par son propriétaire ou par un tiers de confiance appelé **autorité de certification** (CA, *Certificate Authority*).

L'association d'un code à l'entité qui l'a créé et publié supprime l'anonymat de l'exécution du code. Par ailleurs, l'association d'une signature numérique à un certificat de signature du code peut être comparée à l'utilisation d'une marque pour établir une relation de confiance et de fiabilité. Pourvus de ces informations, les utilisateurs de scripts et de fichiers de configuration PowerShell peuvent faire des choix en toute connaissance de cause sur l'exécution d'un script ou le chargement de fichiers de configuration. En résumé, c'est pour cela que la signature du code est importante pour la sécurité dans PowerShell.

Obtenir un certificat de signature du code

Il existe deux manières d'obtenir un certificat de signature du code : générer des certificats auto-signés et utiliser une autorité de certification dans une infrastructure à clés publiques (PKI, *Public Key Infrastructure*) valide.

Un certificat auto-signé est plus simple et plus rapide à générer. Il a également l'avantage d'être gratuit. Cependant, aucun tiers n'en vérifie l'authenticité, ce qui n'offre donc pas le niveau de confiance attendue pour une signature de code. Par conséquent, aucune autre entité ne fera par défaut confiance à votre certificat. Pour distribuer votre script ou votre fichier de

configuration PowerShell sur d'autres machines, votre certificat doit être ajouté en tant qu'autorité principale de confiance et en tant qu'éditeur approuvé.

Bien qu'il soit possible de modifier les éléments approuvés, il existe deux problèmes. Premièrement, les entités situées hors de votre zone d'influence peuvent décider de ne pas faire confiance à votre certificat car rien ne leur permet de vérifier votre identité. Deuxièmement, si la clé privée associée au certificat auto-signé vient à être compromise ou invalide, il est impossible de gérer la validité du certificat chez les autres entités. À cause de ces inconvénients, les certificats auto-signés ne doivent être employés que sur une machine locale ou dans une phase de test.

Si vous envisagez de signer numériquement vos scripts et vos fichiers de configuration afin de les utiliser dans une entreprise ou de les rendre publics, vous devez choisir la seconde méthode d'obtention d'un certificat de signature du code : une CA depuis une PKI valide. Une PKI valide peut être une entreprise commerciale bien connue et digne de confiance, comme **www.globalsign.net**, **www.thawte.com** ou **www.verisign.com**, ou une infrastructure interne qui appartient et qui est gérée par votre société. Si vous respectez quelques mises en garde, l'obtention d'un certificat de signature du code depuis une PKI externe peut être rapide et simple.

Tout d'abord, un certificat doit être acheté auprès du propriétaire de la PKI externe. Ensuite, puisque l'achat du certificat se fait auprès de l'identité externe, cela signifie que vous placez une grande confiance dans l'intégrité de cet organisme. C'est pourquoi les certificats de signature du code acquis auprès de PKI commerciales doivent se limiter aux certificats utilisés pour signer des scripts et les fichiers de configuration destinés à une distribution publique.

De ce fait, une PKI interne doit être utilisée pour les scripts et les fichiers de configuration réservés à un usage privé. N'oubliez pas que son déploiement et sa gestion demandent une organisation, des efforts et de l'argent (les modules matériels de sécurité, les consultants en sécurité, etc., peuvent être très coûteux). La plupart des entreprises ont tendance à éviter d'en mettre en place. Elles préfèrent installer des CA *ad hoc*, acheter des certificats auprès de PKI commerciales ou ignorer les besoins. Une PKI commerciale n'apportera peut-être pas le niveau de confiance nécessaire à votre entreprise et l'approche *ad hoc* est déconseillée car elle diminue le crédit des certificats générés par des CA illégitimes, c'est-à-dire dont l'intégrité n'est pas totalement garantie. Ne pas avoir de PKI valide risque de compliquer la distribution interne des fichiers signés numériquement. Enfin, les entreprises qui ignorent les besoins en PKI illustrent un autre inconvénient de la mise en place d'une PKI interne : le temps.

Si votre entreprise ne possède pas sa PKI, l'obtention d'un certificat de signature du code peut demander beaucoup de temps. Cela ne se fait pas en une nuit. Si vous avez identifié un besoin de PKI pour vos scripts, vous en découvrirez probablement d'autres au sein de votre société. Il faut d'abord les identifier et les étudier. La mise en place d'une PKI autour de vos besoins propres n'est pas la meilleure technique pour un service qui doit satisfaire les besoins de toute une entreprise. Après avoir présenté vos besoins de PKI, vous devrez sans doute attendre que les services soient opérationnels. Cependant, une fois la PKI en place, vous pouvez obtenir des certificats de signature du code en sachant que l'infrastructure prend totalement en charge la distribution de vos scripts et de vos fichiers de configuration PowerShell signés.

Méthode 1 : certificat auto-signé

La création d'un certificat auto-signé s'appuie sur l'utilitaire makecert fourni avec le kit de développement (SDK, *Software Development Kit*) pour .NET Framework. En voici les étapes :

1. Téléchargez la dernière version du Microsoft .NET Framework SDK à l'adresse **http://msdn2.microsoft.com/fr-fr/netframework/aa569263.aspx**. Au moment de l'écriture de ces lignes, il s'agit de la version 3.0.
2. Installez le SDK sur la machine où vous souhaitez générer le certificat auto-signé.
3. Repérez l'emplacement de l'outil makecert sur votre système. Par défaut, il s'agit du répertoire C:\Program Files\Microsoft SDKs\Windows\v6.0.
4. Ouvrez une invite de commande Windows et allez dans le répertoire de makecert à l'aide de la commande `cd`.
5. Créez un certificat auto-signé :

```
makecert -r -pe -n "CN=NomCommunAutoriteCertification" -b 01/01/2000
-e 01/01/2099 –eku 1.3.6.1.5.5.7.3.3 -ss My
```

Vous devez obtenir le résultat suivant :

```
C:\Program Files\Microsoft SDKs\Windows\v6.0>makecert -r -pe -n "CN=Mon autorite
de signature du code" -b 01/01/2000 -e 01/01/2099 -eku 1.3.6.1.5.5.7.3.3 -ss My
Succeeded
```

6. Enfin, saisissez la commande PowerShell suivante pour vérifier l'installation du certificat :

```
PS C:\> get-childitem cert:\CurrentUser\My -codesign

    Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint                                Subject
----------                                -------
944E910757A862B53DE3113249E12BCA9C7DD0DE  CN=Mon autorite de signature du code

PS C:\>
```

Méthode 2 : certificat signé par une CA

Cette méthode s'appuie sur l'obtention d'un certificat à partir d'une autorité de certification Microsoft Windows. Ces étapes supposent qu'une PKI a été déployée dans votre entreprise. Si ce n'est pas le cas, l'installation des services de certificats Windows pour répondre à votre besoin immédiat est déconseillée. Procédez comme suit pour obtenir un certificat de signature du code :

1. Demandez à votre administrateur PKI de créer et d'activer un modèle de certificat de signature du code pour vos scripts et vos fichiers de configuration PowerShell.
2. À partir d'Internet Explorer, allez sur le site des services d'inscription Web de l'autorité de certification à l'adresse **https://*NomServeurCA*/certsrv** (en remplaçant *NomServeurCA* par le nom de votre serveur).
3. Cliquez sur le lien Demander un certificat.
4. Sur la page Demander un certificat, cliquez sur le lien Demande de certificat avancée.
5. Sur la page Demande de certificat avancée, cliquez sur le lien Créer et soumettre une demande de requête auprès de cette Autorité de certification.
6. Dans la section Modèle de certificat, sélectionnez le certificat de signature du code créé par votre administrateur PKI.
7. Saisissez les informations d'identification manquantes et les options de demande de certificat conformément à la stratégie de votre entreprise. Vous pouvez utiliser la Figure 4.1 comme aide.
8. Cliquez sur le bouton Envoyer.
9. Dans la boîte de message Violation de script potentielle qui s'affiche (voir Figure 4.2), cliquez sur Oui pour continuer.

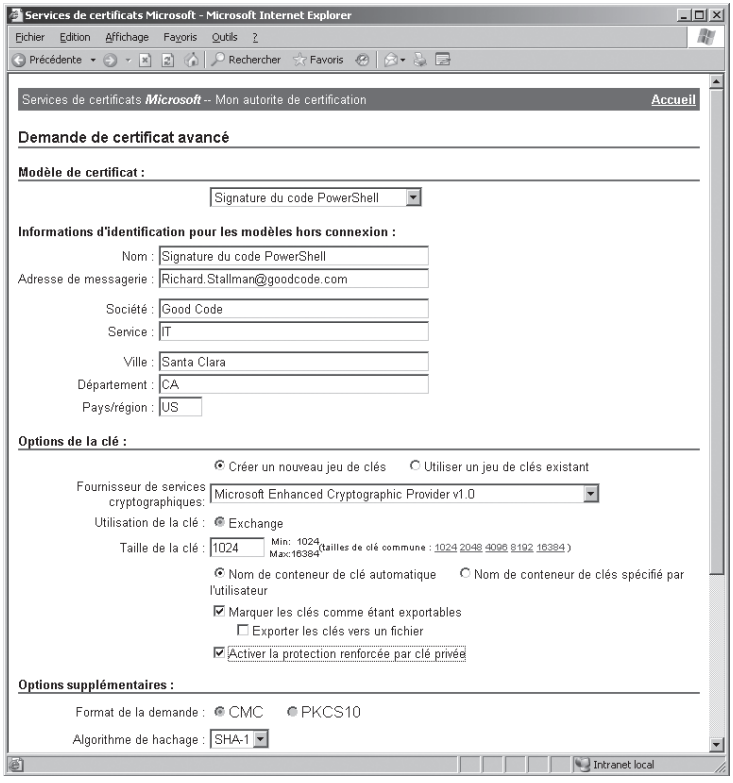


Figure 4.1

Exemple de demande de certificat de signature du code.

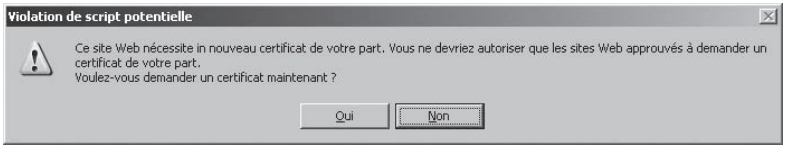


Figure 4.2

Boîte de message Violation de script potentielle.

10. Ensuite, si nécessaire, fixez le niveau de sécurité de la clé privée en fonction de la stratégie mise en place dans votre entreprise (voir Figure 4.3), puis cliquez sur OK.
11. Si la stratégie de votre entreprise exige l'approbation d'un administrateur de certificats, demandez-lui d'approuver la demande que vous venez de soumettre. Sinon, passez directement à l'étape 16.
12. Après l'approbation de la demande de certificat, utilisez Internet Explorer pour accéder au site des services d'inscription Web de l'autorité de certification à l'adresse **https://*NomServeurCA*/certsrv** (en remplaçant *NomServeurCA* par le nom de votre serveur).
13. Cliquez sur le lien Afficher le statut d'une requête de certificat en attente.

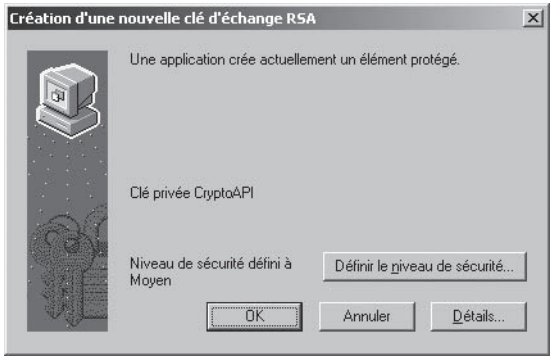


Figure 4.3

Boîte de dialogue Création d'une nouvelle clé d'échange RSA.

14. Sur la page suivante, cliquez sur le lien de demande adéquat.
15. Sur la page Certificat émis, cliquez sur le lien Installer ce certificat.
16. Dans la boîte de message Violation de script potentielle qui s'affiche (voir Figure 4.4), cliquez sur Oui pour continuer.

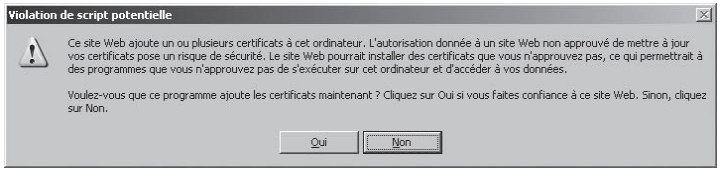


Figure 4.4

Boîte de message Violation de script potentielle.

17. Enfin, le site des services d'inscription Web de l'autorité de certification signale que le certificat a été installé avec succès. Saisissez la commande PowerShell suivante pour vérifier son installation :

```
PS C:\> get-childitem cert:\CurrentUser\My -codesign


    Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My


Thumbprint                                Subject
----------                                -------
5CBCE258711676061836BC45C1B4ACA6F6C7D09E  E=Richard.Stallman@goodcode.com, C...


PS C:\>
```

L'outil d'importation de fichiers de certificats

Lorsqu'un certificat numérique est généré, la clé privée est parfois enregistrée dans un fichier PVK (*Private Key*) et le certificat numérique correspondant est placé dans un fichier SPC (*Software Publishing Certificate*). Par exemple, si le certificat de signature du code a été obtenu auprès de Verisign ou de Thawte, il est envoyé au client sous la forme d'un fichier SPC et d'un fichier PVK. Si vous souhaitez employer ce certificat pour signer numériquement des scripts ou des fichiers de configuration PowerShell, vous devez importer les fichiers SPC et PVK dans votre magasin de certificats personnels.

INFO

Un magasin de certificats est un emplacement sur votre ordinateur ou sur un périphérique dans lequel sont stockées les informations concernant les certificats. Sous Windows, vous pouvez utiliser la console MMC Autorité de certification pour afficher le magasin d'un utilisateur, d'un ordinateur ou d'un service. Votre magasin de certificats personnels fait référence à votre propre magasin d'utilisateur.

Pour importer la combinaison SPC+PVK, vous devez utiliser l'outil de Microsoft appelé PVK Digital Certificate Files Importer. Il est disponible sur le site de téléchargement de Microsoft à l'adresse **www.microsoft.com/downloads/details.aspx?FamilyID=F9992C94-B129-46BC-B240-414BDFF679A7&displaylang=EN**.

Ensuite, saisissez la commande suivante pour importer les fichiers SPC et PVK, en indiquant vos propres fichiers :

```
pvkimprt -IMPORT "moncertificat.spc" "macleprivee.pvk"
```

Signer des scripts

La signature d'un script PowerShell se fait à l'aide de l'applet de commande `Set-AuthenticodeSignature`, qui attend deux paramètres obligatoires. Le premier, `filePath`, est le chemin et le nom de fichier du script à signer numériquement. Le second, `certificate`, désigne le certificat X.509 utilisé pour signer le script. Pour que le certificat X.509 soit dans un

format reconnu par `Set-AuthenticodeSignature`, nous devons l'obtenir par le biais de `Get-ChildItem` :

```
PS C:\Scripts> set-authenticodesignature –filePath ScriptSigne.ps1 -
certificate @(get-childitem cert:\CurrentUser\My -codeSigningCert)[0] -
includeChain "All"


    Répertoire : C:\Scripts


SignerCertificate                          Status            Path
----------------                           ------            ----
661BC0A2A11850CEF4862F97D7335B40FCCCCF06   Valid             ScriptSigne.ps1


PS C:\>
```

Pour obtenir le certificat depuis votre propre magasin, vous devez utiliser l'applet de commande `Get-ChildItem` avec le paramètre `codeSigningCert`. Ce paramètre ne peut être employé qu'avec le fournisseur `Certificate` et joue le rôle de filtre afin d'obliger `Get-ChildItem` à ne récupérer que les certificats de signature du code. Enfin, pour être certain que l'intégralité de la chaîne du certificat est incluse dans la signature numérique, le paramètre `includeChain` est défini.

Après l'exécution de l'applet de commande `Set-AuthenticodeSignature`, le fichier signé possède un bloc de signature valide qui contient la signature numérique. Un bloc de signature se trouve toujours à la fin du script ou du fichier de configuration PowerShell. Il est facile à identifier car il est placé entre les balises `SIG # Begin signature block` et `SIG # End signature block` :

```
write-host ("Ce script est signé !") -Foregroundcolor Green

# SIG # Begin signature block
# MIIEGwYJKoZIhvcNAQcCoIIEDDCCBAgCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUSuRNU62C2nvVTbf/JyWBXqC6
# puigggIyMIICLjCCAZegAwIBAgIQnewYWKhDlYdNnO6cjYQWBDANBgkqhkiG9w0B
...
# AQQBgjcCARUwIwYJKoZIhvcNAQkEMRYEFA+YleMX8BYxLif8zxiKQi5T1QR3MA0G
# CSqGSIb3DQEBAQUABIGApcdFZz2vnLWFbFYIgsjFsCTgPgAgOCa7iAVsyXz+Z/ga
# LwbgwtZpqIZhczQQ4UezAooaUPMkBMKhpJ2XITiFLgDDf8bAnPVxuxoLbmO9iH8Z
# weDJypYlMKe5ion5+S3Ahm3h92UnTk+kXav7m20bdLSw8x+R4yS2z2pL+0iGaX4=
# SIG # End signature block
```

INFO

Ce processus de signature numérique des scripts s'applique également aux fichiers de configuration PowerShell. Comme nous l'avons expliqué au Chapitre 3, les fichiers de configuration, selon la stratégie d'exécution en place, peuvent également avoir besoin d'être signés pour être chargés dans une session PowerShell.

Vérifier des signatures numériques

Pour vérifier la signature d'un script ou d'un fichier de configuration PowerShell, nous devons utiliser l'applet de commande `Get-AuthentiCodeSignature`. Elle retourne un état de validité, comme `HashMismatch`, qui indique l'existence ou non d'un problème avec le fichier.

État valide :

```
PS C:\Scripts> Get-AuthenticodeSignature ScriptSigne.ps1


    Répertoire : C:\Scripts


SignerCertificate                          Status            Path
-----------------                          ------            ----
661BC0A2A11850CEF4862F97D7335B40FCCCCF06   Valid             ScriptSigne.ps1


PS C:\Scripts> .\ScriptSigne.ps1
Ce script est signé!
PS C:\Scripts>
```

État invalide :

```
PS C:\Scripts> Get-AuthenticodeSignature ScriptSigne.ps1


    Répertoire : C:\Scripts


SignerCertificate                          Status            Path
-----------------                          ------            ----
661BC0A2A11850CEF4862F97D7335B40FCCCCF06   HashMismatch      ScriptSigne.ps1
```

```
PS C:\Scripts> .\ScriptSigne.ps1
Impossible de charger le fichier C:\Scripts\ScriptSigne.ps1. Le contenu du fichier
C:\Scripts\ScriptSigne.ps1 peut avoir été falsifié, car le hachage du fichier
ne correspond pas à celui qui figure dans la signature numérique. Le scriptne
sera pas exécuté sur le système. Pour plus d'informations, consultez « get-help
about_signing »..
Au niveau de ligne : 1 Caractère : 26
+ C:\Scripts\ScriptSigne.ps1 <<<<
PS C:\>
```

D'après l'erreur affichée par l'exemple précédent, le script a été modifié, falsifié ou corrompu. S'il a été modifié par son propriétaire, il doit être à nouveau signé avant de pouvoir être utilisé. S'il a été falsifié, il doit être supprimé car sa validité et son authenticité ne sont plus garanties.

Distribuer du code signé

Lorsqu'un script ou un fichier de configuration PowerShell signé est distribué, l'utilisateur doit déterminer s'il peut faire confiance au code d'un éditeur particulier. La première étape consiste à valider l'identité de l'éditeur en fonction d'une chaîne de confiance. Pour établir celle-ci, l'utilisateur se sert du certificat de signature de code de l'éditeur associé à la signature numérique et vérifie que le propriétaire du certificat est l'éditeur. Par exemple, la Figure 4.5 présente un chemin (ou chaîne) de certification non interrompu et valide, à partir du certificat de l'éditeur vers une autorité de certification racine approuvée.Lorsqu'une autorité de certification racine publique ou interne approuvée constitue l'autorité principale de confiance du certificat de l'éditeur, l'utilisateur fait explicitement confiance à l'identité de l'éditeur.

Si une autorité de certification racine est approuvée, le certificat de cette autorité se trouve dans le magasin Autorités de certification racines de confiance (voir Figure 4.6).

Lorsqu'une autorité de certification racine n'est pas une autorité principale de confiance ou lorsque le certificat est auto-signé, l'utilisateur doit décider s'il peut faire confiance à l'identité annoncée par l'éditeur. Si c'est le cas, le certificat de l'autorité de certification racine ou le certificat auto-signé doit être ajouté au magasin Autorités de certification racines de confiance pour établir une chaîne de confiance valide.

Après vérification ou approbation de l'identité de l'éditeur, l'étape suivante consiste à décider si l'exécution du code signé est sûre. Lorsqu'un utilisateur a déjà validé la fiabilité de l'exécution du code publié par l'éditeur, le code (un script ou un fichier de configuration PowerShell) s'exécute sans autre intervention.

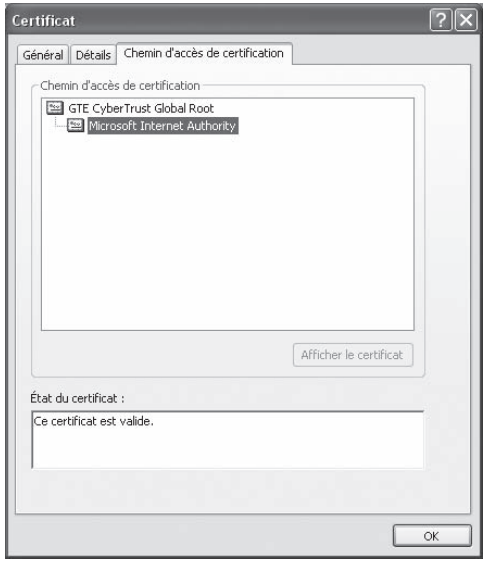


Figure 4.5
Le chemin de certification.

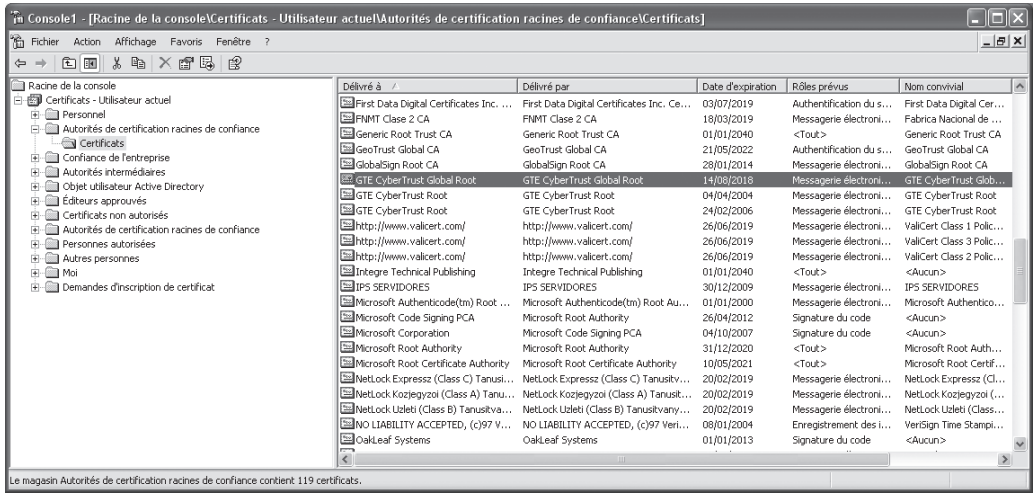


Figure 4.6
Magasin de certificats Autorités de certification racines de confiance.

Si un éditeur est approuvé, son certificat de signature du code réside dans le magasin Éditeurs approuvés (voir Figure 4.7).

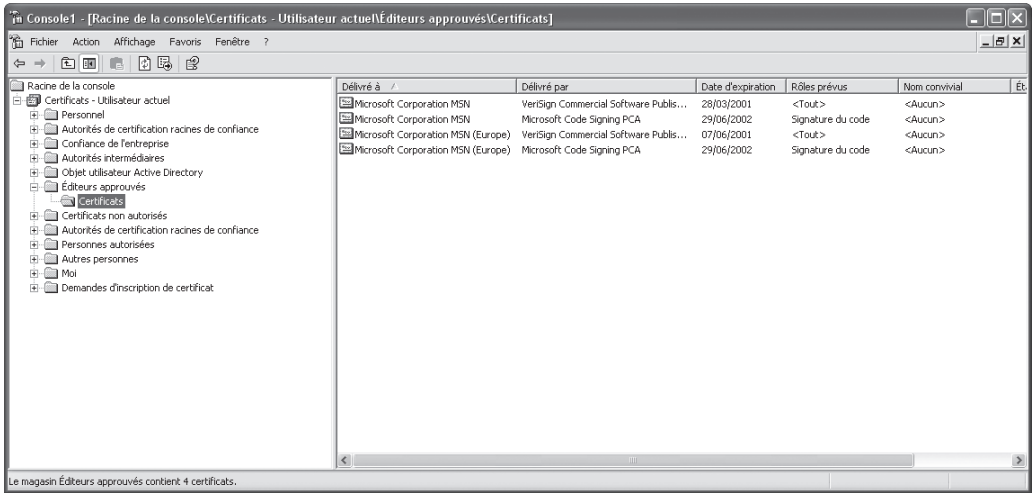


Figure 4.7
Magasin de certificats Éditeurs approuvés.

Si un éditeur n'est pas approuvé, PowerShell demande à l'utilisateur de choisir si le code signé peut être exécuté :

```
PS C:\Scripts> .\ScriptSigne.ps1

Voulez-vous exécuter le logiciel de cet éditeur non approuvé ?
Le fichier C:\Scripts\ScriptSigne.ps1 est publié par CN=companyabc.com, OU=IT,
O=companyabc.com, L=Oakland, S=California, C=US et n'est pas approuvé sur
votre système. N'exécutez que des scripts provenant d'éditeurs approuvés.
[M] Ne jamais exécuter  [N] Ne pas exécuter  [O] Exécuter une fois
[T] Toujours exécuter[?] Aide (la valeur par défaut est « N ») :
```

Voici la signification des options disponibles :

- *[M] Ne jamais exécuter*. Cette option place le certificat de l'éditeur dans le magasin Certificats non autorisés. Lorsque le certificat d'un éditeur a été déclaré non digne de confiance, PowerShell interdit à tout jamais l'exécution du code provenant de cet éditeur, sauf si le certificat est retiré du magasin Certificats non autorisés ou si la stratégie d'exécution est fixée à `Unrestricted` ou à `RemoteSigned`.
- *[N] Ne pas exécuter*. Cette option, par défaut, interrompt l'exécution du code non approuvé.

- *[O] Exécuter une fois*. Cette option autorise une seule exécution du code non approuvé.
- *[T] Toujours exécuter*. Cette option place le certificat de l'éditeur dans le magasin Éditeurs approuvés. Par ailleurs, le certificat de l'autorité de certification racine est placé dans le magasin Autorités de certification racines de confiance, s'il ne s'y trouve pas déjà.

Distribuer du code dans l'entreprise

Vous vous demandez peut-être comment contrôler le code qui peut être approuvé dans votre entreprise. Si le choix du code approuvé est laissé aux utilisateurs ou aux machines, la distribution du code signé dans un environnement géré est contrariée. Si votre environnement est géré, la mise en œuvre de votre PKI doit offrir des méthodes permettant de maîtriser ce qui est digne de confiance dans une entreprise. Si votre société se trouve dans un environnement Windows, la méthode classique passe par un GPO. Par exemple, vous pouvez définir des éditeurs approuvés en utilisant une liste de certificats de confiance (CTL, *Certificate Trust List*) ou au travers de l'extension Maintenance d'Internet Explorer.

Distribuer du code dans le domaine public

Déterminer la confiance dans le domaine public est totalement différent. Lorsqu'on établit une confiance entre deux entités privées, elles sont en mesure de définir ce qui est digne de confiance ou non. Lorsque des entités publiques interviennent, ce niveau de contrôle n'existe pas. C'est à ces entités publiques de déterminer ce à quoi elles accordent leur confiance.

En résumé

Ce chapitre a présenté en détail la signature du code. Avec ce que vous avez appris, vous devez à présent comprendre l'importance de la signature du code dans la sécurité de PowerShell et comment vous en servir. Si vous n'avez pas encore assimilé cet impératif, nous répétons que la signature du code doit être comprise et utilisée dans le développement de vos scripts.

Vous devez également mieux comprendre l'infrastructure requise pour que la signature du code constitue une méthode viable pour accorder sa confiance à du code dans une entreprise. Même si la PKI n'est pas simple à maîtriser, l'un des premiers objectifs de ce chapitre était de la présenter du point de vue de l'écriture des scripts – cette option a été choisie afin que vous ne soyez pas trop dérouté et qu'elle soit associée à son utilisation dans PowerShell. Grâce à ces connaissances, vous devez à présent être capable de déterminer, ou tout au moins d'indiquer, un besoin de PKI et d'orienter un projet de manière que les scripts que vous développez puissent être approuvés dans votre entreprise.

5

Suivre les bonnes pratiques

Dans ce chapitre

- Introduction
- Développer des scripts
- Concevoir des scripts
- Sécuriser des scripts
- Utiliser les standards d'écriture

Introduction

Il existe de nombreux guides pour apprendre les bonnes pratiques de développement des scripts. Le plus souvent, ils traitent d'un langage spécifique, de concepts généraux ou même des préférences d'un développeur. Quel que soit le type de guide consulté, l'objectif reste toujours l'amélioration de la manière d'écrire des scripts.

Ce chapitre propose quelques méthodes de programmation de scripts fondées sur une expérience de développement de logiciels. Le développement de scripts est analogue à celui de logiciels car ces deux activités impliquent l'écriture de code de manière sensée. Par ailleurs, de nombreux aspects d'un projet de développement logiciel s'appliquent également aux projets de scripts. Les bonnes pratiques du développement de logiciels constituent de bonnes bases à l'amélioration des scripts.

Développer des scripts

Les sections suivantes décrivent les pratiques à suivre pour le développement général de scripts. Nous vous conseillons fortement de les suivre, tout au moins en partie, lors de vos développements. Ainsi, vous constaterez que vos scripts commencent à satisfaire aux exigences d'un projet, demandent moins de temps de développement et présentent moins de problèmes de déploiement.

Considérer le développement de scripts comme de véritables projets

La mise en œuvre d'un script peut demander autant d'efforts que n'importe quel projet de développement logiciel. Par exemple, il ne faut pas oublier les phases de prototypage et de test afin d'éviter un impact négatif sur l'environnement. Par conséquent, lors de l'écriture d'un script, la portée de ses effets doit être vérifiée. S'il est complexe, si son temps d'exécution est supérieur à quelques minutes, s'il implique d'autres ressources que son développeur (comme d'autres personnes) ou si son exécution montre un niveau de risque élevé, il est préférable de transformer l'écriture du script en un réel projet de développement.

Mettre en place un cycle de développement

Comme pour tout projet logiciel, un modèle du cycle de développement correspondant aux besoins du script doit être mis en place. Les modèles vont du traditionnel modèle en cascade aux plus récents, comme les méthodes agiles, l'Extreme Programming (XP), le cycle en spirale, le cycle itératif, etc. Cependant, le choix n'est pas aussi important que la définition d'un processus formel de gestion des projets de scripts.

Si les modèles mentionnés semblent trop complexes pour un projet de scripts, la Figure 5.1 présente une suite d'étapes simples conçues pour le développement de scripts.

Bien qu'elles soient comparables à un modèle intégral de cycle de développement, ces étapes ne sont que des références vers des tâches qui doivent être accomplies. Vous pouvez suivre ce modèle ou développer le vôtre, mais l'important est d'établir une méthode de gestion des projets de scripts.

Concevoir et prototyper les scripts avec du pseudo-code

La conception et le prototypage d'un script avec pseudo-code ont pour objectif de développer la structure et la logique du script avant d'écrire ne serait-ce qu'une seule ligne de code. De cette manière, il est plus facile de s'assurer que le script répond aux exigences et de détecter les problèmes de logique dès le début du processus de développement.

Par ailleurs, le pseudo-code permet de s'affranchir du langage et peut être écrit de manière que d'autres personnes, en particulier celles qui doivent commenter la conception de scripts, puissent le lire et le comprendre facilement. En voici un exemple :

```
Param domaine
Param fichier CSV de comptes de ressources

Se lier au domaine
Ouvrir et lire le fichier CSV

Pour chaque compte de ressource dans le fichier CSV:
    - créer un nouveau compte dans l'UO indiquée ;
    - fixer le mot de passe (14 caractères générés aléatoirement) ;
    - journaliser le mot de passe dans l'archive des mots de passe ;
    - fixer les attributs du compte d'utilisateurs en fonction des
      informations données par le fichier CSV ;
    - activer le courrier électronique sur le compte ;
    - ajouter l'utilisateur au groupe adéquat en fonction des informations
      données par le fichier CSV.
Suivant
```

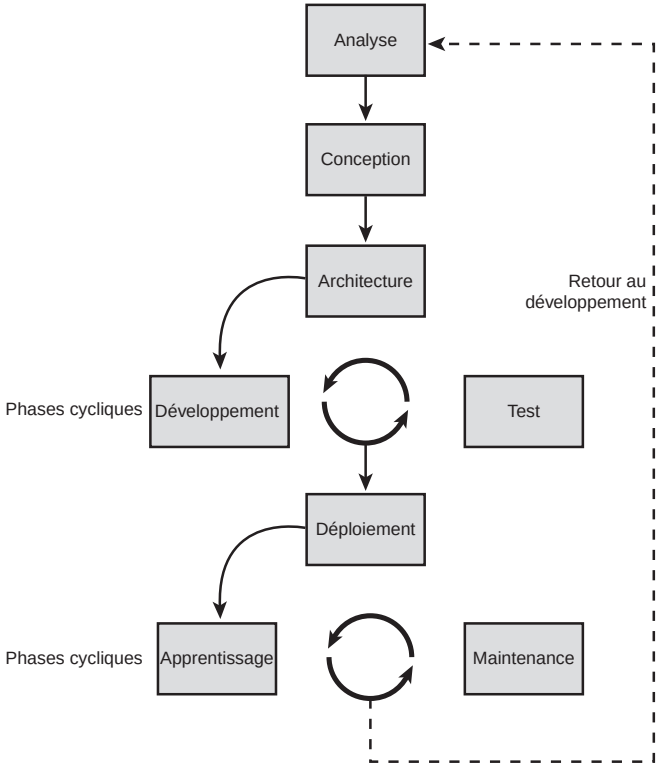


Figure 5.1
Les étapes d'un projet de script.

Rassembler efficacement les contraintes du script

Comme pour n'importe quel projet, il faut définir les problèmes résolus par le script afin de déterminer ce qu'il doit réaliser. Parfois, un script doit juste satisfaire un simple besoin d'automation et ses contraintes sont alors faciles à cerner. Lorsqu'il doit résoudre des problèmes d'automation plus complexes, il peut être nécessaire d'en savoir plus sur les processus métiers à automatiser pour pouvoir déterminer les contraintes. Dans tous les cas, identifier les exigences d'un script et demander à tous les participants de les approuver constitue un élément essentiel du succès du projet. Si ces étapes du processus de développement sont sous-estimées, le script final risque de ne pas répondre aux besoins et de ne pas être retenu comme une solution au problème initial.

Ne pas développer des scripts dans un environnement de production

La plupart des scripts apportent des changements au système. Il est donc possible que l'exécution d'un script dans un environnement de production occasionne des dégâts inattendus. Même si le script n'effectue aucune modification, il peut avoir des effets indésirables ou ses implications peuvent ne pas être totalement maîtrisées. Pire encore, lorsqu'il est exécuté afin de vérifier son fonctionnement, s'il n'est pas lancé dans l'environnement de test défini, il peut affecter des systèmes en production. Par conséquent, les scripts ne doivent jamais être développés dans un environnement de production.

Tester, tester et tester

Les scripts ont généralement pour objectif une forme d'automation, comme modifier un attribut pour chaque utilisateur d'un domaine Active Directory. La tâche d'automation peut avoir un impact élevé ou faible, mais il est indispensable de réaliser un test de qualité du code avant de le placer dans un environnement de production. Les scripts doivent être testés consciencieusement car ils peuvent avoir un effet sur l'environnement.

Écrire des scripts professionnels

De nombreux développeurs ont tendance à considérer les scripts comme une solution rapide et facile pour effectuer des tâches et ne voient pas la nécessité d'une méthodologie professionnelle impliquant planification, documentation et norme. Cette mentalité vient de l'époque où l'écriture de scripts était considérée comme une tâche réservée aux programmeurs UNIX et Linux. Cette vision est en train de changer, avec la sortie de PowerShell. La ligne de commande, les scripts et l'automation deviennent les bases sur lesquelles les administrateurs de systèmes

Windows fondent la gestion de leurs environnements. Grâce à cette évolution, la souplesse et la puissance brute des scripts sont de plus en plus considérées comme une solution aux besoins d'automation et leur développement doit donc se faire avec professionnalisme.

Pour créer des scripts professionnels, le travail accompli doit satisfaire un certain niveau de qualité. Cela implique de développer des standards pour tous les scripts, d'écrire une documentation claire et concise, de respecter les bonnes pratiques en termes de planification et d'organisation, d'effectuer des tests consciencieux, etc. Une adhésion aux standards professionnels permet également de s'assurer que les autres intervenants du projet seront bien disposés quant à l'acceptation du travail.

Concevoir des scripts

Les sections suivantes proposent de bonnes pratiques pour la conception de scripts PowerShell. Le terme "conception" est employé de manière assez souple car l'objectif est d'entrer dans les aspects de la conception qui doivent ou ne doivent pas être retenus lors de l'écriture d'un script PowerShell. Par exemple, les informations fournies au script doivent être validées. Une fois encore, nous vous recommandons fortement d'appliquer, sous une forme ou sous une autre, les pratiques données dans ces sections. En les respectant, les scripts seront plus lisibles, plus utilisables, plus robustes et moins bogués.

Ajouter des informations de configuration au début du script

Les variables ou les paramètres qui définissent la configuration du script doivent toujours être placés au début. Ainsi, toute personne qui utilise, lit ou modifie le script pourra les retrouver très simplement. Par exemple :

```
#--------------------------
# Définir les variables.
#--------------------------
$Proprietaire = "Administrateurs"
$Cibles = import-csv $FichierImport

#--------------------------
# Corps du script.
#--------------------------
...
```

Cette pratique permet également de réduire les erreurs dues à la modification de la configuration du script. Si les informations de configuration sont réparties dans tout le script, il risque d'être mal configuré, de contenir des doublons ou des oublis.

Utiliser les commentaires

Rien ne garantit que les utilisateurs comprendront immédiatement la logique du script ou seront familiers des méthodes employées pour réaliser certaines tâches. Par conséquent, des commentaires doivent les aider à comprendre le fonctionnement du script, mais, en aucun cas, ils ne doivent ressembler à un roman. En revanche, ils doivent apporter suffisamment d'informations pour que la logique du script apparaisse clairement. Par ailleurs, si le script inclut une méthode, une classe ou une fonction complexe, un commentaire doit l'expliquer. Les commentaires ont également pour avantage de faciliter le retour sur un script ou sa mise à jour. L'exemple suivant montre comment des commentaires apportent des informations utiles :

```
#-----------------------------------------------
# Add-DACL
#-----------------------------------------------
# Usage :        Accorder des droits à un dossier ou un fichier.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").
# $Identity :    Nom d'utilisateur ou de groupe ("Administrateurs" ou
#                "monDomaine\utilisateur1").
# $AccessMask :  Les droits à utiliser pour la création de la règle d'accès
#                ("FullControl", "ReadAndExecute, Write", etc.).
# $Type :        Accorder ou refuser les droits ("Allow" ou "Deny").
```

Éviter de figer des informations de configuration

Figer des informations de configuration dans le code est une erreur classique. Les informations requises ne sont pas demandées aux utilisateurs, elles sont figées dans des variables ou réparties aléatoirement dans le script. Cette méthode impose aux utilisateurs d'intervenir manuellement dans les scripts pour en modifier la configuration. Cela augmente donc les risques d'erreurs et des problèmes d'exécution. N'oubliez pas que l'un des objectifs est de fournir des scripts utilisables. Les informations figées dans le code ne permettent pas d'utiliser facilement un script dans des environnements différents. Nous vous conseillons d'utiliser

des paramètres ou des fichiers de configuration, comme le montre l'exemple suivant, afin que les utilisateurs puissent configurer plus aisément le script.

```
param([string] $CheminRechercheADSI=$(throw "Veuillez indiquer le chemin ADSI ! "))
```

Si nécessaire, utiliser des variables

Si les informations de configuration ne doivent pas être figées dans le script, elles doivent cependant être représentées par des variables. Le fait de définir une information dans une variable en un seul endroit, au lieu de la placer en plusieurs endroits du script, réduit les risques d'introduire des erreurs lors de la modification de cette information. Par ailleurs, le regroupement des informations de configuration en un même endroit, en particulier au début du script, diminue également le temps nécessaire à la reconfiguration d'un script pour des environnements différents.

Donner des instructions

La plupart des scripts sont écrits par une personne mais utilisés par une autre, un administrateur, très souvent, qui ne maîtrise pas le code et les interfaces en ligne de commande. Autrement dit, les scripts doivent être aussi utilisables qu'utiles. Si vous ne fournissez aucune instruction pour que même le débutant puisse lancer le script et comprendre son rôle, vous n'êtes pas un véritable développeur de scripts.

Il n'est pas rare de rencontrer des scripts sans explications, avec des instructions incorrectes ou très peu d'explications de son objectif. Cela est généralement frustrant pour les utilisateurs, qui peuvent n'avoir aucune idée de l'impact des scripts sur leur environnement et les exécuter, ce qui peut conduire à des désastres.

L'exemple suivant comprend des instructions à inclure dans un fichier qui documente l'objectif et le fonctionnement du script :

```
=================================================================
Informations sur le script.
=================================================================
Nom : AddProxyAddress.ps1
Auteur : Tyson Kopczynski
Date : 6/02/2006
```

```
Description :
Ce script permet ajouter des adresses de proxy secondaires aux utilisateurs,
conformément au fichier CSV importé. Avant d'ajouter ces adresses, il vérifie
les conditions suivantes :
    L'utilisateur existe-t-il ?
    Dispose-t-il de courrier électronique ?
    L'adresse du proxy existe-t-elle déjà ?

Ce script crée un journal à chacune de ses exécutions.

Format du fichier CSV :
[NomCompteSAM],[AdressesProxy]
tyson,tyson@cco.com;tyson@taosage.net
maiko,maiko@cco.com
bob,bob@cco.com
erica,erica@cco.com

Pour placer plusieurs adresses de proxy dans la colonne AdressesProxy,
utilisez le caractère ; comme séparateur.
```

Valider les paramètres requis

Une erreur classique consiste à ne pas valider les paramètres requis. Si le script demande à l'utilisateur de saisir des informations, l'absence de cette vérification peut l'empêcher de fournir une entrée invalide et le script risque de s'arrêter avec une erreur. Pour les petits scripts, cet oubli ne constituera sans doute pas un problème, mais il risque d'affecter sérieusement la simplicité d'utilisation des longs scripts complexes.

Supposons que le script gère un inventaire de logiciels. Dans un environnement de développement constitué de quelques machines, nous exécutons le script, mais sans fournir l'information correcte pour un paramètre obligatoire. Il s'exécute et échoue quelques secondes plus tard. Nous comprenons notre erreur, la corrigeons et relançons le script.

Ensuite, sur des milliers de machines, l'administrateur système lance le script, lequel s'exécute pendant six heures puis échoue. En consultant les informations d'erreur, l'administrateur découvre que le script s'est arrêté à cause d'un paramètre mal saisi. Il vient de dépenser six heures de son temps uniquement pour rencontrer une erreur. Il risque de conclure que le script n'est pas utilisable. Autrement dit, le script fonctionne dans notre environnement mais pas dans celui de l'administrateur.

Pour éviter ce problème, tous les paramètres requis doivent être validés, comme le montre l'exemple suivant :

```
param([string] $CheminModele = $(throw write-host `
    "Veuillez préciser le chemin du modèle source de la structure de dossiers" `
    "à copier." -Foregroundcolor Red), [string] $FichierImport = $(throw `
     write-host "Veuillez préciser le nom du fichier CSV à importer." `
    -Foregroundcolor Red))

write-host "Vérification du chemin du modèle" -NoNewLine

if (!(test-path $CheminModele)){
    throw write-host `t "$CheminModele n'est pas un dossier valide !" `
        -Foregroundcolor Red
    }
else {
    write-host `t "[OK]" -Foregroundcolor Green
    }

write-host "Vérification du fichier d'importation" -NoNewLine

if (!(test-path $FichierImport)){
throw write-host `t "$FichierImport n'est pas un fichier valide !" `
    -Foregroundcolor Red
    }
else {
    write-host `t "[OK]" -Foregroundcolor Green
    }
```

Écrire des scripts et des fonctions réutilisables

Si vous avez passé du temps à développer une fonctionnalité sophistiquée, vous devez aussi la rendre réutilisable. Un jeu de scripts ou de fonctions commun permet de gagner du temps lors de l'écriture de nouveaux scripts. Supposons, par exemple, que nous ayons écrit dans un script une logique d'analyse du contenu d'un fichier CSV afin de créer un tableau HTML. Au lieu de copier et de modifier cette logique dans de nouveaux scripts, il est préférable de créer un script ou un fichier de bibliothèque qui la met en œuvre de manière à la réutiliser dans un autre script.

La réutilisabilité est une bonne pratique importante. Dans PowerShell, le concept de réutilisabilité prend tout son sens car les fichiers de scripts et de bibliothèque sont faciles à importer, en invoquant du code réutilisable depuis une console PowerShell ou en chargeant le fichier de script ou de bibliothèque à l'aide de l'instruction point. L'exemple suivant montre une suite de fichiers de scripts appelés depuis la console PowerShell au sein d'un pipeline.

```
PS C:\> .\get-utilisateursinvalides.ps1 mondomaine.fr | .\sortie-html.ps1 |
.\sortie-ie.ps1
```

Choisir des noms descriptifs à la place des alias

L'utilisation des alias dans PowerShell permet de gagner du temps, mais les scripts sont alors plus difficiles à lire. Le langage de PowerShell est conçu pour rester simple à écrire et à lire, mais les conventions de nommage de chacun et l'emploi des alias ont un effet sur la lisibilité. Pour garantir la bonne lisibilité du code, il est préférable de suivre des conventions de nommage cohérentes et de remplacer les alias par des noms descriptifs.

En produisant du code lisible, les utilisateurs comprendront plus facilement le script et les mises à jour et les modifications futures seront simplifiées. Si vous respectez des conventions de nommage cohérentes et évitez des alias, la modification du script doit être un jeu d'enfant.

Afficher des informations d'état

Dans un script d'automation, il est important d'afficher des informations d'état afin que les utilisateurs suivent la progression du script et sachent quelles tâches ont été accomplies avec succès. Ces informations permettent également aux utilisateurs de connaître les erreurs qui se sont produites et peuvent même indiquer une estimation de l'heure de terminaison du script.

Pour fournir ces informations aux utilisateurs, nous pouvons nous servir de la console, comme le montre la Figure 5.2, avec les applets de commande `Write-Host` et `Write-Progress`, les écrire dans un journal ou utiliser les classes Windows Forms.

INFO

Quelle que soit la méthode employée, l'idée est de fournir suffisamment d'informations d'état sans noyer les utilisateurs sous des détails inutiles. S'il faut différents niveaux de détails des informations, utilisez les applets de commande `Write-Verbose` et `Write-Debug`, les paramètres `Verbose` et `Debug` ou un affichage personnalisé.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> C:\Scripts\ProvisionWebFolders.ps1 \template .\users.csv
----------------------------------------
-         ProvisionWebFolders          -
----------------------------------------

Vérification du chemin des modèles       [OK]
Vérification du fichier importé  [OK]

Création des dossiers web :
D:\Work\WWWs\Tyson
 Dossier [COPIE]
 SetOwner pour Administrateurs [OK]
 AddACE pour Administrateurs [OK]
 ClearInherit [OK]
 AddACE pour SYSTEM [OK]
 AddACE pour Tyson [OK]

D:\Work\WWWs\Maiko
 Dossier [COPIE]
 SetOwner pour Administrateurs [OK]
 AddACE pour Administrateurs [OK]
 ClearInherit [OK]
 AddACE pour SYSTEM [OK]
 AddACE pour Maiko [OK]

D:\Work\WWWs\Garett
 Dossier [COPIE]
 SetOwner pour Administrateurs [OK]
 AddACE pour Administrateurs [OK]
 ClearInherit [OK]
 AddACE pour SYSTEM [OK]
 AddACE pour Garett [OK]

D:\Work\WWWs\Fujio
 Dossier [COPIE]
 SetOwner pour Administrateurs [OK]
 AddACE pour Administrateurs [OK]
 ClearInherit [OK]
 AddACE pour SYSTEM [OK]
 AddACE pour Fujio [OK]

D:\Work\WWWs\Kiyomi
 Dossier [COPIE]
 SetOwner pour Administrateurs [OK]
 AddACE pour Administrateurs [OK]
 ClearInherit [OK]
 AddACE pour SYSTEM [OK]
 AddACE pour Kiyomi [OK]

Terminé : les dossiers web ont été créés
PS C:\Scripts>
```

Figure 5.2

Exemple d'affichage d'informations d'état.

Utiliser les paramètres WhatIf et Confirm

Comme nous l'avons expliqué au Chapitre 2, "Les fondamentaux de PowerShell", deux paramètres d'applet de commande permettent aux auteurs de scripts et aux administrateurs système d'éviter des modifications non voulues. Le paramètre `WhatIf` retourne des informations sur les modifications qui seront effectuées par l'applet de commande, mais sans les appliquer réellement :

```
PS C:\> get-process expl* | stop-process -WhatIf
WhatIf: Opération « Stop-Process » en cours sur la cible « explorer (2172) ».
```

Dans cet exemple, l'objet de processus retourné par `Get-Process` correspond à `explorer.exe`. Normalement, si un objet de processus est envoyé à l'applet de commande `Stop-Process`, le processus correspondant est arrêté. Cependant, lorsque le paramètre `WhatIf` est passé à l'applet `Stop-Process`, la commande retourne des informations sur les changements qui se seraient produits si elle avait été réellement exécutée. Par exemple, supposons que nous saisissions la commande suivante :

ATTENTION

N'exécutez pas la commande suivante. Il s'agit d'un exemple qu'il ne faut pas suivre.

```
PS C:\> get-process | stop-process
```

Sans le paramètre `WhatIf`, cette commande arrêterait la console PowerShell, ainsi que le système. Parce que `WhatIf` a été ajouté, les informations affichées permettent de comprendre que la commande va conduire au dysfonctionnement du système :

```
PS C:\> get-process | stop-process -WhatIf
WhatIf: Opération « Stop-Process » en cours sur la cible « alg (1048) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « ati2evxx (1400) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « ati2evxx (1696) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « atiptaxx (3644) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « BTSTAC~1 (2812) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « BTTray (3556) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « btwdins (1652) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « csrss (1116) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « ctfmon (1992) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « eabservr (3740) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « explorer (2172) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « googletalk (1888) ».
WhatIf: Opération « Stop-Process » en cours sur la cible « GoogleToolbarNotifie
(2236) ».
...
```

Le paramètre `Confirm` évite les modifications inattendues en obligeant PowerShell à consulter l'utilisateur avant d'effectuer tout changement :

```
PS C:\> get-process expl* | stop-process -confirm

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « Stop-Process » en cours sur la cible « explorer (2172) ».
[O] Oui  [T] Oui pour tout  [N] Non  [U] Non pour tout  [S] Suspendre
[?] Aide(la valeur par défaut est « O ») :
```

Les bonnes pratiques recommandent d'utiliser les paramètres `WhatIf` et `Confirm` dès que possible afin d'identifier les changements potentiellement dangereux et de donner aux utilisateurs la possibilité d'interrompre la commande.

INFO

Les paramètres `WhatIf` et `Confirm` ne sont reconnus que par les applets de commande qui effectuent des modifications.

Sécuriser des scripts

La sécurité reste souvent un élément qui n'est pas pris en compte pendant le développement d'un logiciel. C'est aussi le cas lors de l'écriture de scripts. Cependant, la prise en charge de la sécurité dans les scripts fait partie de la liste des bonnes pratiques. C'est pourquoi les trois sections à venir, qui traitent de la sécurité des scripts PowerShell, sont peut-être les plus importantes de ce chapitre.

Signer numériquement les scripts et les fichiers de configuration

Comme nous l'avons souligné au Chapitre 4, "Signer du code", les scripts et les fichiers de configuration PowerShell doivent toujours être signés numériquement pour que les utilisateurs et les machines cibles connaissent l'origine du code et sachent qu'il n'est pas corrompu. En adhérant à cette pratique, vous pouvez conserver une stratégie d'exécution `AllSigned` sur toutes les machines de votre entreprise, y compris la vôtre.

INFO

La signature du code ne concerne pas exclusivement les scripts et les fichiers de configuration PowerShell. Les mêmes principes s'appliquent à d'autres éléments, comme les fichiers exécutables, les macros, les DLL, d'autres scripts, les pilotes de périphériques, les images de micrologiciel et ainsi de suite. Tout code peut bénéficier de la sécurité des signatures numériques et vous pouvez ainsi limiter les possibilités d'exécution d'un code illégitime dans votre environnement.

Ne jamais fixer les stratégies d'exécution à Unrestricted

Fixer la stratégie d'exécution à `Unrestricted` revient à accepter l'exécution d'un code malveillant sur le système. Étant donné ce risque, la stratégie d'exécution doit au moins être fixée à `RemoteSigned`. Cette configuration autorise l'exécution des scripts et le chargement des fichiers de configuration créés localement sur la machine, mais empêche celle de tout code distant non signé et non approuvé. Cependant, elle n'est pas à toute épreuve et pourrait autoriser PowerShell à exécuter certains codes distants.

Le respect des conseils et la maîtrise du processus de signature du code sont essentiels à la protection de l'environnement PowerShell. Choisir la stratégie d'exécution `AllSigned` accroît la sécurité car tous les scripts et tous les fichiers de configuration doivent alors être signés par une source de confiance avant de pouvoir être exécutés ou chargés.

Exécuter les scripts avec les droits minimaux

Les bonnes pratiques de sécurité incluent le principe des privilèges moindres. Ils garantissent que les entités, comme les utilisateurs, les processus et les logiciels, ne reçoivent que les droits minimaux nécessaires pour effectuer une opération légitime. Par exemple, si un utilisateur n'a pas besoin des droits d'administration pour lancer un logiciel de traitement de texte, il n'y a aucune raison de les lui accorder.

Le principe des privilèges moindres s'applique également aux scripts. Lors du développement d'un script, il faut essayer d'écrire le code de manière que son exécution exige des droits minimaux. Par ailleurs, il est indispensable d'expliquer les droits requis afin que les utilisateurs agissent en connaissance de cause. Sinon, ils risquent de lancer l'exécution d'un script avec des droits d'administration, ce qui augmente les possibilités de dommages infligés à l'environnement.

Utiliser les standards d'écriture

Comme pour le développement de logiciels, l'écriture des scripts doit inclure une forme de standardisation. Le terme "standardisation" ne signifie pas ici une norme, comme celles rédigées par l'ISO (*International Organization for Standardization*) ou l'IEEE (*Institute of Electrical and Electronics Engineers*). Nous faisons référence à l'usage de méthodes cohérentes pour le nom, l'organisation et la structure des scripts, pour leur fonctionnement et pour leur traitement des erreurs. En standardisant ces aspects, nous assurons une certaine cohérence dans l'interaction avec nos scripts, leur dépannage et leur utilisation.

La lisibilité d'un script peut être améliorée si vous utilisez des conventions de nommage cohérentes entre les scripts et au sein même des scripts. Répondant à une organisation standard, ils deviennent plus faciles à lire, à dépanner ou à modifier. Cette standardisation permet également de réduire le temps de développement de nouveaux scripts. Par exemple, nous pouvons créer des structures standard pour le traitement des erreurs, la création de journaux et la mise en forme des sorties, puis les réutiliser très simplement.

Standards d'écriture employés dans ce livre

Les chapitres suivants de cet ouvrage se concentrent sur des exemples de scripts PowerShell. Ces scripts opérationnels sont tirés de projets réels répondant à des besoins d'entreprises. Nous présenterons leur code source au cours des chapitres à venir, mais il est également disponible sur la page Web consacrée à cet ouvrage à l'adresse **http://www.pearsoneducation.fr**. L'archive proposée contient les scripts de chaque chapitre, répartis dans des sous-dossiers[2].

Pour résoudre quelques problèmes de standardisation, nous avons fait des choix de présentation des scripts dans cet ouvrage. Premièrement, ils se limitent aux langages PowerShell et VBScript, afin de réduire la complexité inhérente à l'utilisation de différents langages. Deuxièmement, les scripts VBScript sont placés dans un fichier WSF (*Windows Scripting File*). Troisièmement, chacun des scripts PowerShell et VBScript respecte une structure commune facile à comprendre. Les Figures 5.3 et 5.4 présentent des exemples de structures employées dans ce livre.

2. N.d.T : Les commentaires et les messages affichés par les scripts étudiés dans ce livre ont été traduits afin d'en faciliter la compréhension. Cependant, le code source disponible dans l'archive téléchargeable est resté en anglais. En effet, les scripts sont signés et il n'est donc pas possible d'en modifier le contenu sans qu'ils soient considérés comme corrompus, ce qui est parfaitement cohérent avec l'objectif des signatures numériques.

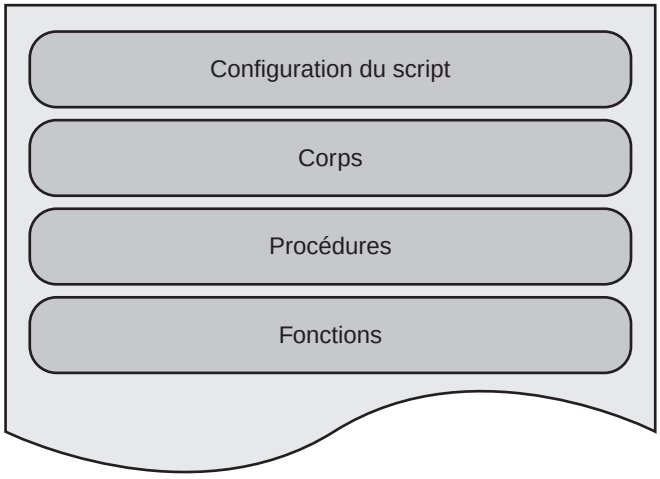


Figure 5.3
Organisation d'un script WSF.

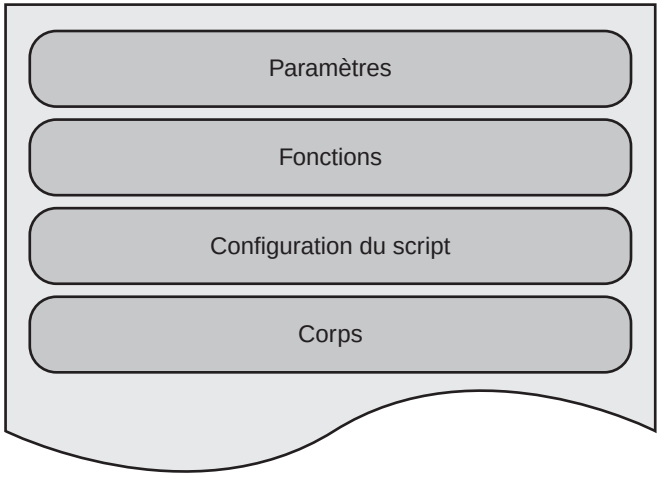


Figure 5.4
Organisation d'un script PowerShell.

Quatrièmement, un certificat de signature numérique du code a été acheté auprès de Thawte et tous les scripts PowerShell ont été signés par l'entité companyabc.com. Si vous avez respecté les bonnes pratiques concernant la stratégie d'exécution, vous devez définir companyabc.com comme un éditeur approuvé avant d'exécuter les scripts PowerShell.

ATTENTION

Les scripts fournis avec ce livre sont opérationnels. Ils ont été testés et doivent se comporter comme nous l'expliquons. Cependant, cela ne signifie pas qu'ils peuvent être utilisés dans un environnement de production. Si vous envisagez d'exécuter l'un de ces scripts dans un tel environnement, vous devez commencer par le tester.

Enfin, les scripts PowerShell et VBScript ont tendance à offrir le même type d'interaction quant aux entrées et aux sorties, bien qu'il existe certaines différences pour les concepts nouveaux. Cependant, les méthodes d'entrée et de sortie sont claires et concises, grâce à l'utilisation de la console PowerShell, des fichiers de journalisation et des classes Windows Forms.

En résumé

Au fil de ce chapitre, nous avons vu un certain nombre de bonnes pratiques pour l'écriture des scripts PowerShell. Elles concernent leur développement, leur conception et leur sécurisation. L'objectif est également d'améliorer vos qualités de programmeur de scripts. L'origine de ces pratiques se trouve autant dans le développement de logiciels que dans une expérience pratique du développement de scripts. Leur mise en œuvre n'est en aucun cas figée.

Le véritable objectif de ce chapitre était de vous inviter à réfléchir sur la mise en place d'un bon processus d'écriture des scripts. Vous choisirez peut-être d'étendre ces pratiques lorsque vous les inclurez dans votre prochain développement de scripts PowerShell. L'équipe PowerShell a fait beaucoup d'efforts pour tenter de produire le shell parfait et vous pouvez la remercier en tentant d'écrire des scripts bien pensés, bien conçus et sûrs.

II

Appliquer ses connaissances à PowerShell

6

PowerShell et le système de fichiers

Dans ce chapitre

- Introduction
- Gérer le système de fichiers depuis WSH et PowerShell
- Manipuler les autorisations
- De VBScript à PowerShell

Introduction

Ce chapitre s'intéresse à la gestion du système de fichiers de Windows depuis PowerShell. Pour cela, il présente des exemples détaillés fondés sur WSH (*Windows Script Host*) et sur PowerShell. Ils sont donnés sous ces deux formes afin que le lecteur puisse passer à PowerShell en bénéficiant de son expérience de l'écriture de scripts pour Windows. Outre la comparaison des exemples, ce chapitre décrit également un script opérationnel pour la gestion des fichiers dans un cas réel. L'objectif est de donner au lecteur la possibilité d'appliquer les techniques de scripts PowerShell à des besoins d'automation réels.

Gérer le système de fichiers depuis WSH et PowerShell

WSH propose plusieurs méthodes de manipulation du système de fichiers de Windows. L'objet `FileSystemObject` (FSO), WMI (*Windows Management Instrumentation*) et certains outils, comme `copy`, `calcs` et `xcalcs`, n'en sont que quelques exemples. Grâce à ces outils pléthoriques, nous pouvons effectuer des tâches comme copier, créer et supprimer des fichiers et des dossiers. La plupart des créateurs de scripts se servent du modèle FSO pour manipuler le système de fichiers.

FSO fait partie du modèle d'objet de WSH. Il joue le rôle de racine d'une hiérarchie d'objets, de méthodes et de collections COM qui donnent accès au système de fichiers. Il permet généralement aux programmeurs de manipuler le système de fichiers comme bon leur semble, mais, dans certains cas, ses fonctionnalités ne sont pas suffisantes et des outils et des méthodes complémentaires sont nécessaires à certaines tâches.

D'un autre côté, PowerShell dispose d'un fournisseur intégré, appelé FileSystem, pour interagir avec le système de fichiers de Windows. Ce fournisseur apporte une couche d'abstraction entre PowerShell et le système de fichiers de Windows qui fait que ce dernier ressemble à un magasin de données hiérarchique. Par conséquent, l'accès au système de fichiers se fait exactement comme pour n'importe quel autre magasin de données disponible par le biais d'un fournisseur PowerShell. Comme nous l'avons expliqué au Chapitre 3, "Présentation avancée de PowerShell", les applets de commande principales permettant d'accéder aux magasins de données et de les manipuler servent également pour le système de fichiers. La commande suivante affiche la liste des applets de commande principales qui manipulent les magasins de données accessibles par le biais des fournisseurs PowerShell :

```
PS C:\> help about_core_commands
```

Manipuler les lecteurs

Dans WSH, nous pouvons utiliser l'objet FSO `Drive` pour obtenir des informations concernant les lecteurs disponibles sur un système :

```
Dim FSO, objDrive
Set FSO = CreateObject("Scripting.FileSystemObject")
Set objDrive = fso.GetDrive(fso.GetDriveName("C:\"))

WScript.Echo "Espace total : " & FormatNumber(objDrive.TotalSize / 1024, 0)
```

Dans PowerShell, nous pouvons obtenir des informations sur un lecteur à l'aide des applets de commande `Get-PSDrive` et `Get-Item`. Cependant, comme nous l'avons expliqué au Chapitre 3, PowerShell traite les lecteurs différemment de WSH. Par conséquent, si nous voulons les mêmes informations que celles fournies par l'objet FSO `Drive`, nous devons utiliser la classe .NET appropriée, comme le montre l'exemple suivant, ou WMI :

```
PS C:\> $LecteurC = new-object System.IO.DriveInfo C
PS C:\> $TailleLecteur = ($LecteurC.TotalSize / 1024) / 1000 /1000
PS C:\> $TailleLecteur = "{0:N0}" -f $TailleLecteur
PS C:\> write-host "La taille du lecteur C est égale à $TailleLecteur Go."
La taille du lecteur C est égale à 69 Go.
PS C:\>
```

Manipuler les dossiers

Dans WSH, nous pouvons accéder aux informations d'un dossier et créer, supprimer, copier et déplacer des dossiers à l'aide de l'objet FSO `Folder` :

```
Dim FSO, objFolder
Set FSO = CreateObject("Scripting.FileSystemObject")
Set objFolder = FSO.GetFolder("C:\outils")

WScript.Echo objFolder.DateLastAccessed
```

Dans PowerShell, les applets de commande principales réalisent les mêmes tâches :

```
PS C:\> get-item C:\outils | select LastAccessTime

LastAccessTime
--------------
06/10/2007 14:03:20


PS C:\>
```

Manipuler les fichiers

Dans WSH, nous pouvons accéder aux informations d'un fichier et créer, modifier, copier, déplacer et supprimer des fichiers à l'aide de l'objet FSO `File` :

```
Dim FSO
Set FSO = CreateObject("Scripting.FileSystemObject")
strNomExtension = FSO.GetExtensionName("C:\outils\Plan_Domination_Monde_R1.doc")

WScript.Echo strExtensionName
```

Dans PowerShell, les applets de commande principales permettent d'accéder aux informations d'un fichier et de manipuler des fichiers :

```
PS C:\outils> $Fichier = get-item Plan_Domination_Monde_R1.doc
PS C:\outils> $Fichier.extension
.doc
PS C:\outils>
```

Comme le montrent ces exemples, les méthodes de manipulation du système de fichiers Windows avec FSO et PowerShell sont similaires. Les applets de commande principales de PowerShell permettent de réaliser pratiquement les mêmes tâches que les objets FSO.

Manipuler les autorisations

Dans WSH, la manipulation des autorisations du système de fichiers présente quelques limites. Par exemple, il n'existe aucune solution simple pour modifier les autorisations d'un fichier ou d'un dossier. Les développeurs de scripts doivent choisir entre utiliser un outil externe, comme `cacls`, `Xcacls`, `Xcalcs.vbs` ou `SubInACL`, ou passer par `ADsSecurity.dll` ou la classe WMI `Win32_LogicalFileSecuritySetting`. Aucune de ces méthodes n'offre une solution complète ou standard de gestion des autorisations du système de fichiers dans WSH. En général, un script permet de compenser l'absence de fonctionnalités.

Fixer des autorisations avec SubInACL

Étant donné les limites de WSH, l'outil `SubInACL` est très souvent employé pour modifier les autorisations du système de fichiers. Il n'est pas parfait, mais, en le complétant par des scripts, il suffit généralement à cette tâche. Par ailleurs, `SubInACL` prend en charge les fichiers, les

répertoires, les partages de fichiers et les partages d'imprimantes. Il peut également être utilisé sur le Registre, les services système et même la métabase IIS (*Internet Information Services*). Vous pouvez télécharger `SubInACL` à l'adresse **www.microsoft.com/downloads/details.aspx?FamilyId=E8BA3E56-D8FE-4A91-93CF-ED6985E3927B&displaylang=en**.

La syntaxe de `SubInACL` prend la forme `[/Option] /type_objet nom_objet [[/Action[=Paramètre]..]]`. Bien que cette syntaxe semble simple, `SubInACL` est un outil complexe qui permet de répondre à diverses situations.

Quel que soit l'outil utilisé, les changements d'autorisation suivants sont les plus répandus :

- devenir le propriétaire ;
- effacer des autorisations ;
- ajouter des autorisations ;
- retirer des autorisations.

Cette liste n'est en rien exhaustive, mais elle peut servir de base au développement de fonctions très souvent utilisées. L'écriture de fonctions réutilisables est une bonne pratique fortement recommandée. Elles peuvent servir dans de nombreux scripts et réduire les temps de développement. Pour modifier les autorisations du système de fichiers, l'écriture de fonctions réutilisables prend tout son sens car travailler avec les interfaces reconnues par WSH ou les outils existants peut demander beaucoup de temps. Par conséquent, les fonctions `SubInACL` décrites à la section suivante ont été créées de manière à être réutilisées dans des scripts.

Fonctions SubInACL

Nous proposons quatre fonctions `SubInACL` : `SetOwner`, `DumpPerm`, `AddPerm` et `RemovePerm`. Chacune d'elles attend des arguments et construit une chaîne de commande destinée à `SubInACL`. En utilisant un objet `WshShell`, nous exécutons `SubInACL` avec la chaîne de commande préparée. Ensuite, le contenu du fichier `log.temp` géré par `SubInACL` est examiné, à la recherche d'erreurs, grâce à la fonction `ParseTempFile`. Selon les informations d'erreurs trouvées, un code de succès ou d'échec est finalement écrit sur la console.

```
Function SetOwner(path, account)
    'Fixer le propriétaire d'un dossier ou de sous-dossiers.
    On Error Resume Next
    strCommand = "subinacl /verbose /output=log.temp " _
        & "/subdirectories """ & path & """ /setowner=""" & account & """"
```

```
    ErrorCode = objWS.Run(strCommand, 0, TRUE)

    If ErrorCode <> 0 Then
        StdOut.Write("  " & account & ":" _
            & " [Échec de SetOwner] sur " & path)
    Else
        return = inStr(1, ParseTempFile("log.temp"), "ne sera pas examiné")

        If Not return = 0 Then
        StdOut.Write("  " & account & ":" _
            & " [Échec de SetOwner] sur " & path)
        Else
        StdOut.Write("  " & account & ":" _
            & " [Succès de SetOwner] sur " & path)
        End If
    End If

    ErrorCode = vbNullString
End Function

Function DumpPerm(path)
    ' Effacer les autorisations d'un dossier ou de sous-dossiers.
    On Error Resume Next
    strCommand = "subinacl /verbose /output=log.temp " _
        & "/subdirectories """ & path & """ /perm"

    ErrorCode = objWS.Run(strCommand, 0, TRUE)

    If ErrorCode <> 0 Then
        StdOut.Write("  Autorisations non effacées sur " & path)
    Else
        StdOut.Write("  Autorisations effacées sur " & path)
    End If

    ErrorCode = vbNullString
End Function

Function AddPerm(path, account, access)
    ' Accorder des droits à un utilisateur sur un dossier ou des sous-dossiers.
    On Error Resume Next
    strCommand = "subinacl /verbose /output=log.temp" _
        & " /subdirectories """ & path & """ /grant=""" _
        & account & """ =" & access

    ErrorCode = objWS.Run(strCommand, 0, TRUE)
```

```
    If ErrorCode <> 0 Then
        StdOut.Write("  " & account & ": " & access _
            & " [Échec de AddPerm] sur " & path)
    Else
        return = inStr(1, ParseTempFile("log.temp"), "ne sera pas examiné")

        If Not return = 0 Then
            StdOut.Write("  " & account & ": " & access _
                & " [Échec de AddPerm] sur " & path)
        Else
            StdOut.Write("  " & account & ": " & access _
                & " [Succès de AddPerm] sur " & path)
        End If
    End If

    ErrorCode = vbNullString
End Function

Function RemovePerm(path, account, access)
    ' Retirer des droits à l'utilisateur sur un dossier ou des sous-dossiers.
    On Error Resume Next
    strCommand = "subinacl /verbose /output=log.temp" _
        & " /subdirectories """ & path & """ /revoke=""" _
        & account & """ =" & access

    ErrorCode = objWS.Run(strCommand, 0, TRUE)

    If ErrorCode <> 0 Then
        StdOut.Write("  " & account & ": " & access _
            & " [Échec de AddPerm] sur " & path)
    Else
        return = inStr(1, ParseTempFile("log.temp"), "ne sera pas examiné")

        If Not return = 0 Then
            StdOut.Write("  " & account & ": " & access _
                & " [Échec de AddPerm] sur " & path)
        Else
            StdOut.Write("  " & account & ": " & access _
                & " [Succès de AddPerm] sur " & path)
        End If
    End If

    ErrorCode = vbNullString
End Function
```

Fixer des autorisations dans PowerShell

Vous pourriez penser que, grâce aux applets de commandes `Get-ACL` et `Set-ACL`, la gestion des autorisations du système de fichiers est plus facile dans PowerShell. Cependant, `Set-ACL` a besoin d'un objet descripteur de sécurité défini par la classe `System.Security.AccessControl.ObjectSecurity`. La création d'un descripteur de sécurité n'est pas complexe, mais la gestion des autorisations n'est pas aussi évidente à mettre en place dans un script qu'on pourrait le croire. Face à des termes tels que descripteur de sécurité et liste de contrôle d'accès (ACL, *Access Control List*), vous pourriez être tenté de revenir à des outils plus familiers, comme `SubInACL`. Cependant, si vous étudiez le processus étape par étape, vous vous apercevrez qu'il n'est finalement pas très compliqué :

1. Obtenir le descripteur de sécurité d'un objet à l'aide de `Get-ACL`.
2. Créer une ACL à partir d'entrées de contrôle d'accès (ACE, *Access Control Entry*).
3. Ajouter l'ACL au descripteur de sécurité.
4. Associer le nouveau descripteur de sécurité à l'objet à l'aide de `Set-ACL`.

Le code suivant illustre ces quatre étapes :

```
PS C:\> $DS = get-acl "Planning de Helena.csv"
PS C:\> $Regle = new-object System.Security.AccessControl.
FileSystemAccessRule( "maiko", "FullControl", "Allow")
PS C:\> $DS.AddAccessRule($Regle)
PS C:\> set-acl "Planning de Helena.csv" $DS
PS C:\>
```

L'étape la plus difficile à comprendre reste la construction de la règle d'accès. Celle-ci est constituée de trois paramètres qui définissent un utilisateur ou un groupe, un droit d'accès et un type de contrôle d'accès. Le premier paramètre, `Identity`, désigne l'utilisateur ou le groupe à ajouter à la règle d'accès. Le deuxième, `FileSystemRights`, est plus subtil car il demande de comprendre les droits du système de fichiers avant de pouvoir définir l'accès. La commande suivante génère la liste des droits reconnus :

```
PS C:\> [enum]::GetNames([System.Security.AccessControl.FileSystemRights])
ListDirectory
ReadData
WriteData
CreateFiles
```

```
CreateDirectories
AppendData
ReadExtendedAttributes
WriteExtendedAttributes
Traverse
ExecuteFile
DeleteSubdirectoriesAndFiles
ReadAttributes
WriteAttributes
Write
Delete
ReadPermissions
Read
ReadAndExecute
Modify
ChangePermissions
TakeOwnership
Synchronize
FullControl
PS C:\>
```

À partir de cette liste, nous pouvons définir un seul droit, comme Modify (*modifier*), ou combiner des droits dans une liste, comme Read (*lire*), Write (*écrire*) et Delete (*supprimer*). Le troisième paramètre, `AccessControlType`, accepte uniquement les valeurs Allow (*autoriser*) et Deny (*refuser*).

Fonctions PowerShell

Comme pour `SubInACL`, nous pouvons développer un ensemble de fonctions réutilisables pour la gestion des autorisations. Voici des exemples de telles fonctions :

```
#-------------------------------------------------
# Clear-Inherit
#-------------------------------------------------
# Usage :        Se protéger contre les règles d'accès héritées et
#                retirer toutes les règles héritées indiquées.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").

function Clear-Inherit{
    param ($Object)
```

```
    $SD = get-acl $Object
    $SD.SetAccessRuleProtection($True, $False)
    set-acl $Object $SD
}
```

`Clear-Inherit` n'est peut-être pas le nom approprié pour cette fonction car, si elle empêche l'application des autorisations héritées depuis l'objet parent et si elle efface les autorisations héritées de l'objet racine et des sous-objets, elle efface également les autorisations définies sur les sous-objets. Par conséquent, avant de l'invoquer, il est préférable que vous deveniez le propriétaire de l'objet ou que vous vérifiiez que vous avez défini explicitement vos droits sur l'objet du système de fichiers racines. Si vous n'êtes pas certain d'avoir accès aux objets du système de fichiers, vous risquez de rencontrer des messages "accès refusé" après avoir effacé les droits hérités.

La fonction suivante, `Set-Owner`, fixe le propriétaire d'un objet du système de fichiers :

```
#-----------------------------------------------
# Set-Owner
#-----------------------------------------------
# Usage :        Fixer le propriétaire d'un dossier ou d'un fichier.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").
# $Identity :    Nom d'utilisateur ou de groupe ("Administrateurs" ou
#                "monDomaine\utilisateur1").

function Set-Owner{
       param ($Object,
       [System.Security.Principal.NTAccount]$Identity)

    # Obtenir l'élément à modifier.
    $Item = get-item $Object

    # Fixer son propriétaire.
    $SD = $Item.GetAccessControl()
    $SD.SetOwner($Identity)
    $Item.SetAccessControl($SD)
    }
```

Ensuite, la fonction `Clear-SD` permet d'effacer le descripteur de sécurité d'un objet du système de fichiers :

```
#-----------------------------------------------
# Clear-SD
#-----------------------------------------------
# Usage :        Effacer toutes les autorisations d'un dossier ou d'un fichier.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").

function Clear-SD{
    param ($Object)

    # Obtenir le descripteur de sécurité de l'objet.
    $SD = get-acl $Object

    # Fixer le descripteur de sécurité à Tout le monde - Contrôle total.
    #
    # Effectivement, cela ne fait pas partie des bonnes pratiques. Si cela ne
    # convient pas, fixer le descripteur de sécurité à l'utilisateur
    # courant.
    $SD.SetSecurityDescriptorSddlForm("D:PAI(A;OICI;FA;;;WD)")
    set-acl $Object $SD
    }
```

Bien que la fonction `Clear-SD` ne soit pas utilisée dans le script de gestion du système de fichiers décrit plus loin, elle illustre bien la définition d'un descripteur de sécurité à l'aide du **langage SDDL** (*Security Descriptor Definition Language*). SDDL permet de décrire un descripteur de sécurité sous forme d'une chaîne de texte. Si la fonction `Clear-SD` est utilisée, le descripteur de sécurité d'un objet est effacé, puis il est fixé à FullControl (*contrôle total*) pour le groupe Everyone (*tout le monde*), la chaîne `"D:PAI(A;OICI;FA;;;WD)"` étant employée.

INFO

Pour plus d'informations sur la construction d'un descripteur de sécurité à l'aide d'une chaîne, consultez la page http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/security_descriptor_string_format.asp.

La fonction suivante, `Add-ACE`, accorde des droits à un utilisateur ou un groupe sur un objet du système de fichiers. Bien qu'elle soit analogue à l'exemple donné au début de cette section, elle montre également comment contrôler les paramètres d'héritage d'une nouvelle ACE (*Access Control Entry*) avec les énumérations `System.Security.AccessControl.PropagationFlags` et `System.Security.AccessControl.InheritanceFlags` :

```
#-----------------------------------------------
# Add-ACE
#-----------------------------------------------
# Usage :        Accorder des droits à un dossier ou un fichier.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").
# $Identity :    Nom d'utilisateur ou de groupe ("Administrateurs" ou
#                "monDomaine\utilisateur1").
# $AccessMask :  Les droits à utiliser pour la création de la règle d'accès
#                ("FullControl", "ReadAndExecute, Write", etc.).
# $Type :        Accorder ou refuser les droits ("Allow" ou "Deny").

function Add-ACE{
    param ($Object,
           [System.Security.Principal.NTAccount]$Identity,
           [System.Security.AccessControl.FileSystemRights]$AccessMask,
           [System.Security.AccessControl.AccessControlType]$Type)

    $InheritanceFlags = `
        [System.Security.AccessControl.InheritanceFlags]`
        "ContainerInherit, ObjectInherit"
    $PropagationFlags = `
        [System.Security.AccessControl.PropagationFlags]"None"

    # Obtenir le descripteur de sécurité de l'objet.
    $SD = get-acl $Object

    # Ajouter les règles d'accès.
    $Rule = new-object `
        System.Security.AccessControl.FileSystemAccessRule($Identity, `
        $AccessMask, $InheritanceFlags, $PropagationFlags, $Type)

    $SD.AddAccessRule($Rule)
    set-acl $Object $SD
    }
```

Ne vous laissez pas perturber par le nom de ces indicateurs. Ils déterminent l'application d'une ACE à un objet et à tous ses sous-objets. Dans la fonction `Add-ACE`, les indicateurs sont définis de manière que l'ACE soit appliquée à "Ce dossier et à tous les sous-dossiers et fichiers". Autrement dit, elle est non seulement appliquée à l'objet en cours de modification, mais elle est également propagée à tous les sous-objets de cet objet. Cette propagation devrait suffire pour la plupart des tâches de gestion du système de fichiers. Si ce n'est pas le cas, vous pouvez toujours modifier la fonction afin qu'elle accepte des paramètres d'héritage en arguments.

La dernière fonction se nomme `Remove-ACE`. Elle supprime une ACE dans une ACL :

```
#-----------------------------------------------
# Remove-ACE
#-----------------------------------------------
# Usage :        Retirer des droits à un dossier ou un fichier.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").
# $Identity :    Nom d'utilisateur ou de groupe ("Administrateurs" ou
#                "monDomaine\utilisateur1").
# $AccessMask :  Les droits à utiliser pour la création de la règle d'accès
#                ("FullControl", "ReadAndExecute, Write", etc.).
# $Type :        Accorder ou refuser les droits ("Allow" ou "Deny").

function Remove-ACE{
    param ($Object,
        [System.Security.Principal.NTAccount]$Identity,
        [System.Security.AccessControl.FileSystemRights]$AccessMask,
        [System.Security.AccessControl.AccessControlType]$Type)

    # Obtenir le descripteur de sécurité de l'objet.
    $SD = get-acl $Object

    # Retirer la règle l'accès.
    $Rule = new-object `
        System.Security.AccessControl.FileSystemAccessRule($Identity, `
        $AccessMask, $Type)
    $SD.RemoveAccessRule($Rule)
    set-acl $Object $SD
    }
```

De VBScript à PowerShell

Si cet ouvrage présente des applications pratiques de PowerShell, il montre également comment convertir des scripts VBScript en scripts PowerShell. Le premier exemple est un script de création de comptes pour l'entreprise companyabc.com, un fournisseur d'accès à Internet à la croissance rapide. Lors de l'ajout de nouveaux comptes, un dossier de site Web est également créé pour chaque compte. Pour cela, un modèle de structure est recopié vers les dossiers de site Web des nouveaux utilisateurs. Par le passé, la société companyabc.com demandait à des employés ou à des contractuels de créer ces dossiers et de fixer les autorisations de l'arborescence.

Après plusieurs erreurs de configuration des autorisations et des suppressions accidentelles de dossiers, le service informatique a décidé que cette manière de procéder était loin d'être satisfaisante. Pour remplacer le processus manuel, il a souhaité automatiser la création du dossier Web de l'utilisateur, la copie de la structure modèle dans ce dossier et la définition des autorisations.

Le script ProvisionWebFolders.wsf

`ProvisionWebFolders.wsf` est un script VBScript de type WSF (*Windows Script File*) développé pour répondre aux besoins d'automation de la société companyabc.com. Vous le trouverez dans le dossier `Scripts\Chapitre 6\ProvisionWebFolders` et vous pouvez le télécharger depuis **www.pearsoneducation.fr**. Ce script attend deux paramètres.

Tout d'abord, l'argument de `templatepath` doit désigner le chemin du dossier modèle à recopier dans le dossier Web des nouveaux utilisateurs. Ensuite, l'argument d'`importfile` doit préciser le nom d'un fichier CSV, qui définit les nouveaux utilisateurs, et l'emplacement de leur dossier Web. Voici la commande qui permet d'exécuter le script `ProvisionWebFolders.wsf`. Un exemple d'exécution est présenté à la Figure 6.1 :

```
cscript ProvisionWebFolders.wsf /templatepath:".\Template" /importfile:"
.\users.csv"
```

Voici les actions effectuées par le script `ProvisionWebFolders.wsf` :

1. Il vérifie le chemin du dossier modèle.
2. Ensuite, il ouvre et lit le contenu du fichier CSV (nouveaux utilisateurs et emplacement du dossier) dans un tableau.

```
C:\WINDOWS\system32\cmd.exe
C:\Scripts>cscript ProvisionWebFolders.wsf /templatepath:".\template" /importfil
e:".\users.csv"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

#########################################
#         ProvisionWebFolders           #
#########################################


Vérification du chemin des modèles              [OK]
Vérification du fichier importé         [OK]

Création des dossiers web :
 D:\Work\WWWs\Tyson [COPIE]
  Administrateurs: [Succès de SetOwner] sur D:\Work\WWWs\Tyson
  Administrateurs: [Succès de SetOwner] sur D:\Work\WWWs\Tyson\*.*
  Autorisations effacées sur D:\Work\WWWs\Tyson
  Autorisations effacées sur D:\Work\WWWs\Tyson\*.*
  Administrateurs: F [Succès de AddPerm] sur D:\Work\WWWs\Tyson
  Administrateurs: F [Succès de AddPerm] sur D:\Work\WWWs\Tyson\*.*
  SYSTEM: F [Succès de AddPerm] sur D:\Work\WWWs\Tyson
  SYSTEM: F [Succès de AddPerm] sur D:\Work\WWWs\Tyson\*.*
  Tyson: F [Succès de AddPerm] sur D:\Work\WWWs\Tyson
  Tyson: F [Succès de AddPerm] sur D:\Work\WWWs\Tyson\*.*

 D:\Work\WWWs\Maiko [COPIE]
  Administrateurs: [Succès de SetOwner] sur D:\Work\WWWs\Maiko
  Administrateurs: [Succès de SetOwner] sur D:\Work\WWWs\Maiko\*.*
  Autorisations effacées sur D:\Work\WWWs\Maiko
  Autorisations effacées sur D:\Work\WWWs\Maiko\*.*
  Administrateurs: F [Succès de AddPerm] sur D:\Work\WWWs\Maiko
  Administrateurs: F [Succès de AddPerm] sur D:\Work\WWWs\Maiko\*.*
  SYSTEM: F [Succès de AddPerm] sur D:\Work\WWWs\Maiko
  SYSTEM: F [Succès de AddPerm] sur D:\Work\WWWs\Maiko\*.*
  Maiko: F [Succès de AddPerm] sur D:\Work\WWWs\Maiko
  Maiko: F [Succès de AddPerm] sur D:\Work\WWWs\Maiko\*.*

 D:\Work\WWWs\Garett [COPIE]
  Administrateurs: [Succès de SetOwner] sur D:\Work\WWWs\Garett
  Administrateurs: [Succès de SetOwner] sur D:\Work\WWWs\Garett\*.*
  Autorisations effacées sur D:\Work\WWWs\Garett
  Autorisations effacées sur D:\Work\WWWs\Garett\*.*
  Administrateurs: F [Succès de AddPerm] sur D:\Work\WWWs\Garett
  Administrateurs: F [Succès de AddPerm] sur D:\Work\WWWs\Garett\*.*
  SYSTEM: F [Succès de AddPerm] sur D:\Work\WWWs\Garett
  SYSTEM: F [Succès de AddPerm] sur D:\Work\WWWs\Garett\*.*
  Garett: F [Succès de AddPerm] sur D:\Work\WWWs\Garett
  Garett: F [Succès de AddPerm] sur D:\Work\WWWs\Garett\*.*
```

Figure 6.1

Exécution du script ProvisionWebFolders.wsf.

3. Pour chaque utilisateur dans le tableau, il invoque `xcopy` afin de recopier l'arborescence du dossier modèle dans le dossier Web du nouvel utilisateur.
4. Il utilise ensuite `SubInACL` pour fixer les autorisations de chaque dossier :
   - Administrateurs : Propriétaire
   - Administrateurs : Contrôle total
   - Système : Contrôle total
   - Nouvel utilisateur : Contrôle total

INFO

Ce script utilise différentes fonctions de sortie vers la console ou un fichier de journalisation : `Mess`, `StatStart` et `StatDone`. Lorsque vous écrivez des scripts destinés aux administrateurs qui ne sont pas eux-mêmes des développeurs de scripts, faites en sorte que les interactions avec l'utilisateur soient cohérentes. Cela améliore l'utilisabilité des scripts et leur donne un niveau professionnel. Le code source de ces fonctions se trouve à la fin du script.

Notre premier exemple de code est constitué des éléments XML initiaux d'un fichier WSF. Ils définissent les paramètres acceptés, décrivent le script, donnent des exemples d'utilisation et précisent le langage employé :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<package>
<job id="ProvisionWebFolders">
    <runtime>
        <description>
**********************************************************************
Ce script crée le dossier Web des utilisateurs indiqués dans une liste.
**********************************************************************
        </description>
        <named name="templatepath" helpstring="Chemin du modèle de
l'arborescence de dossiers à copier." type="string" required="1" />
        <named name="importfile" helpstring="Chemin du fichier CSV à
importer." type="string" required="1" />
        <example>
Exemple :
cscript ProvisionWebFolders.wsf /templatepath:"C:\Dossiers Modèles\Dossier1"
/importfile:"c:\temp\utilisateurs.csv"
        </example>
    </runtime>
<script language="VBScript">
<![CDATA[
```

Ensuite, le script vérifie que les arguments des paramètres obligatoires `templatepath` et `importfile` sont définis. Si ce n'est pas le cas, il affiche les informations d'utilisation (voir le

code précédent) sur la console et se termine. Si les arguments sont définis, il établit son environnement en définissant les variables utilisées par la suite :

```
On Error Resume Next

'====================================================================
' Vérifier les arguments obligatoires.
'====================================================================
If WScript.Arguments.Named.Exists("templatepath") = FALSE Then
    WScript.Arguments.ShowUsage()
    WScript.Quit
End If

If WScript.Arguments.Named.Exists("importfile") = FALSE Then
    WScript.Arguments.ShowUsage()
    WScript.Quit
End If

'====================================================================
' Définir l'environnement de travail.
'====================================================================
Const ForReading = 1
ReDim arrTargs(0)
Dim StdOut
Dim FSO, objWS
Dim strTemplatePath, strImportFile

Set StdOut = WScript.StdOut
Set FSO = CreateObject("Scripting.FileSystemObject")
Set objWS = CreateObject("WScript.Shell")

strTemplatePath = WScript.Arguments.Named("templatepath")
strImportFile = WScript.Arguments.Named("importfile")
```

L'exemple de code suivant correspond au début de l'automation. Tout d'abord, le script affiche son en-tête sur la console, puis vérifie que `templatepath` correspond à un chemin valide du système de fichiers. Si ce n'est pas le cas, il se termine. Vous remarquerez comment les

informations de validité du chemin données par `templatepath` et l'état de l'exécution du script sont affichés sur la console afin que l'administrateur puisse les consulter à l'aide des fonctions `StatStart` et `StatDone` :

```
'==================================================================
' Commencer le travail.
'==================================================================
Mess "#######################################"
Mess "#         ProvisionWebFolders          #"
Mess "#######################################"
Mess vbNullString
'==================================================================
Mess vbNullString

'--------------------
' Confirmer l'existence de TemplatePath.
'--------------------
StatStart "Vérification du chemin des modèles"

If (FSO.FolderExists(strTemplatePath)) Then
    StatDone
Else
    StdOut.WriteLine("Erreur fatale : le chemin des modèles n'existe pas...")
    WScript.Quit()
End If
```

Dans le code suivant, la fonction `ParseFile` lit chaque ligne du fichier CSV, en sautant la première, et l'ajoute en tant qu'élément d'un tableau existant. Elle est écrite de manière à appeler la fonction `Xerror` en cas d'erreur. `Xerror` stoppe l'exécution, affiche l'erreur sur la console et quitte le script :

```
'--------------------
' Vérifier le fichier CSV.
'--------------------
StatStart "Vérification du fichier importé"
    ParseFile strImportFile, arrTargs
StatDone
```

L'exemple de code suivant montre l'utilisation de `xcopy` pour créer le dossier Web d'un utilisateur et y recopier la structure modèle :

```
'-------------------
' Créer les dossiers Web.
'-------------------
Mess vbNullString
Mess "Création des dossiers Web :"

For Each Targ In arrTargs
    arrTargRecord = split(Targ, ",")
    strUserName = arrTargRecord(0)
    strPath = arrTargRecord(1)

    StdOut.Write(" " & strPath)
    StdOut.Write("\" & strUserName)

    strCommand = "xcopy """ & strTemplatePath & """ """ & strPath & "\" _
        & strUserName & """ /O /E /I /Y"

    ErrorCode = objWS.Run(strCommand, 0, TRUE)

    If ErrorCode <> 0 Then
        StdOut.WriteLine(" [ECHEC][Commande invoquée : " & strCommand & "]")
    Else
        StdOut.WriteLine(" [COPIE]")
```

Pour l'appel à `xcopy`, le script utilise une chaîne qui définit la commande (`strCommand`) et un objet `WScript.Shell` nommé `objWS`. Nous pourrions obtenir les mêmes résultats avec un objet FSO, mais `xcopy` réduit le nombre de lignes de code nécessaires à cette opération.

Une fois le dossier Web de l'utilisateur créé, l'étape suivante consiste à en fixer les autorisations. Pour cela, le script se sert de l'outil `SubInACL` en invoquant les fonctions `DumpPerm`, `SetOwner` et `AddPerm`. Dans l'exemple de code suivant, faites particulièrement attention à la manière dont les fonctions sont appelées deux fois lorsque les autorisations d'un objet sont modifiées.

```
        ' Administrateurs devient le propriétaire du dossier.
        SetOwner strPath & "\" & strUserName, "Administrateurs"
        Mess vbNullString

        ' Administrateurs devient le propriétaire de tout le contenu du dossier.
        SetOwner strPath & "\" & strUserName & "\*.*", "Administrateurs"
        Mess vbNullString

        ' Effacer les autorisations du dossier.
        DumpPerm strPath & "\" & strUserName
        Mess vbNullString

        ' Effacer les autorisations du contenu du dossier.
        DumpPerm strPath & "\" & strUserName & "\*.*"
        Mess vbNullString

        ' Ajouter le groupe Administrateurs.
        AddPerm strPath & "\" & strUserName, "Administrateurs", "F"
        Mess vbNullString

        ' Ajouter le groupe Administrateurs à tout le contenu du dossier.
        AddPerm strPath & "\" & strUserName & "\*.*", "Administrateurs", "F"
        Mess vbNullString

        ' Ajouter SYSTEM.
        AddPerm strPath & „\" & strUserName, „SYSTEM", „F"
        Mess vbNullString

        ' Ajouter SYSTEM à tout le contenu du dossier.
        AddPerm strPath & "\" & strUserName & "\*.*", "SYSTEM", "F"
        Mess vbNullString

        ' Ajouter l'utilisateur.
        AddPerm strPath & "\" & strUserName, strUserName, "F"
        Mess vbNullString

        ' Ajouter l'utilisateur à tout le contenu du dossier moins
        AddPerm strPath & "\" & strUserName & "\*.*", strUserName, "F"
        Mess vbNullString

    End If

    Mess vbNullString

    ErrorCode = vbNullString
Next

Mess "Terminé : les dossiers Web ont été créés"
```

Le premier appel à `SubInACL` modifie les autorisations sur le dossier racine, tandis que le second modifie celles des sous-dossiers et fichiers du dossier racine. Le deuxième appel n'est sans doute pas nécessaire si les autorisations du dossier racine ont été effacées. Cependant, effacer les autorisations d'une arborescence de dossiers ne fixe pas toujours correctement les paramètres d'héritage. Certains sous-dossiers et fichiers risquent alors de ne pas hériter des autorisations du dossier racine. En appelant une seconde fois `SubInACL` pour modifier les autorisations des sous-dossiers et des fichiers du dossier racine, il semble que nous résolvions le problème d'héritage.

Le dernier exemple de code est constitué des procédures et des fonctions utilisées tout au long du script et des éléments XML terminant le script. Il est inutile de détailler cette partie finale du script car ces procédures et fonctions sont suffisamment compréhensibles par elles-mêmes et ont déjà été présentées :

```
'=====================================================================
' Procédures.
'=====================================================================
'--------------------
' Procédure générale pour les messages.
'--------------------
Sub Mess(Message)
    ' Écrire sur la console.
    StdOut.WriteLine(Message)
End Sub

'--------------------
' Procédure générale pour le début d'un message.
'--------------------
Sub StatStart(Message)
    , Écrire sur la console.
    StdOut.Write(Message)
End Sub

'--------------------
' Procédure générale pour la fin d'un message.
'--------------------
Sub StatDone
    ' Écrire sur la console.
    StdOut.Write(vbTab & vbTab)
    StdOut.WriteLine("[OK]")
End Sub
```

```
'-------------------
' Procédure générale pour Xerror.
'-------------------
Sub Xerror
    If Err.Number <> 0 Then
        ' Écrire sur la console.
        StdOut.WriteLine(" Erreur fatale : " & CStr(Err.Number) _
            & " " & Err.Description)

        WScript.Quit()
    End If
End Sub

'==================================================================
' Fonctions.
'==================================================================
Function ParseFile(file, arrname)
    ' Cette fonction lit un fichier et retourne un tableau de son contenu.
    ' La première ligne est sautée !!!
    On Error Resume Next
    count = -1

    ' Ouvrir le fichier en lecture.
    Set objFile = FSO.OpenTextFile(file, ForReading)
    objFile.SkipLine ' note : cette ligne donne les en-têtes des colonnes.
    Xerror

    ' Lire chaque ligne du fichier et la placer dans un tableau.
    Do While objFile.AtEndOfStream <> True
        count = count + 1
        If count > UBound(arrname) Then ReDim Preserve arrname(count)
            arrname(count) = objFile.Readline
    Loop
    Xerror

    ' Fermer le fichier.
    objFile.Close()
    Set objFile = Nothing
    count = 0
End Function

Function ParseTempFile(path)
    ' Ouvrir un fichier en lecture.
    Set objFile = FSO.OpenTextFile(path, ForReading)
```

```
    tempfileinfo = vbNullString

    Do While objFile.AtEndOfStream <> True
        tempfileinfo = tempfileinfo & objFile.Readline
    Loop

    ParseTempFile = tempfileinfo

    objFile.Close()
    Set objFile = Nothing
End Function

Function SetOwner(path, account)
    ' Fixer le propriétaire d'un dossier ou de sous-dossiers.
    On Error Resume Next
    strCommand = "subinacl /verbose /output=log.temp " _
        & "/subdirectories """ & path & """ /setowner=""" & account & """"

    ErrorCode = objWS.Run(strCommand, 0, TRUE)

    If ErrorCode <> 0 Then
        StdOut.Write("  " & account & ":" _
            & " [Échec de SetOwner] sur " & path)
    Else
        return = inStr(1, ParseTempFile("log.temp"), "ne sera pas examiné")

        If Not return = 0 Then
        StdOut.Write("  " & account & ":" _
            & " [Échec de SetOwner] sur " & path)
        Else
        StdOut.Write("  " & account & ":" _
            & " [Succès de SetOwner] sur " & path)
        End If
    End If

    ErrorCode = vbNullString
End Function

Function DumpPerm(path)
    ' Effacer les autorisations d'un dossier ou de sous-dossiers.
    On Error Resume Next
    strCommand = "subinacl /verbose /output=log.temp " _
        & "/subdirectories """ & path & """ /perm"
```

```
    ErrorCode = objWS.Run(strCommand, 0, TRUE)

    If ErrorCode <> 0 Then
        StdOut.Write("  Autorisations non effacées sur " & path)
    Else
        StdOut.Write("  Autorisations effacées sur " & path)
    End If

    ErrorCode = vbNullString
End Function

Function AddPerm(path, account, access)
    ' Accorder des droits à un utilisateur sur un dossier ou des sous-dossiers.
    On Error Resume Next
    strCommand = "subinacl /verbose /output=log.temp" _
        & " /subdirectories """ & path & """ /grant=""" _
        & account & """ =" & access

    ErrorCode = objWS.Run(strCommand, 0, TRUE)

    If ErrorCode <> 0 Then
        StdOut.Write("  " & account & ": " & access _
            & " [Échec de AddPerm] sur " & path)
    Else
        return = inStr(1, ParseTempFile("log.temp"), "ne sera pas examiné")

        If Not return = 0 Then
            StdOut.Write("  " & account & ": " & access _
                & " [Échec de AddPerm] sur " & path)
        Else
            StdOut.Write("  " & account & ": " & access _
                & " [Succès de AddPerm] sur " & path)
        End If
    End If

    ErrorCode = vbNullString
End Function

]]>
  </script>

</job>
</package>
```

Le script ProvisionWebFolders.ps1

`ProvisionWebFolders.ps1` est une conversion en PowerShell du script `ProvisionWebFolders.wsf`. Vous en trouverez une version opérationnelle dans le dossier `Scripts\Chapitre 6\ProvisionWebFolders` et en téléchargement sur le site **www.pearsoneducation.fr**.

Ce script attend deux paramètres. Tout d'abord, l'argument de `TemplatePath` doit désigner le chemin du dossier modèle à recopier dans le dossier Web des nouveaux utilisateurs. Ensuite, l'argument d'`ImportFile` doit préciser le nom d'un fichier CSV qui définit les nouveaux utilisateurs et l'emplacement de leur dossier Web. Voici la commande qui permet d'exécuter le script `ProvisionWebFolders.ps1`. Un exemple d'exécution est présenté à la Figure 6.2 :

```
PS D:\Travail> .\ProvisionWebFolders.ps1 .\template .\users.csv
```

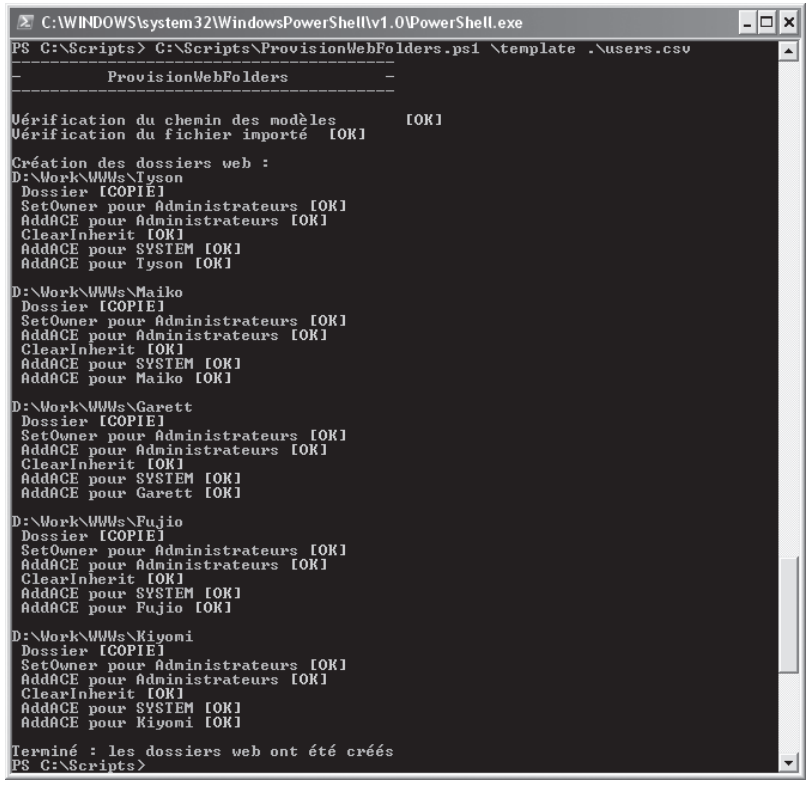


Figure 6.2

Exécution du script ProvisionWebFolders.ps1.

Voici les actions effectuées par le script `ProvisionWebFolders.ps1` :

1. Il vérifie que le chemin du dossier modèle existe.
2. Ensuite, il vérifie que le chemin du fichier importé existe.
3. Il importe le fichier CSV dans la variable `$Targets`.
4. Pour chaque utilisateur dans la variable `$Targets`, il copie l'arborescence du dossier modèle dans le dossier Web du nouvel utilisateur.
5. Pour finir, le script fixe les autorisations de chaque dossier :
   - Administrateurs : Propriétaire
   - Administrateurs : Contrôle total
   - Système : Contrôle total
   - Nouvel utilisateur : Contrôle total

Le premier exemple de code contient l'en-tête du script `ProvisionWebFolders.ps1`. Cet en-tête inclut des informations sur l'objectif du script, la date de sa dernière mise à jour, ainsi que le nom de son auteur. Les paramètres du script viennent ensuite :

```
##################################################
# ProvisionWebFolders.ps1
# Ce script crée le dossier Web des nouveaux utilisateurs.
#
# Créé le : 12/09/2006
# Auteur : Tyson Kopczynski
##################################################
param([string] $TemplatePath = $(throw write-host `
    "Veuillez préciser le chemin du modèle source de la structure de dossiers" `
    "à copier." -Foregroundcolor Red), [string] $ImportFile = $(throw `write-host
    "Veuillez préciser le nom du fichier CSV à importer." `-Foregroundcolor Red))
```

Notez l'utilisation du mot clé `throw` dans la déclaration `param`. Il génère une erreur lorsqu'un argument de paramètre n'est pas précisé. Cette technique impose la définition d'un paramètre en arrêtant l'exécution du script et en affichant à l'utilisateur des informations concernant le paramètre requis, avec l'applet de commande `Write-Host`. Cette applet accepte un paramètre `Foregroundcolor`, qui fixe la couleur du texte affiché. Cette fonctionnalité permet d'attirer l'attention sur des détails de l'état du script (voir Figure 6.3).

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> C:\Scripts\ProvisionWebFolders.ps1 \template .\inconnu.csv
----------------------------------------
-          ProvisionWebFolders          -
----------------------------------------

Vérification du chemin des modèles        [OK]
Vérification du fichier importé  .\inconnu.csv n'est pas un fichier valide !
ScriptHalted
Au niveau de C:\Scripts\ProvisionWebFolders.ps1 : 111 Caractère : 10
+     throw  <<<< write-host `t "$ImportFile n'est pas un fichier valide !" -Fo
regroundcolor Red
PS C:\Scripts>
```

Figure 6.3

Texte affiché en vert et en rouge sur la console pour indiquer l'état du script.

Ensuite, le script place les fonctions de gestion du système de fichiers dans sa portée. Il est inutile de les expliquer plus en détail car elles ont déjà été décrites précédemment :

```
##################################################
# Fonctions.
##################################################
#-------------------------------------------------
# Clear-Inherit
#-------------------------------------------------
# Usage :        Se protéger contre les règles d'accès héritées et
#                retirer toutes les règles héritées indiquées.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").

function Clear-Inherit{
    param ($Object)

    $SD = get-acl $Object
    $SD.SetAccessRuleProtection($True, $False)
    set-acl $Object $SD
    }
```

```
#-----------------------------------------------
# Set-Owner
#-----------------------------------------------
# Usage :        Fixer le propriétaire d'un dossier ou d'un fichier.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").
# $Identity :    Nom d'utilisateur ou de groupe ("Administrateurs" ou
#                "monDomaine\utilisateur1").

function Set-Owner{
       param ($Object,
       [System.Security.Principal.NTAccount]$Identity)

    # Obtenir l'élément à modifier.
    $Item = get-item $Object

    # Fixer son propriétaire.
    $SD = $Item.GetAccessControl()
    $SD.SetOwner($Identity)
    $Item.SetAccessControl($SD)
    }

#-----------------------------------------------
# Add-ACE
#-----------------------------------------------
# Usage :        Accorder des droits à un dossier ou un fichier.
# $Object :      Le chemin du dossier ou du fichier ("c:\monDossier" ou
#                "c:\monFichier.txt").
# $Identity :    Nom d'utilisateur ou de groupe ("Administrateurs" ou
#                "monDomaine\utilisateur1").
# $AccessMask :  Les droits à utiliser pour la création de la règle d'accès
#                ("FullControl", "ReadAndExecute, Write", etc.).
# $Type :        Accorder ou refuser les droits ("Allow" ou "Deny").

function Add-ACE{
    param ($Object,
           [System.Security.Principal.NTAccount]$Identity,
           [System.Security.AccessControl.FileSystemRights]$AccessMask,
           [System.Security.AccessControl.AccessControlType]$Type)

    $InheritanceFlags = `
        [System.Security.AccessControl.InheritanceFlags]`
        "ContainerInherit, ObjectInherit"
```

```
    $PropagationFlags = `
        [System.Security.AccessControl.PropagationFlags]"None"

    # Obtenir le descripteur de sécurité de l'objet.
    $SD = get-acl $Object

    # Ajouter les règles d'accès.
    $Rule = new-object `
        System.Security.AccessControl.FileSystemAccessRule($Identity, `
        $AccessMask, $InheritanceFlags, $PropagationFlags, $Type)

    $SD.AddAccessRule($Rule)
    set-acl $Object $SD
}
```

Le prochain exemple de code commence la partie d'automation du script. Tout d'abord, il vérifie si la chaîne contenue dans la variable `$TemplatePath` correspond à un chemin de dossier valide, puis si la variable `$ImportFile` désigne un chemin de fichier valide. Pour réaliser ces tests, les instructions `if...then` utilisent l'applet de commande `Test-Path`. Cette applet très pratique permet de vérifier si un dossier ou un fichier (`-pathType container` ou `leaf`) est valide. Si l'un des chemins n'est pas valide, l'exécution du script est arrêtée et des informations concernant les chemins invalides sont affichées à l'opérateur de script :

```
##################################################
# Code principal.
##################################################
write-host "----------------------------------------"
write-host "-         ProvisionWebFolders          -"
write-host "----------------------------------------"
write-host
write-host "Vérification du chemin des modèles" -NoNewLine

if (!(test-path $TemplatePath -pathType container)){
    throw write-host `t "$TemplatePath n'est pas un répertoire valide !"
        -Foregroundcolor Red
    }
else {
    write-host `t "[OK]" -Foregroundcolor Green
    }

write-host "Vérification du fichier importé" -NoNewLine
```

```
if (!(test-path $ImportFile -pathType leaf)){
    throw write-host `t "$ImportFile n'est pas un fichier valide !"
        -Foregroundcolor Red
    }
else {
    write-host `t "[OK]" -Foregroundcolor Green
    }
```

Puis, nous définissons les autres variables employées dans le script. La première, `$Owner`, définit le propriétaire de l'arborescence du dossier Web de chaque utilisateur, qui, dans ce cas, est le groupe local Administrateurs. Ensuite, nous définissons la variable `$Targets` à l'aide de l'applet de commande `Import-Csv`, laquelle permet de lire des valeurs depuis un fichier CSV importé (`$ImportFile`) dans la variable `$Targets`, qui sert ensuite à créer les dossiers Web des nouveaux utilisateurs :

```
#--------------------
# Définir les variables.
#--------------------
$Owner = "Administrateurs"
$Targets = import-csv $ImportFile
```

Dans le code suivant, le script utilise le chemin et l'utilisateur indiqués dans la variable `$Targets` pour construire le chemin final, en invoquant l'applet de commande `Join-Path`. Puis, l'applet `Copy-Item` copie les dossiers modèles vers le chemin de destination :

```
#--------------------
# Créer les dossiers Web.
#--------------------
write-host
write-host "Création des dossiers Web :"

foreach ($Target in $Targets){
    $Path = join-path $Target.DestPath $Target.UserName
    $UserName = $Target.UserName

    write-host $Path
       if (!(test-path $Path)){
        copy-item $TemplatePath -Destination $Path -Recurse `
```

```
        -ErrorVariable Err -ErrorAction SilentlyContinue

        if (!$Err){
            write-host " Dossier " -NoNewLine
            write-host "[COPIE]" -Foregroundcolor Green

            # Pour arrêter les boucles.
            $Err = $False
```

Ensuite, la fonction `Set-Owner` définit le groupe local Administrateurs comme propriétaire de l'arborescence du dossier Web de l'utilisateur :

```
            .{
              trap{write-host "[ERREUR] Impossible de changer de propriétaire !" `
                   -Foregroundcolor Red;
                   $Script:Err = $True;
                   Continue}

              # Fixer le propriétaire.
              write-host " SetOwner pour $Owner " -NoNewLine

              Set-Owner $Path $Owner

              if ($Err -eq $False){
                  $Items = get-childitem $Path -Recurse
                  [void]($Items | foreach-object `
                          {Set-Owner $_.FullName $Owner})
                  }
              else{
                  # Arrêter la boucle.
                  Continue
                  }

              write-host "[OK]" -Foregroundcolor Green
            }
```

Vous vous demandez peut-être pourquoi le code de `Set-Owner` est placé à l'intérieur d'un bloc de script. L'opérateur d'appel point (`.`) qui précède le bloc de script demande à PowerShell d'exécuter ce bloc au sein de la portée courante. S'il n'était pas utilisé, PowerShell n'exécuterait pas le bloc de script. En créant un bloc de script indépendant pour gérer l'appel

à `Set-Owner`, nous nous assurons que la portée de l'instruction `trap` est limitée à ce bloc de code. Cette technique de contrôle de la portée de `trap` sera souvent employée dans ce livre.

```
.{
   trap{write-host "[ERREUR] Impossible d'ajouter des droits !" `
        -Foregroundcolor Red;
        $Script:Err = $True;
        Continue}

    # Ajouter le groupe Administrateurs.
    write-host " AddACE pour Administrateurs " -NoNewLine

    Add-ACE $Path "Administrateurs" "FullControl" "Allow"

    if ($Err -eq $False){
        write-host "[OK]" -Foregroundcolor Green
        }
    else{
        # Arrêter la boucle.
        Continue
        }
}

.{
    trap{write-host "[ERREUR] Impossible d'effacer les
    autorisations héritées !" -Foregroundcolor Red;
        $Script:Err = $True;
        Continue}

    # Effacer les autorisations héritées.
    write-host " ClearInherit " -NoNewLine

    Clear-Inherit $Path

    if ($Err -eq $False){
        write-host "[OK]" -Foregroundcolor Green
        }
    else{
        # Arrêter la boucle.
        Continue
        }
}
```

Comme nous l'avons mentionné précédemment, la fonction `Clear-Inherit` supprime les autorisations héritées pour le dossier racine, ses sous-dossiers et ses fichiers, ainsi que les autorisations explicitement définies sur tous les sous-dossiers et les fichiers. Si le groupe Administrateurs n'avait pas défini explicitement des droits sur le dossier racine, la suite du script ne s'exécuterait pas par manque de droits.

INFO

Les autorisations définies explicitement sont celles qui sont spécifiées directement pour un utilisateur sur un objet. Les autorisations définies implicitement sont celles qui sont héritées ou définies par l'appartenance à un groupe.

Dans le dernier exemple de code, l'autorisation `FullControl` sur le dossier Web de l'utilisateur est donnée à `SYSTEM` et à l'utilisateur. Pour finir, le script signale à l'opérateur la fin de son travail :

```
            # Ajouter SYSTEM.
            write-host " AddACE pour SYSTEM " -NoNewLine

            if ((Add-ACE $Path "SYSTEM" "FullControl" "Allow") -eq $True){
                write-host "[OK]" -Foregroundcolor Green
                }

            # Ajouter l'utilisateur.
            write-host " AddACE pour $UserName " -NoNewLine

            if ((Add-ACE $Path $UserName "FullControl" "Allow") -eq $True){
                write-host "[OK]" -Foregroundcolor Green
                }
            }
        else {
            write-host " Dossier " -NoNewLine
            write-host "Erreur :" $Err -Foregroundcolor Red
            }
        }
    else {
        write-host " Dossier " -NoNewLine
        write-host "[EXISTE]" -Foregroundcolor Yellow
        }

    write-host
    }

write-host "Terminé : les dossiers Web ont été créés"
```

En résumé

Ce chapitre s'est attaché à décrire la gestion du système de fichiers de Windows quand on utilise WSH et PowerShell. Même si ces deux solutions de scripts disposent de méthodes de gestion du système de fichiers, le fournisseur `FileSystem` de PowerShell permet de mettre en œuvre une méthode de type source de données. Lors du développement de vos prochains scripts ou de l'utilisation de PowerShell depuis la console, vous remarquerez sans doute que la méthode de PowerShell apporte une plus grande liberté d'accès, de consultation et de manipulation du système de fichiers.

Vous avez pu comprendre les différences entre WSH et PowerShell, quant à la manipulation du système de fichiers de Windows. Ce chapitre a également expliqué la gestion des autorisations lorsque ces deux interfaces de scripts sont utilisées. Vous pensiez peut-être que la manipulation des autorisations au travers de ces interfaces était une tâche difficile. Même si c'est un peu vrai, vous devez à présent reconnaître qu'elle est loin d'être insurmontable. La gestion des autorisations à l'aide d'un script d'automation peut devenir un outil très puissant. Par exemple, vous pouvez développer des scripts qui imposent des autorisations en fonction d'une stratégie définie, qui vérifient si les autorisations respectent un modèle ou recherchent des instances sur lesquelles un utilisateur ou un groupe possède des droits.

7

PowerShell et le Registre

Dans ce chapitre

- Introduction
- Gérer le Registre depuis WSH et PowerShell
- De VBScript à PowerShell

Introduction

Ce chapitre s'intéresse à la gestion du Registre de Windows depuis PowerShell. Pour cela, il présente des exemples détaillés fondés sur WSH (*Windows Script Host*) et sur PowerShell. Ils sont donnés sous ces deux formes afin que vous puissiez passer à PowerShell en bénéficiant de votre expérience de l'écriture de scripts pour Windows. Outre la comparaison des exemples, ce chapitre propose tout un ensemble de fonctions opérationnelles pour la gestion du Registre. L'objectif est de donner au lecteur la possibilité d'appliquer les techniques de scripts PowerShell à des besoins d'automation réels.

Gérer le Registre depuis WSH et PowerShell

WSH fournit un objet pour la manipulation des applications en cours d'exécution, le lancement de nouvelles applications, la création de raccourcis, la création de menus contextuels, la gestion des variables d'environnement, la journalisation des événements et même l'accès

ou la modification du Registre local. Pour ces dernières opérations, l'objet `WshShell` offre trois méthodes :

- `RegDelete` supprime une clé ou l'une de ses valeurs du Registre.
- `RegRead` lit le contenu de la valeur indiquée depuis le Registre.
- `RegWrite` crée de nouvelles clés, ajoute une nouvelle valeur nommée à une clé existante ou modifie une valeur nommée existante.

L'utilisation de l'objet `WshShell` et de ses méthodes de manipulation du Registre est simple. Il s'agit d'un objet COM et, en tant que tel, il peut être créé à l'aide de la méthode `CreateObject()` de WSH. Une fois cette opération effectuée, les méthodes du registre de cet objet peuvent être invoquées comme n'importe quelle autre méthode dans WSH.

Dans PowerShell, l'accès au Registre se fait différemment. Comme nous l'avons expliqué au Chapitre 3, "Présentation avancée de PowerShell", PowerShell dispose d'un fournisseur intégré, Registry, qui permet d'accéder au Registre et de le manipuler sur la machine locale. Les ruches du Registre accessibles au travers de ce fournisseur sont `HKEY_LOCAL_MACHINE` (`HKLM`) et `HKEY_CURRENT_USER` (`HKCU`). Elles sont représentées par deux objets `PSDrive` supplémentaires nommés `HKLM:` et `HKCU:`.

INFO

L'objet `WshShell` n'est pas limité aux ruches `HKLM:` et `HKCU:`. Il permet également d'accéder à `HKEY_CLASSES_ROOT` (`HKCR`), `HKEY_USERS` et `HKEY_CURRENT_CONFIG`. Pour manipuler ces ruches depuis PowerShell, il faut employer l'applet de commande `Set-Location` afin de modifier l'emplacement de la racine du fournisseur Registry.

Le Chapitre 3 a également expliqué que l'utilisation du fournisseur Registry signifie que PowerShell traite les données des objets `HKLM:` et `HKCU:` comme n'importe quel magasin de données hiérarchique. Par conséquent, la manipulation des données de ces `PSDrive` ne peut se faire qu'avec les applets de commande principales de PowerShell :

```
PS C:\> set-location hkcu:
PS HKCU:\> get-childitem

   Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
```

```
SKC  VC Name                           Property
---  -- ----                           --------
  2   0 AppEvents                      {}
  2  32 Console                        {ColorTable00, ColorTable01, ColorTab...
 24   1 Control Panel                  {Opened}
  0   2 Environment                    {TEMP, TMP}
  1   6 Identities                     {Identity Ordinal, Migrated5, Last Us...
  4   0 Keyboard Layout                {}
  3   1 Printers                       {DeviceOld}
 32   1 Software                       {(default)}
  0   0 UNICODE Program Groups         {}
  2   0 Windows 3.1 Migration Status   {}
  0   1 SessionInformation             {ProgramCount}
  0   8 Volatile Environment           {LOGONSERVER, HOMESHARE, HOMEPATH, US...


PS HKCU:\> get-itemproperty 'Volatile Environment'

PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Volatile
                Environment
PSParentPath  : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName   : Volatile Environment
PSDrive       : HKCU
PSProvider    : Microsoft.PowerShell.Core\Registry
LOGONSERVER   : \\SOL
HOMESHARE     : \\taosage.internal\homes\tyson
HOMEPATH      : \
USERDNSDOMAIN : TAOSAGE.INTERNAL
CLIENTNAME    :
SESSIONNAME   : Console
APPDATA       : C:\Documents and Settings\tyson\Application Data
HOMEDRIVE     : U:


PS HKCU:\>
```

Grâce aux applets de commande principales de PowerShell, nous pouvons manipuler le Registre local, tout comme avec les méthodes de Registre de l'objet `WshShell`. Cependant, la syntaxe et la méthodologie sont un tantinet différentes. Dans WSH, nous créons un objet puis nous utilisons ses méthodes pour intervenir sur le Registre. Dans PowerShell, nous accédons

au Registre et nous le manipulons de la même manière que le système de fichiers. Par exemple, pour lire une valeur du Registre en WSH, nous invoquons la méthode `RegRead` :

```
Dim objWS
Set objWS = CreateObject("WScript.Shell")

strKey = "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\"

WScript.Echo objWS.RegRead(strKey & "ProductName")
```

Dans PowerShell, il suffit d'utiliser `Get-ItemProperty` :

```
PS C:\> $Chemin = "HKLM:\Software\Microsoft\Windows NT\CurrentVersion"
PS C:\> $Cle = get-itemproperty $Chemin
PS C:\> $Cle.ProductName
Microsoft Windows XP
PS C:\>
```

La création ou la modification d'une valeur du registre dans WSH se font avec la méthode `RegWrite` :

```
Dim objWS
Set objWS = CreateObject("WScript.Shell")

strKey = "HKEY_CURRENT_USER\Software\"

objWS.RegWrite strKey & "PSinfo", "Vive_PowerShell"

WScript.Echo objWS.RegRead(strKey & "PSinfo")
```

Dans PowerShell, la même opération emploie l'applet `Set-ItemProperty` :

```
PS C:\> $Chemin = "HKCU:\Software"
PS C:\> set-itemproperty -path $Chemin -name "PSinfo" –type "String" -value
"Vive_PowerShell"
PS C:\>
PS C:\> $Cle = get-itemproperty $Chemin
PS C:\> $Cle.PSinfo
Vive_PowerShell
PS C:\>
```

N'oubliez pas que le Registre de Windows gère différents types de valeurs. `Set-ItemProperty` accepte le paramètre `Type` pour la création ou la modification des valeurs du Registre. Les bonnes pratiques conseillent de toujours définir explicitement les valeurs lors de l'utilisation de `Set-ItemProperty`. Sinon, l'applet de commande donne à la valeur le type par défaut, c'est-à-dire `String`. Voici les autres types existants :

- `ExpandString` ;
- `Binary` ;
- `DWord` ;
- `MultiString` ;
- `Qword`.

INFO

La valeur des données doit être dans le format qui correspond à la valeur du Registre en cours de création ou de modification. Par exemple, si la valeur du registre est de type `REG_BINARY`, vous devez utiliser une valeur binaire, comme `$Bin = 101, 118, 105`.

Pour supprimer une valeur du Registre dans WSH, la méthode `RegDelete` doit être invoquée :

```
Dim objWS
Set objWS = CreateObject("WScript.Shell")

strKey = "HKEY_CURRENT_USER\Software\"

objWS.RegDelete strKey & "PSinfo"
```

Dans PowerShell, cette opération est réalisée par l'applet de commande `Remove-ItemProperty` :

```
PS C:\> $Chemin = "HKCU:\Software"
PS C:\> remove-itemproperty -path $Chemin -name "PSinfo"
PS C:\>
```

Ces exemples illustrent la manière d'accéder au Registre. Elle reste assez simple, à partir du moment où nous avons compris comment utiliser les applets de commande principales et n'oublions pas qu'elle ressemble énormément à l'accès au système de fichiers de Windows.

Cependant, aucune applet ne permet d'accéder au Registre d'une machine distante. Cette absence est tout à fait compréhensible, car il n'existe aucun fournisseur PowerShell pour accéder à des magasins de données distants. En attendant que quelqu'un écrive un fournisseur permettant de gérer à distance un Registre, nous devons nous tourner vers une méthode alternative existante, comme l'explique la section suivante.

De VBScript à PowerShell

Cette section détaille un script VBScript qui permet de lire et de manipuler le Registre, puis sa conversion en PowerShell. La société companyabc.com était en cours d'évaluation de l'efficacité de son service informatique. Lors de l'analyse des développements des scripts d'automation, les évaluateurs ont noté une répétition de certaines tâches dans de nombreux scripts. Ces tâches incluaient la création des comptes d'utilisateurs, la définition des informations des comptes, la gestion à distance des machines, la mise en œuvre des activités de maintenance, etc.

L'équipe d'évaluation a conclu que le regroupement du code répétitif dans une bibliothèque réutilisable permettrait de réduire le temps nécessaire au développement des scripts. Cette méthode simple consiste à écrire une fonction ou un script générique qui réalise une tâche fréquente, comme la génération d'un mot de passe aléatoire. Lorsqu'un nouveau script a besoin de cette opération, il n'est plus nécessaire d'écrire du code. Dans WSH et PowerShell, il suffit d'inclure ou de charger le fichier de la bibliothèque correspondante dans le script ou depuis la console.

Les exemples de scripts donnés ici incluent un ensemble de fonctions permettant de lire et de modifier le Registre sur une machine locale ou distante. Ils ont été développés pour l'entreprise companyabc.com. Pour utiliser ces fonctions, les programmeurs peuvent simplement les copier dans un script ou les appeler depuis une bibliothèque qui a été incluse ou chargée dans le script.

Outre la diminution des temps de développement des scripts, l'emploi d'un code réutilisable enregistré dans un fichier de bibliothèque permet d'obtenir du code plus normalisé et interchangeable. En réalité, Jeffrey Snover, l'architecte de PowerShell, a souvent recommandé cette bonne pratique pour l'écriture de scripts.

Le script LibraryRegistry.vbs

`LibraryRegistry.vbs` est un fichier VBScript qui fournit des fonctions de lecture et de modification du Registre sur la machine locale ou une machine distante. Vous le trouverez dans le dossier `Scripts\Chapitre 7\LibraryRegistry` et vous pouvez le télécharger depuis le site **www.pearsoneducation.fr**. Pour l'utiliser dans un autre script, il doit être inclus dans celui-ci. Ensuite, le script appelant a accès aux fonctions, aux routines, aux constantes, etc., définies dans le fichier inclus.

Dans VBScript, il existe deux méthodes pour inclure un fichier de script dans un autre. La première ne fonctionne qu'avec les fichiers VBScript (`.vbs`) et repose sur l'instruction `ExecuteGlobal`. Cette instruction attend une valeur de type chaîne et l'exécute comme une instruction VBScript dans l'espace de noms global du script. Celui-ci peut ensuite accéder au contenu de la valeur chaîne. Le code suivant illustre cette procédure :

```
' Méthode pour inclure des fichiers VBScript.
Sub Include(strFileName)
    On Error Resume Next

    Dim objFSO, objFile, strScript

    Set objFSO = CreateObject("Scripting.FileSystemObject")

    If objFSO.FileExists(strFileName) Then
        Set objFile = objFSO.OpenTextFile(strFileName)
        strScript = objFile.ReadAll
        objFile.Close

        ExecuteGlobal strScript
    End If

    Set objFSO = Nothing
    Set objFile = Nothing
End Sub
```

Cette méthode présente cependant plusieurs inconvénients. Tout d'abord, nous risquons d'écraser des variables et des fonctions globales au moment de l'exécution. Deuxièmement, il n'existe aucune bonne manière de déboguer le contenu de la chaîne passée à `ExecuteGlobal`. En effet, cette valeur n'est qu'une chaîne qui est exécutée. Troisièmement, VBScript n'offre aucune instruction d'inclusion officielle et cette méthode n'est qu'une solution de rechange.

Pour toutes ces raisons, l'instruction `ExecuteGlobal` n'est pas une solution conseillée pour inclure des fichiers dans un script. La méthode plus fiable et robuste consiste à passer par un fichier WSF, car ce format prend en charge les instructions d'inclusion :

```
<job>
   <script src="MaBibliotheque.js" language="JScript" />
   <script language="vbscript">

strEvent = "C'est les vacances !"

strDate = GetCalendarDate(strEvent)
WScript.Echo strDate

   </script>
</job>
```

Comme le montre cet exemple, une tâche VBScript dans un fichier WSF peut parfaitement inclure un fichier JScript. L'opération inverse est également possible ; une tâche JScript peut inclure un fichier VBScript. Il est également possible d'incorporer les deux types de fichiers dans un script ou de créer un seul fichier WSF qui effectue plusieurs tâches en utilisant des langages (moteurs) différents pour chacune. Quelle que soit la méthode choisie, après avoir inclus un fichier de script, nous avons accès à ses fonctions, ses constantes, ses routines, etc. depuis notre script.

Chaque fonction du script `LibraryRegistry.vbs` utilise la classe WMI `StdRegProv`, qui se trouve dans l'espace de noms `root\default`. Elles offrent des méthodes permettant de lire et de manipuler les clés et les valeurs du Registre, dans le but d'effectuer les tâches suivantes :

- vérifier qu'un utilisateur dispose des autorisations indiquées ;
- créer, énumérer et supprimer des clés du Registre ;
- créer, énumérer et supprimer des valeurs du Registre ;
- obtenir ou actualiser un descripteur de sécurité pour une clé du Registre (uniquement dans Vista ou Longhorn).

La suite de cette section donne des exemples de code qui illustrent les fonctions de `LibraryRegistry.vbs`.

La fonction `ReadRegValue` :

```
'--------------------
' ReadRegValue
'--------------------
Function ReadRegValue(strComputer, strKeyPath, strValueName, strType)
    On Error Resume Next

    const HKEY_LOCAL_MACHINE = &H80000002

    Set objReg = GetObject("winmgmts:{impersonationLevel=impersonate}!\\" _
        & strComputer & "\root\default:StdRegProv")

    If strType = "BIN" Then
        objReg.GetBinaryValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, arrValue

        ReadRegValue = arrValue
    End If
    If strType = "DWORD" Then
        objReg.GetDWORDValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, strValue

        ReadRegValue = strValue
    End If
    If strType = "EXP" Then
        objReg.GetExpandedStringValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, strValue

        ReadRegValue = strValue
    End If
    If strType = "MULTI" Then
        objReg.GetMultiStringValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, arrValue

        ReadRegValue = arrValue
    End If
    If strType = "STR" Then
        objReg.GetStringValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, strValue

        ReadRegValue = strValue
    End If
End Function
```

La fonction `ReadRegValue` retourne la valeur de donnée du Registre qui correspond aux valeurs nommées de la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `strComputer`. Le nom ou l'adresse IP de l'ordinateur dont nous voulons examiner le Registre ; "." désigne la machine locale.
- `strKeyPath`. Le chemin de la clé qui contient la valeur du Registre.
- `strValueName`. Le nom de la valeur du Registre dont nous voulons obtenir les données.
- `strType`. Une chaîne qui représente le type de la valeur du Registre dont nous voulons obtenir les données, comme `BIN` (`REG_BINARY`), `DWORD` (`REG_DWORD`), `EXP` (`REG_EXPAND_SZ`), `MULTI` (`REG_MULTI_SZ`) et `STR` (`REG_SZ`).

En fonction de la valeur de `strType`, la fonction `ReadRegValue` invoque la méthode `StdRegProv` adéquate pour retrouver les données de la valeur du Registre indiquée. Les données retournées peuvent être sous la forme d'une chaîne de caractères, d'un entier ou d'un tableau. Elles doivent être traitées conformément au type de la valeur du Registre lue. Par exemple, si la valeur est de type `REG_BINARY`, les données retournées par `ReadRegValue` se trouvent dans un tableau qui contient des valeurs binaires. Pour les lire, nous devons parcourir le tableau de la manière suivante :

```
Set StdOut = WScript.StdOut
strServer = "serverxyz.companyabc.com"

binValue = ReadRegValue(strServer, "SOFTWARE\Turtle_Worm", "binValue", "BIN")

StdOut.WriteLine "Valeur BIN :"
For i = lBound(binValue) to uBound(binValue)
    StdOut.WriteLine binValue(i)
Next
```

La fonction `CreateRegKey` :

```
Function CreateRegKey(strComputer, strKeyPath)
    On Error Resume Next

    const HKEY_LOCAL_MACHINE = &H80000002

    Set objReg = GetObject("winmgmts:{impersonationLevel=impersonate}!\\" &_
        strComputer & "\root\default:StdRegProv")
```

```
    objReg.CreateKey HKEY_LOCAL_MACHINE, strKeyPath

End Function
```

La fonction `CreateRegKey` crée une clé dans la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `strComputer`. Le nom ou l'adresse IP de l'ordinateur sur lequel nous voulons créer la clé ; "." désigne la machine locale.
- `strKeyPath`. Le chemin de la nouvelle clé du Registre.

Voici un exemple d'utilisation de cette fonction :

```
strServer = "serverxyz.companyabc.com"

CreateRegKey strServer, "SOFTWARE\Turtle_Worm"
```

La fonction `CreateRegValue` :

```
Function CreateRegValue(strComputer, strKeyPath,_
                            strValueName, strvalue, strType)
    On Error Resume Next

    const HKEY_LOCAL_MACHINE = &H80000002

    Set objReg = GetObject("winmgmts:{impersonationLevel=impersonate}!\\" &_
        strComputer & "\root\default:StdRegProv")

    If strType = "BIN" Then
        objReg.SetBinaryValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, strValue
    End If
    If strType = "DWORD" Then
        objReg.SetDWORDValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, strValue
    End If
    If strType = "EXP" Then
        objReg.SetExpandedStringValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, strValue
```

```
    End If
    If strType = "MULTI" Then
        objReg.SetMultiStringValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, strValue
    End If
    If strType = "STR" Then
        objReg.SetStringValue HKEY_LOCAL_MACHINE, strKeyPath,_
            strValueName, strValue
    End If
End Function
```

La fonction `CreateRegValue` crée ou modifie une valeur du Registre dans la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `strComputer`. Le nom ou l'adresse IP de l'ordinateur sur lequel nous voulons créer ou modifier une valeur du Registre ; "." désigne la machine locale.
- `strKeyPath`. Le chemin de la clé qui contient la valeur du Registre.
- `strValueName`. Le nom de la valeur du Registre que nous voulons créer ou modifier.
- `strValue`. Le nouveau contenu de la valeur du Registre.
- `strType`. Une chaîne qui représente le type de la valeur du Registre que nous voulons créer ou modifier, comme `BIN` (`REG_BINARY`), `DWORD` (`REG_DWORD`), `EXP` (`REG_EXPAND_SZ`), `MULTI` (`REG_MULTI_SZ`) et `STR` (`REG_SZ`).

La valeur passée dans le paramètre `strValue` dépend du type de la valeur du Registre que nous créons ou modifions. Si la valeur est de type `REG_BINARY`, celle passée à la fonction `CreateRegValue` doit être un tableau contenant des valeurs binaires. Pour le type `REG_MULTI_SZ`, la valeur doit être un tableau contenant des chaînes de caractères. Pour `REG_SZ` et `REG_EXPAND_SZ`, les valeurs doivent prendre la forme d'une chaîne de caractères. Cependant, avec `REG_EXPAND_SZ`, elle doit également inclure une variable d'environnement valide. Dans le cas contraire, la méthode `GetExpandedStringValue` ne parviendra pas à développer la chaîne lors de l'obtention de la valeur. Enfin, lors de la création ou de la modification d'une valeur de type `REG_DWORD`, il faut que la valeur passée à `CreateRegValue` soit un `DWORD` valide.

Voici un exemple d'utilisation de cette fonction :

```
Set StdOut = WScript.StdOut
strServer = "serverxyz.companyabc.com"

Multi = Array("PowerShell", "est", "super !")
CreateRegValue strServer, "SOFTWARE\Turtle_Worm", "multiValue", Multi,_
"MULTI"
```

La fonction `DeleteRegKey` :

```
Function DeleteRegKey(strComputer, strKeyPath)
    On Error Resume Next

    const HKEY_LOCAL_MACHINE = &H80000002

    Set objReg = GetObject("winmgmts:{impersonationLevel=impersonate}!\\" &_
        strComputer & "\root\default:StdRegProv")

    objReg.DeleteKey HKEY_LOCAL_MACHINE, strKeyPath

End Function
```

La fonction `DeleteRegKey` supprime une clé dans la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `strComputer`. Le nom ou l'adresse IP de l'ordinateur sur lequel nous voulons supprimer la clé ; "." désigne la machine locale.
- `strKeyPath`. Le chemin de la clé du Registre à supprimer.

INFO

La suppression d'une clé supprime également toutes les sous-clés et leurs valeurs.

Voici un exemple d'utilisation de cette fonction :

```
Set StdOut = WScript.StdOut
strServer = "serverxyz.companyabc.com"

DeleteRegKey strServer, "SOFTWARE\Turtle_Worm"
```

La fonction `DeleteRegValue` :

```
Function DeleteRegValue(strComputer, strKeyPath, strValueName)
    On Error Resume Next

    const HKEY_LOCAL_MACHINE = &H80000002
```

```
    Set objReg = GetObject("winmgmts:{impersonationLevel=impersonate}!\\" &_
        strComputer & "\root\default:StdRegProv")

    objReg.DeleteValue HKEY_LOCAL_MACHINE, strKeyPath, strValueName

End Function
```

La fonction `DeleteRegValue` supprime une valeur dans la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `strComputer`. Le nom ou l'adresse IP de l'ordinateur sur lequel nous voulons supprimer une valeur du Registre ; "." désigne la machine locale.
- `strKeyPath`. Le chemin de la valeur du Registre.
- `strValueName`. Le nom de la valeur du Registre à supprimer.

Voici un exemple d'utilisation de cette fonction :

```
Set StdOut = WScript.StdOut
strServer = "server1000"

DeleteRegValue strServer, "SOFTWARE\Turtle_Worm", "binValue"
```

Le script LibraryRegistry.ps1

`LibraryRegistry.ps1` est la version PowerShell du fichier VBScript `LibraryRegistry.vbs`. Vous le trouverez dans le dossier `Scripts\Chapitre 7\LibraryRegistry` et en téléchargement depuis le site **www.pearsoneducation.fr**. Avant d'utiliser cette bibliothèque dans une session PowerShell, vous devez la charger comme nous l'avons expliqué au Chapitre 3. Le format de la commande point est un point suivi d'un espace, puis du nom du fichier. Par exemple, `. .\monScript.ps1`. Ainsi, pour charger `LibraryRegistry.ps1` dans une session PowerShell, saisissez la commande suivante :

```
PS C:\> . "D:\Scripts\LibraryRegistry.ps1"
```

Cependant, cette procédure de chargement d'un script chaque fois que nous voulons utiliser l'une de ses fonctions est vite fastidieuse. Lorsqu'un fichier de script est chargé avec la commande point, son contenu est placé dans la portée globale de la session PowerShell en cours. Tout ce qui se trouve dans la portée globale disparaît si nous fermons cette session et en ouvrons une nouvelle, et nous devons recharger le fichier de script à chaque nouvelle session.

Pour éviter ce problème, nous pouvons utiliser un profil PowerShell afin de définir la configuration de la console PowerShell. Grâce à un profil comme `Profile.ps1` et en utilisant la commande point dans ce profil, nous pouvons charger des fichiers de script dans la portée globale chaque fois que nous démarrons une nouvelle session de console. Voici un exemple de fichier `Profile.ps1` :

```
. "D:\Scripts\LibraryRegistry.ps1"


set-location C:\
cls

# Message de bienvenue.
"Bienvenue dans votre session PowerShell : " + $ENV:UserName
```

INFO

`LibraryRegistry.ps1` peut également être lu par la commande point dans un fichier de script. Dans ce cas, PowerShell charge le fichier dans la portée du script appelant. N'oubliez pas que la portée parente d'un script peut être une session PowerShell ou un autre script.

Une fois le profil `Profile.ps1` personnalisé chargé dans la session, l'invite de console ressemble à la suivante :

```
Bienvenue dans votre session PowerShell : script_master_snover
PS C:\>
```

En récupérant les informations fournies par l'objet `PSDrive Function`, comme le montre l'exemple suivant, nous pouvons déterminer si les fonctions du Registre définies dans `LibraryRegistry.ps1` ont été chargées dans la session PowerShell en cours :

```
PS C:\> get-childitem Function:

CommandType     Name                            Definition
-----------     ----                            ----------
Function        prompt                          'PS ' + $(Get-Location) + $(...
Function        TabExpansion                    ...
Function        Clear-Host                      $spaceType = [System.Managem...
```

```
Function        more                          param([string[]]$paths);  if...
Function        help                          param([string]$Name,[string[...
Function        man                           param([string]$Name,[string[...
Function        mkdir                         param([string[]]$paths); New...
Function        md                            param([string[]]$paths); New...
Function        A:                            Set-Location A:
Function        B:                            Set-Location B:
Function        C:                            Set-Location C:

...

Function        W:                            Set-Location W:
Function        X:                            Set-Location X:
Function        Y:                            Set-Location Y:
Function        Z:                            Set-Location Z:
Function        Get-RegValue                  param($Computer, $KeyPath, $...
Function        Set-RegKey                    param($Computer, $KeyPath) $...
Function        Set-RegValue                  param($Computer, $KeyPath, $...
Function        Remove-RegKey                 param($Computer, $KeyPath) $...
Function        Remove-RegValue               param($Computer, $KeyPath, $...


PS C:\>
```

L'exemple précédent montre que nos cinq fonctions de manipulation du Registre peuvent être employées dans la session PowerShell en cours. Nous allons à présent les examiner.

La fonction `Get-RegValue` :

```
#-----------------------------------------------
# Get-RegValue
#-----------------------------------------------
# Usage :        Lire une valeur dans la ruche HKLM, sur une machine locale
#                ou distante.
# $Computer :    Nom de l'ordinateur.
# $KeyPath :     Chemin de la clé du registre
#                ("SYSTEM\CurrentControlSet\Control").
# $ValueName :   Nom de la valeur ("CurrentUser").
# $Type :        Type de la valeur ("BIN", "DWORD", "EXP", "MULTI" ou "STR").

function Get-RegValue{
    param ($Computer, $KeyPath, $ValueName, $Type)
```

```
$HKEY_LOCAL_MACHINE = 2147483650

trap{write-host "[ERREUR] $_" -Foregroundcolor Red; Continue}

$Reg = get-wmiobject -Namespace Root\Default -computerName `
    $Computer -List | where-object `
    {$_.Name -eq "StdRegProv"}

if ($Type -eq "BIN"){
    return $Reg.GetBinaryValue($HKEY_LOCAL_MACHINE, $KeyPath, `
        $ValueName)
    }
elseif ($Type -eq "DWORD"){
    return $Reg.GetDWORDValue($HKEY_LOCAL_MACHINE, $KeyPath, `
        $ValueName)
    }
elseif ($Type -eq "EXP"){
    return $Reg.GetExpandedStringValue($HKEY_LOCAL_MACHINE, `
        $KeyPath, $ValueName)
    }
elseif ($Type -eq "MULTI"){
    return $Reg.GetMultiStringValue($HKEY_LOCAL_MACHINE, `
        $KeyPath, $ValueName)
    }
elseif ($Type -eq "STR"){
    return $Reg.GetStringValue($HKEY_LOCAL_MACHINE, `
        $KeyPath, $ValueName)
    }
}
```

La fonction `Get-RegValue` retrouve dans le Registre la valeur qui correspond aux valeurs nommées sous la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `$Computer`. Le nom ou l'adresse IP de l'ordinateur dont nous voulons examiner le Registre ; "." désigne la machine locale.
- `$KeyPath`. Le chemin de la clé qui contient la valeur du Registre.
- `$ValueName`. Le nom de la valeur du Registre dont nous voulons obtenir les données.
- `$Type`. Une chaîne qui représente le type de la valeur du Registre dont nous voulons obtenir les données, comme `BIN` (`REG_BINARY`), `DWORD` (`REG_DWORD`), `EXP` (`REG_EXPAND_SZ`), `MULTI` (`REG_MULTI_SZ`) et `STR` (`REG_SZ`).

L'exemple suivant montre comment utiliser cette fonction :

```
PS C:\> get-regvalue "Arus" "SOFTWARE\Voltron" "BlueLion" "BIN"
```

La fonction `Set-RegKey` :

```
#-----------------------------------------------
# Set-RegKey
#-----------------------------------------------
# Usage :        Créer ou fixer une clé dans la ruche HKLM sur une machine
#                locale ou distante.
# $Computer :    Nom de l'ordinateur.
# $KeyPath :     Chemin de la clé du registre
#                ("SYSTEM\CurrentControlSet\Control").

function Set-RegKey{
    param ($Computer, $KeyPath)

    $HKEY_LOCAL_MACHINE = 2147483650

    trap{write-host "[ERREUR] $_" -Foregroundcolor Red; Continue}

    $Reg = get-wmiobject -Namespace Root\Default -computerName `
        $Computer -List | where-object `
        {$_.Name -eq "StdRegProv"}

    return $Reg.CreateKey($HKEY_LOCAL_MACHINE, $KeyPath)
    }
```

La fonction `Set-RegKey` crée une clé du Registre dans la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `$Computer`. Le nom ou l'adresse IP de l'ordinateur sur lequel nous voulons créer la clé ; "." désigne la machine locale.
- `$KeyPath`. Le chemin de la nouvelle clé du Registre.

Voici un exemple d'utilisation de cette fonction :

```
PS C:\> set-regkey "Arus" "SOFTWARE\Voltron"
```

La fonction `Set-RegValue` :

```
#-----------------------------------------------
# Set-RegValue
#-----------------------------------------------
# Usage :        Créer ou fixer une valeur dans la ruche HKLM, sur une machine
#                locale ou distante.
# $Computer :    Nom de l'ordinateur.
# $KeyPath :     Chemin de la clé du registre
#                ("SYSTEM\CurrentControlSet\Control").
# $ValueName :   Nom de la valeur ("CurrentUser").
# $Value :       Nouvelle valeur ("valeur1", Array, Integer).
# $Type :        Type de la valeur ("BIN", "DWORD", "EXP", "MULTI" ou "STR").

function Set-RegValue{
    param ($Computer, $KeyPath, $ValueName, $Value, $Type)

    $HKEY_LOCAL_MACHINE = 2147483650

    trap{write-host "[ERREUR] $_" -Foregroundcolor Red; Continue}

    $Reg = get-wmiobject -Namespace Root\Default -computerName `
        $Computer -List | where-object `
        {$_.Name -eq "StdRegProv"}

    if ($Type -eq "BIN"){
        return $Reg.SetBinaryValue($HKEY_LOCAL_MACHINE, $KeyPath, `
            $ValueName, $Value)
        }
    elseif ($Type -eq "DWORD"){
        return $Reg.SetDWORDValue($HKEY_LOCAL_MACHINE, $KeyPath, `
            $ValueName, $Value)
        }
    elseif ($Type -eq "EXP"){
        return $Reg.SetExpandedStringValue($HKEY_LOCAL_MACHINE, `
            $KeyPath, $ValueName, $Value)
        }
    elseif ($Type -eq "MULTI"){
        return $Reg.SetMultiStringValue($HKEY_LOCAL_MACHINE, `
            $KeyPath, $ValueName, $Value)
        }
    elseif ($Type -eq "STR"){
        return $Reg.SetStringValue($HKEY_LOCAL_MACHINE, `
            $KeyPath, $ValueName, $Value)
        }
    }
```

La fonction `Set-RegValue` crée ou modifie une valeur du Registre dans la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `$Computer`. Le nom ou l'adresse IP de l'ordinateur sur lequel nous voulons créer ou modifier une valeur du Registre ; "." désigne la machine locale.
- `$KeyPath`. Le chemin de la clé qui contient la valeur du Registre.
- `$ValueName`. Le nom de la valeur du Registre que nous voulons créer ou modifier.
- `$Value`. Le nouveau contenu de la valeur du Registre.
- `$Type`. Une chaîne qui représente le type de la valeur du Registre dont nous voulons obtenir les données, comme `BIN` (`REG_BINARY`), `DWORD` (`REG_DWORD`), `EXP` (`REG_EXPAND_SZ`), `MULTI` (`REG_MULTI_SZ`) et `STR` (`REG_SZ`).

Voici comment utiliser cette fonction :

```
PS C:\> $Multi = "PowerShell", "est", "super !"
PS C:\> set-regvalue "Arus" "SOFTWARE\Voltron" "Lion_Statement" $Multi "MULTI"
```

La fonction `Remove-RegKey` :

```
#-----------------------------------------------
# Remove-RegKey
#-----------------------------------------------
# Usage :        Supprimer une clé dans la ruche HKLM, sur une machine
#                locale ou distante.
# $Computer :    Nom de l'ordinateur.
# $KeyPath :     Chemin de la clé du registre
#                ("SYSTEM\CurrentControlSet\Control").

function Remove-RegKey{
    param ($Computer, $KeyPath)

    $HKEY_LOCAL_MACHINE = 2147483650

    trap{write-host "[ERREUR] $_" -Foregroundcolor Red; Continue}

    $Reg = get-wmiobject -Namespace Root\Default -computerName `
        $Computer -List | where-object `
        {$_.Name -eq "StdRegProv"}

    return $Reg.DeleteKey($HKEY_LOCAL_MACHINE, $KeyPath)
    }
```

La fonction `Remove-RegKey` supprime une clé du Registre dans la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `$Computer`. Le nom ou l'adresse IP de l'ordinateur sur lequel nous voulons supprimer la clé ; "." désigne la machine locale.
- `$KeyPath`. Le chemin de la clé du Registre à supprimer.

Voici un exemple d'utilisation de cette fonction :

```
PS C:\> remove-regkey "Arus" "SOFTWARE\Voltron"
```

La fonction `Remove-RegValue` :

```
#-----------------------------------------------
# Remove-RegValue
#-----------------------------------------------
# Usage :        Supprimer une valeur dans la ruche HKLM, sur une machine
#                locale ou distante.
# $Computer :    Nom de l'ordinateur.
# $KeyPath :     Chemin de la clé du registre
#                ("SYSTEM\CurrentControlSet\Control").
# $ValueName :   Nom de la valeur ("CurrentUser").

function Remove-RegValue{
    param ($Computer, $KeyPath, $ValueName)

    $HKEY_LOCAL_MACHINE = 2147483650

    trap{write-host "[ERREUR] $_" -Foregroundcolor Red; Continue}

    $Reg = get-wmiobject -Namespace Root\Default -computerName `
        $Computer -List | where-object `
        {$_.Name -eq "StdRegProv"}

    return $Reg.DeleteValue($HKEY_LOCAL_MACHINE, $KeyPath, $ValueName)
    }
```

La fonction `Remove-RegValue` supprime une valeur du Registre dans la ruche `HKEY_LOCAL_MACHINE`. Elle attend les paramètres suivants :

- `$Computer`. Le nom ou l'adresse IP de l'ordinateur sur lequel nous voulons supprimer une valeur du Registre ; "." désigne la machine locale.
- `$KeyPath`. Le chemin de la clé qui contient la valeur du Registre.
- `$ValueName`. Le nom de la valeur du Registre que nous voulons supprimer.

Voici comment utiliser cette fonction :

```
PS C:\> remove-regvalue "Arus" "SOFTWARE\Voltron" "Lion_Statement"
```

Utiliser la bibliothèque

À présent que les fonctions de manipulation du Registre développées dans le script `LibraryRegistry.ps1` sont parfaitement comprises, nous pouvons les employer dans différentes situations. La première étape consiste à créer une clé de Registre nommée `Turtle_Worm` sous la clé `HKLM\Software` sur un contrôleur de domaine Active Directory appelé DC1. Pour cela, saisissons la commande suivante :

```
PS C:\> set-regkey "DC1" "SOFTWARE\Turtle_Worm"


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0



PS C:\>
```

La commande retourne un objet WMI qui ne contient aucune information. Si une erreur se produit, le gestionnaire défini dans la fonction affiche les informations concernant cette erreur, par exemple :

```
PS C:\> set-regkey "Pinky" "SOFTWARE\Turtle_Worm"
[ERREUR] Le serveur RPC n'est pas disponible. (Exception de HRESULT: 0x800706BA)
PS C:\>
```

Ensuite, nous invoquons les commandes suivantes pour créer des valeurs dans la clé de Registre `Turtle_Worm` :

```
PS C:\> $Bin = 101, 118, 105, 108, 95, 116, 117, 114, 116, 108, 101
PS C:\> set-regvalue „DC1" „SOFTWARE\Turtle_Worm" „binValue" $Bin „BIN"


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0



PS C:\> $Null = set-regvalue "DC1" "SOFTWARE\Turtle_Worm" "dwordValue" "1" "DWORD"
PS C:\> $Null = set-regvalue "DC1" "SOFTWARE\Turtle_Worm" "expValue" "%SystemRoot%\
system32\Turtle_Hacker.dll" "EXP"
PS C:\> $Multi = "PowerShell", "est", "super !"
PS C:\> $Null = set-regvalue "DC1" "SOFTWARE\Turtle_Worm" "multiValue" $Multi
"MULTI"
PS C:\> $Null = set-regvalue "DC1" "SOFTWARE\Turtle_Worm" "strValue" "Fin de la
modification du Registre !" "STR"
PS C:\>
```

Ces étapes illustrent la création d'une clé du registre et de ses valeurs. Ensuite, nous utilisons les fonctions de notre bibliothèque pour déterminer si certaines valeurs existent. Pour cela, nous invoquons la fonction `Get-RegValue` :

```
PS C:\> get-regvalue "DC1" "SOFTWARE\Turtle_Worm" "binValue" "BIN"



__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
```

```
__PROPERTY_COUNT : 2
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
uValue           : {101, 118, 105, 108...}



PS C:\> get-regvalue "DC1" "SOFTWARE\Turtle_Worm" "dwordValue" "DWORD"


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 2
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
uValue           : 1



PS C:\> get-regvalue "DC1" "SOFTWARE\Turtle_Worm" "expValue" "EXP"


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 2
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
sValue           : C:\WINDOWS\system32\Turtle_Hacker.dll
```

```
PS C:\> get-regvalue "DC1" "SOFTWARE\Turtle_Worm" "multiValue" "MULTI"


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 2
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
sValue           : {PowerShell, est, super !}



PS C:\> get-regvalue "DC1" "SOFTWARE\Turtle_Worm" "strValue" "STR"


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 2
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
sValue           : Fin de la modification du Registre !



PS C:\>
```

Comme vous pouvez le constater avec l'objet WMI retourné, si une valeur existe, ces informations sont données dans une propriété `sValue` ou `uValue`. Si la valeur ou la clé n'existe pas, la propriété `ReturnValue` a la valeur entière 2. Si cette propriété est égale à 0, cela signifie que la méthode WMI s'est parfaitement exécutée.

Nous avons vérifié que les valeurs existent sous la clé du registre `Turtle_Worm` sur la machine DC1. Il est temps à présent de supprimer cette clé et ses valeurs. Deux méthodes permettent d'effectuer cette tâche. Nous pouvons supprimer chaque valeur en utilisant `Remove-RegValue` :

```
PS C:\> remove-regvalue "DC1" "SOFTWARE\Turtle_Worm" "binValue"


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0


PS C:\>
```

Mais nous pouvons également invoquer `Remove-RegKey` pour supprimer la clé `Turtle_Worm` et, par voie de conséquence, toutes ses sous-clés et leurs valeurs :

```
PS C:\> remove-regkey "DC1" "SOFTWARE\Turtle_Worm"


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
```

```
__NAMESPACE      :
__PATH           :
ReturnValue      : 0


PS C:\>
```

En résumé

Ce chapitre s'est focalisé sur la gestion du Registre de Windows par le biais de l'utilisation de WSH et PowerShell. Bien que ces deux interfaces de script fournissent des méthodes de gestion du Registre, la version PowerShell est plus robuste car elle considère le Registre comme un magasin de données hiérarchique. Cependant, l'implémentation actuelle a pour inconvénient de ne proposer aucune méthode de gestion du Registre sur une machine distante (c'est également le cas de WSH). Pour contourner ce problème, nous avons combiné PowerShell et WMI afin d'accéder au Registre d'une machine distante. Grâce à WMI et à PowerShell, vous pourrez accomplir n'importe quelle tâche d'automation du Registre qui se présentera à vous.

Nous avons également abordé la réutilisation du code et les fichiers de bibliothèque. Comme nous l'avons expliqué au Chapitre 5, "Suivre les bonnes pratiques", la réutilisation du code est une pratique très importante qui permet de réduire le temps de développement d'un script. Ce chapitre a développé ce thème en montrant comment placer du code réutilisable, issu d'un exemple réel, dans un fichier de bibliothèque.

8

PowerShell et WMI

Dans ce chapitre

- Introduction
- Comparer l'utilisation de WMI dans WSH et dans PowerShell
- De VBScript à PowerShell

Introduction

Ce chapitre explique comment utiliser PowerShell pour la gestion des systèmes à l'aide de WMI (*Windows Management Instrumentation*) et compare les méthodes employées par WSH (*Windows Script Host*) et PowerShell pour mener à bien des tâches WMI. Nous examinons également quelques scripts qui utilisent WSH pour réaliser certaines tâches WMI, puis avec PowerShell. Enfin, nous présentons la conversion d'un script VBScript en PowerShell afin de mettre en œuvre une tâche d'automation fondée sur WMI. L'objectif est de donner au lecteur la possibilité d'appliquer les techniques de scripts PowerShell à des besoins d'automation réels.

Comparer l'utilisation de WMI dans WSH et dans PowerShell

Pour employer WMI dans des scripts, nous utilisons certains objets de l'API de WMI Scripting avec les méthodes WSH `CreateObject()` et `GetObject()` (ou les méthodes d'un autre langage de scripts qui permettent de créer ou de se lier à des objets COM). Nous pouvons ainsi nous lier à un objet WMI qui peut être une classe WMI ou une instance d'une classe WMI.

Il existe deux méthodes pour se connecter à un objet WMI. La première consiste à créer un objet `SWbemServices` à l'aide de la méthode `CreateObject()` correspondante, puis à se connecter à l'objet WMI en précisant son chemin. Cependant, dans notre description, nous nous limitons à la seconde méthode. Elle s'appuie sur le **moniker** "winmgmts:" (mécanisme COM standard pour encapsuler l'emplacement et la liaison avec un autre objet COM). Ces deux méthodes sont similaires, mais la première est souvent choisie pour des raisons liées à la gestion des erreurs et de l'authentification, tandis que la seconde est préférée pour des raisons pratiques car une seule instruction permet d'établir une connexion.

Utiliser WMI dans WSH

L'exemple de script VBScript suivant utilise un moniker qui crée une connexion à une machine distante et retourne ensuite la quantité de mémoire RAM installée sur cette machine :

```
On Error Resume Next

Dim objWMIService, objComputer, colItems
Dim strComputerName

strComputerName = "Jupiter"

Set objWMIService = GetObject("winmgmts:\\" & strComputerName _
        & "\root\cimv2")

Set colItems = objWMIService.ExecQuery _
    ("Select * from Win32_ComputerSystem")

For Each objItem in colItems
    WScript.Echo "Taille de la mémoire RAM : " _
        & FormatNumber((objItem.TotalPhysicalMemory \ 1024) _
        \ 1000, 0, 0, 0, -1) & " Mo"
Next
```

Si nous enregistrons ce script dans le fichier `ObtenirMemoire.vbs` et l'exécutons avec `cscript`, nous obtenons les résultats suivants :

```
C:\>cscript ObtenirMemoire.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

Taille de la mémoire RAM : 774 Mo

C:\>
```

Les sections suivantes reviennent pas à pas sur ce script et expliquent comment il obtient des informations concernant la mémoire installée sur la machine Jupiter.

Étape 1

Tout d'abord, nous nous connectons à l'objet de service WMI dans l'espace de noms `root\cimv2` sur la machine Jupiter :

```
Set objWMIService = GetObject("winmgmts:\\" & strComputerName _
& "\root\cimv2")
```

Étape 2

Ensuite, nous invoquons la méthode `ExecQuery()` de l'objet de service WMI en utilisant WQL (*WMI Query Language*) pour créer un objet lié à une instance de la classe `Win32_ComputerSytem` :

```
Set colItems = objWMIService.ExecQuery _
    ("Select * from Win32_ComputerSystem")
```

Étape 3

Enfin, à l'aide de la variable `colItems` et d'une boucle `for`, nous parcourons la nouvelle collection d'objets créée et obtenons les informations recherchées à partir de la propriété `TotalPhysicalMemory`. Nous mettons en forme la valeur numérique en appelant la fonction `FormatNumber`, puis nous affichons le résultat (en mégaoctets) sur la console :

```
For Each objItem in colItems
    WScript.Echo "Taille de la mémoire RAM : " _
        & FormatNumber((objItem.TotalPhysicalMemory \ 1024) _
        \ 1000, 0, 0, 0, -1) & " Mo"
Next
```

Utiliser WMI dans PowerShell

L'utilisation de WMI dans PowerShell présente une logique conceptuelle analogue à la précédente. La différence principale est liée au fait que les méthodes de PowerShell s'appuient sur WMI .NET au lieu de l'API de WMI Scripting. Dans PowerShell, il existe trois manières d'employer WMI : WMI .NET (les espaces de noms `System.Management` et `System.Management.Instrumentation` de .NET), l'applet de commande `Get-WmiObject` et les types WMI abrégés de PowerShell (`[WMI]`, `[WMIClass]` et `[WMISearcher]`).

La première méthode, c'est-à-dire les espaces de noms `System.Management` et `System.Management.Instrumentation`, n'est pas décrite ici car elle n'est pas aussi simple que les deux autres. Elle doit servir de solution de repli lorsque PowerShell n'encapsule pas correctement un objet dans un objet `PSObject` nécessaire aux deux autres méthodes.

La deuxième méthode, l'applet de commande `Get-WmiObject`, obtient des objets WMI et réunit des informations à propos des classes WMI. Cette applet est relativement simple. Par exemple, pour obtenir une instance locale de la classe `Win32_ComputerSystem`, il suffit d'indiquer son nom :

```
PS C:\> get-wmiobject "Win32_ComputerSystem"

Domain              : companyabc.com
Manufacturer        : Hewlett-Packard
Model               : Pavilion dv8000 (ES184AV)
Name                : Wii
PrimaryOwnerName    : Damon Cortesi
TotalPhysicalMemory : 2145566720


PS C:\>
```

L'exemple suivant, qui est plus robuste, se connecte à la machine distante Jupiter et obtient une instance de la classe `Win32_Service` dont le nom d'instance est Virtual Server. Le résultat est un objet contenant des informations à propos du service Virtual Server sur Jupiter :

```
PS C:\> get-wmiobject -class "Win32_Service" -computerName "Jupiter" -filter
"Name='Virtual Server'"



ExitCode  : 0
Name      : Virtual Server
ProcessId : 656
StartMode : Auto
State     : Running
Status    : OK


PS C:\>
```

La commande suivante retourne les mêmes informations, mais à partir d'une requête WQL :

```
PS C:\> get-wmiobject -computerName "Jupiter" -query "Select * From Win32_
Service Where Name='Virtual Server'"


ExitCode  : 0
Name      : Virtual Server
ProcessId : 656
StartMode : Auto
State     : Running
Status    : OK


PS C:\>
```

Enfin, voici comment employer `Get-WmiObject` pour obtenir des informations à propos d'une classe WMI :

```
PS C:\> get-wmiobject -namespace "root/cimv2" -list | where {$_.Name -eq
"Win32_Product"} | format-list *


Name             : Win32_Product
__GENUS          : 1
__CLASS          : Win32_Product
__SUPERCLASS     : CIM_Product
__DYNASTY        : CIM_Product
__RELPATH        : Win32_Product
__PROPERTY_COUNT : 12
__DERIVATION     : {CIM_Product}
__SERVER         : PLANX
__NAMESPACE      : ROOT\cimv2
__PATH           : \\PLANX\ROOT\cimv2:Win32_Product


PS C:\>
```

Même si l'utilisation de `Get-WmiObject` reste simple, il est souvent nécessaire de saisir une longue chaîne de commande. Cet inconvénient nous amène à la troisième méthode, les types WMI abrégés. La section suivante présente les abréviations de types et l'utilisation des types WMI abrégés de PowerShell.

Abréviations de types

Nous avons déjà employé les abréviations de types au cours des chapitres précédents, mais elles n'ont pas encore été véritablement expliquées. Une **abréviation de type** n'est rien d'autre qu'un alias permettant d'indiquer un type .NET. Sans les abréviations de types, la définition du type d'une variable impose la saisie du nom de classe complet :

```
PS C:\> $Utilisateur = [System.DirectoryServices.DirectoryEntry]"LDAP://
CN=Fujio Saitoh,OU=Accounts,OU=Managed Objects,DC=companyabc,DC=com"
PS C:\> $Utilisateur

distinguishedName
-----------------
{CN=Fujio Saitoh,OU=Accounts,OU=Managed Objects,DC=companyabc,DC=com}



PS C:\>
```

Au lieu de saisir l'intégralité de ce nom, nous pouvons simplement employer le type abrégé `[ADSI]` pour définir le type de la variable :

```
PS C:\> $Utilisateur = [ADSI]"LDAP://CN=Fujio Saitoh,OU=Accounts,OU=Managed
Objects,DC=companyabc,DC=com"
PS C:\> $Utilisateur

distinguishedName
-----------------
{CN=Fujio Saitoh,OU=Accounts,OU=Managed Objects,DC=companyabc,DC=com}



PS C:\>
```

L'équipe de développement de PowerShell a inclus les abréviations de types principalement pour réduire la saisie nécessaire à la définition d'un type d'objet. Cependant, les types abrégés ne sont pas décrits dans la documentation de PowerShell, même si on retrouve des références à `[WMI]`, `[ADSI]` et d'autres sur de nombreux blogs.

Malgré leur absence dans la documentation, l'abréviation de types est une caractéristique plutôt utile de PowerShell. Le Tableau 8.1 donne la liste des types abrégés les plus employés.

Tableau 8.1 Types abrégés dans PowerShell

Nom	Type
[int]	typeof(int)
[int[]]	typeof(int[])
[long]	typeof(long)
[long[]]	typeof(long[])
[string]	typeof(string)
[string[]]	typeof(string[])
[char]	typeof(char)
[char[]]	typeof(char[])
[bool]	typeof(bool)
[bool[]]	typeof(bool[])
[byte]	typeof(byte)
[double]	typeof(double)
[decimal]	typeof(decimal)
[float]	typeof(float)
[single]	typeof(float)
[regex]	typeof(System.Text.RegularExpressions.Regex)
[array]	typeof(System.Array)
[xml]	typeof(System.Xml.XmlDocument)
[scriptblock]	typeof(System.Management.Automation.ScriptBlock)
[switch]	typeof(System.Management.Automation.SwitchParameter)
[hashtable]	typeof(System.Collections.Hashtable)
[type]	typeof(System.Type)
[ref]	typeof(System.Management.Automation.PSReference)
[psobject]	typeof(System.Management.Automation.PSObject)
[wmi]	typeof(System.Management.ManagementObject)
[wmisearcher]	typeof(System.Management.ManagementObjectSearcher)
[wmiclass]	typeof(System.Management.ManagementClass)
[adsi]	typeof(System.DirectoryServices.DirectoryEntry)

Les sections suivantes expliquent comment utiliser les types WMI abrégés de PowerShell.

Le type abrégé [WMI]

Ce type abrégé pour la classe `ManagementObject` prend un chemin d'objet WMI sous forme d'une chaîne et obtient un objet WMI lié à une instance de la classe WMI indiquée :

```
PS C:\> $InfoOrdi = [WMI]"\\.\root\cimv2:Win32_ComputerSystem.Name='PLANX'"
PS C:\> $InfoOrdi


Domain              : companyabc.com
Manufacturer        : Hewlett-Packard
Model               : Pavilion dv8000 (ES184AV)
Name                : PLANX
PrimaryOwnerName    : Frank Miller
TotalPhysicalMemory : 2145566720


PS C:\>
```

INFO

Pour établir une liaison directement à un objet WMI, vous devez inclure la propriété clé dans le chemin de l'objet WMI. À l'exemple précédent, la propriété clé est `Name`.

Le type abrégé [WMIClass]

Ce type abrégé pour la classe `ManagementClass` prend un chemin d'objet WMI sous forme d'une chaîne et obtient un objet WMI lié à la classe WMI indiquée :

```
PS C:\> $ClasseOrdi = [WMICLASS]"\\.\root\cimv2:Win32_ComputerSystem"
PS C:\> $ClasseOrdi


Win32_ComputerSystem

PS C:\> $ClasseOrdi | format-list *


Name             : Win32_ComputerSystem
__GENUS          : 1
__CLASS          : Win32_ComputerSystem
```

```
__SUPERCLASS     : CIM_UnitaryComputerSystem
__DYNASTY        : CIM_ManagedSystemElement
__RELPATH        : Win32_ComputerSystem
__PROPERTY_COUNT : 54
__DERIVATION     : {CIM_UnitaryComputerSystem, CIM_ComputerSystem, CIM_System,
                   CIM_LogicalElement...}
__SERVER         : PLANX
__NAMESPACE      : ROOT\cimv2
__PATH           : \\PLANX\ROOT\cimv2:Win32_ComputerSystem


PS C:\>
```

Le type abrégé [WMISearcher]

Ce type abrégé pour la classe `ManagementObjectSearcher` prend une chaîne WQL et crée un objet de recherches WMI. Ensuite, nous pouvons appeler la méthode `Get()` pour obtenir un objet WMI lié à une instance de la classe WMI indiquée :

```
PS C:\> $ClasseOrdi = [WMISearcher]"Select * From Win32_ComputerSystem"
PS C:\> $ClasseOrdi.Get()


Domain              : companyabc.com
Manufacturer        : Hewlett-Packard
Model               : Pavilion dv8000 (ES184AV)
Name                : PLANX
PrimaryOwnerName    : Miro
TotalPhysicalMemory : 2145566720


PS C:\
```

De VBScript à PowerShell

Cette section détaille la conversion d'un script VBScript en son équivalent PowerShell. Ce script est utilisé pour surveiller des machines virtuelles sur un hôte Microsoft Virtual Server 2005.

Avant le développement de ce script, la société companyabc.com était en cours de remplacement de la plupart de ses serveurs d'applications matériels par des machines virtuelles.

Ce basculement devait également inclure une méthode simple mais efficace pour surveiller les machines virtuelles hébergées par chaque Microsoft Virtual Server. Cependant, une plateforme de surveillance réelle, comme MOM (*Microsoft Operations Manager*), n'était pas en place. Le service informatique a suggéré de développer un script d'automation qui remplirait les besoins de supervision à court terme de l'entreprise, et c'est ce qui a été réalisé.

Le script MonitorMSVS.wsf

`MonitorMSVS.wsf` est un fichier VBScript développé pour répondre aux besoins de companyabc.com quant à la surveillance d'une machine virtuelle. Vous le trouverez dans le dossier `Scripts\Chapitre 8\MonitorMSVS` et vous pouvez le télécharger depuis le site **www.pearsoneducation.fr**. Pour l'exécuter, il faut définir le paramètre `servername`, dont l'argument indique le nom du système Virtual Server qui héberge les machines virtuelles à superviser. Voici la commande qui permet d'exécuter `MonitorMSVS.wsf`, dont la sortie est illustrée à la Figure 8.1 :

```
D:\Scripts>cscript MonitorMSVS.wsf /servername:vsserver01
```

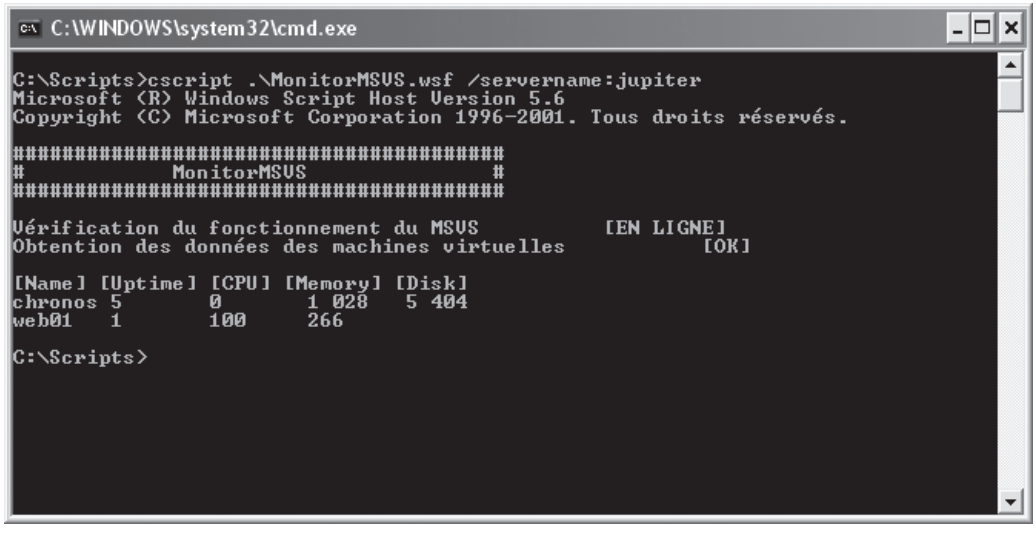


Figure 8.1

Exécution du script MonitorMSVS.wsf.

Voici la suite des opérations réalisées par `MonitorMSVS.wsf` :

1. Le script envoie un ping vers le MSVS (*Microsoft Virtual Server*) indiqué afin de vérifier que le serveur est opérationnel.
2. Puis, il se connecte à l'hôte MSVS en utilisant une chaîne de moniker et, par conséquent, en créant un objet de service WMI.
3. Ensuite, il invoque la méthode `ExecQuery()` de l'objet de service WMI, en lui passant une requête WQL qui demande une collection d'instances de la classe `VirtualMachine`.
4. Enfin, pour chaque machine virtuelle active (présente dans la collection), le script écrit sur la console les valeurs actuelles des propriétés `Uptime`, `CpuUtilization`, `PhysicalMemoryAllocated` et `DiskSpaceUsed`.

Le premier extrait de code est constitué des éléments XML initiaux pour un fichier WSF. Ils définissent les paramètres acceptés, décrivent le script, donnent des exemples de fonctionnement du script et précisent le langage employé :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<package>
<job id="MonitorMSVS">
    <runtime>
        <description>
************************************************************
Ce script permet de surveiller Microsoft Virtual Server 2005.
************************************************************
        </description>
        <named name="servername" helpstring="Nom de l'hôte MSVS à
surveiller." type="string" required="1" />
        <example>
Exemple :
cscript MonitorMSVS.wsf /servername:"vms01.companyabc.com"
        </example>
    </runtime>
<script language="VBScript">
<![CDATA[
```

Le script vérifie ensuite si un argument a été défini pour le paramètre obligatoire `servername`. Si ce n'est pas le cas, il affiche les informations d'utilisation (définies dans le code précédent)

sur la console et se termine. Lorsque l'argument est précisé, le script configure son environnement en définissant les variables utilisées par la suite :

```
On Error Resume Next

'====================================================================
' Vérifier les arguments obligatoires.
'====================================================================
If WScript.Arguments.Named.Exists("servername") = FALSE Then
    WScript.Arguments.ShowUsage()
    WScript.Quit
End If

'====================================================================
' Définir l'environnement de travail.
'====================================================================
Dim StdOut
Dim strServerName

Set StdOut = WScript.StdOut

strServerName = WScript.Arguments.Named("servername")
```

Le prochain extrait montre le début du code d'automation. Tout d'abord, le script affiche son en-tête sur la console, puis vérifie que l'hôte MSVS indiqué par `servername` est opérationnel. Pour cela, il tente de le contacter à l'aide de la fonction `Ping`. Si l'hôte MSVS fonctionne parfaitement, le script se poursuit. Sinon, il s'arrête et le message d'état adéquat est affiché à l'opérateur :

```
'====================================================================
' Commencer le travail.
'====================================================================
Mess "#########################################"
Mess "#            MonitorMSVS               #"
Mess "#########################################"
Mess vbNullString

StatStart "Vérification du fonctionnement du MSVS"
```

```
If Ping(strServerName) = 0 Then
    StdOut.Write(vbTab & vbTab)
    StdOut.WriteLine("[HORS LIGNE]")
    WScript.Quit()
Else
    StdOut.Write(vbTab & vbTab)
    StdOut.WriteLine("[EN LIGNE]")
End If
```

L'étape suivante consiste à établir une connexion avec l'hôte MSVS en utilisant WMI et à obtenir des informations de performance sur ses machines virtuelles. Une fois cela fait, les informations doivent être converties en un format lisible avant d'être affichées sur la console :

```
'--------------------
' Obtenir les données concernant les machines virtuelles.
'--------------------
StatStart "Obtention des données des machines virtuelles"
    Set objWMIService = GetObject("winmgmts:\\" & strServerName _
        & "\root\vm\virtualserver")
    Set colItems = objWMIService.ExecQuery("SELECT * FROM VirtualMachine")

    Xerror
StatDone
StdOut.WriteLine(vbNullString)

' Affichage de l'en-tête.
StdOut.WriteLine("[Name] [Uptime] [CPU] [Memory] [Disk]")

For Each objItem In colItems
    StdOut.Write(objItem.Name & vbTab)
    StdOut.Write(FormatNumber(objItem.Uptime / 60, 0, 0, 0, -1) & vbTab)
    StdOut.Write(FormatNumber(objItem.CpuUtilization, 0) & vbTab)
    StdOut.Write(FormatNumber((objItem.PhysicalMemoryAllocated  _
            / 1024) / 1000, 0, 0, 0, -1) & vbTab)
    StdOut.Write(FormatNumber((objItem.DiskSpaceUsed / 1024) _
            / 1000, 0, 0, 0, -1))
    StdOut.WriteLine(vbNullString)
Next
```

Pour que les valeurs données par les propriétés `Uptime`, `CpuUtilization`, `PhysicalMemoryAllocated` et `DiskSpaceUsed` soient plus lisibles, le script invoque la fonction `FormatNumber`. Elle détermine la mise en forme d'une valeur numérique et permet de préciser les paramètres suivants :

- nombre de chiffres affichés après la virgule ;
- afficher ou non un 0 initial pour les valeurs fractionnaires ;
- entourer ou non les valeurs négatives par des parenthèses ;
- regrouper ou non les nombres en utilisant le séparateur des milliers défini dans le Panneau de configuration.

`MonitorMSVS.wsf` utilise la fonction `FormatNumber` de manière à formater les valeurs numériques sans chiffres après la virgule et avec le séparateur des milliers qui correspond aux paramètres régionaux de la machine. Enfin, ces valeurs sont converties en unités plus représentatives :

- `Uptime`, initialement en secondes, est affiché en minutes.
- `PhysicalMemoryAllocated` n'est plus en octets mais en mégaoctets.
- `DiskSpaceUsed` passe également en mégaoctets.

L'extrait de code suivant est constitué des procédures utilisées tout au long du script :

```
'====================================================================
' Procédures.
'====================================================================
'--------------------
' Procédure générale pour les messages.
'--------------------
Sub Mess(Message)
    ' Écrire sur la console.
    StdOut.WriteLine(Message)
End Sub

'--------------------
' Procédure générale pour le début d'un message.
'--------------------
Sub StatStart(Message)
    ' Écrire sur la console.
    StdOut.Write(Message)
End Sub
```

```
'--------------------
' Procédure générale pour la fin d'un message.
'--------------------
Sub StatDone
    ' Écrire sur la console.
    StdOut.Write(vbTab & vbTab)
    StdOut.WriteLine("[OK]")
End Sub

'--------------------
' Procédure générale pour Xerror.
'--------------------
Sub Xerror
    If Err.Number <> 0 Then
        ' Écrire sur la console.
        StdOut.WriteLine(" Erreur fatale : " & CStr(Err.Number) _
            & " " & Err.Description)

        WScript.Quit()
    End If
End Sub
```

Le script `MonitorMSVS.wsf` doit s'assurer que l'hôte MSVS est opérationnel avant de pouvoir continuer. Ce contrôle est effectué à l'aide d'un ping ICMP :

```
'====================================================================
' Fonctions.
'====================================================================
'--------------------
' Envoyer un ping à une machine.
'--------------------
' Cette fonction teste si une machine est connectée au réseau.
Function Ping(Machine)
    On Error Resume Next

    Set colItems = GetObject("winmgmts:{impersonationLevel=impersonate}")._
        ExecQuery("select * from Win32_PingStatus where address = '"_
        & Machine & "'")

    For Each colItem in colItems
        If IsNull(colItem.StatusCode) or colItem.StatusCode <> 0 Then
```

```
            Ping = 0
        Else
            Ping = 1
        End If
    Next
End Function
```

Pour mettre en œuvre le ping ICMP, le script utilise une fonction nommée, fort à propos, `Ping`. Elle réalise la séquence d'opérations suivante :

1. La fonction `Ping` appelle la méthode `ExecQuery()` de l'objet de service WMI.
2. Elle passe à `ExecQuery()` une requête WQL qui demande toutes les propriétés de l'instance de la classe `Win32_PingStatus`. L'adresse indiquée est celle de l'hôte que nous tentons de contacter.
3. La collection d'instances obtenue (dans ce cas, une seule instance, qui n'est qu'un objet) est affectée à la variable `colItems`.
4. Le résultat du ping est pris dans `colItems` et retourné au script afin que celui-ci détermine s'il peut ou non poursuivre son exécution.

Grâce au ping ICMP, nous diminuons le temps demandé par le script pour échouer si le serveur interrogé n'était pas capable de répondre. Cette gestion élaborée des erreurs permet de prévoir l'échec du script et inclut une logique qui empêche cet échec. De plus, nous utilisons une méthode WMI à la place de `ping.exe` car les résultats retournés par WMI sont plus faciles à manipuler que ceux de cette commande.

Le dernier exemple de code est constitué des éléments XML qui terminent le script :

```
]]>
  </script>
</job>
</package>
```

Le script MonitorMSVS.ps1

`MonitorMSVS.ps1` est la version PowerShell du script `MonitorMSVS.wsf`. Vous le trouverez dans le dossier `Scripts\Chapitre 8\MonitorMSVS` et en téléchargement sur le site **www.pearson-education.fr**. Pour l'exécuter, il faut définir le paramètre `ServerName`, dont l'argument doit être le nom du système Virtual Server qui héberge les machines virtuelles à surveiller.

Voici la commande qui permet de lancer `MonitorMSVS.ps1`, ainsi qu'un exemple de sortie à la Figure 8.2 :

```
PS D:\Scripts> .\MonitorMSVS.ps1 -ServerName Jupiter
```

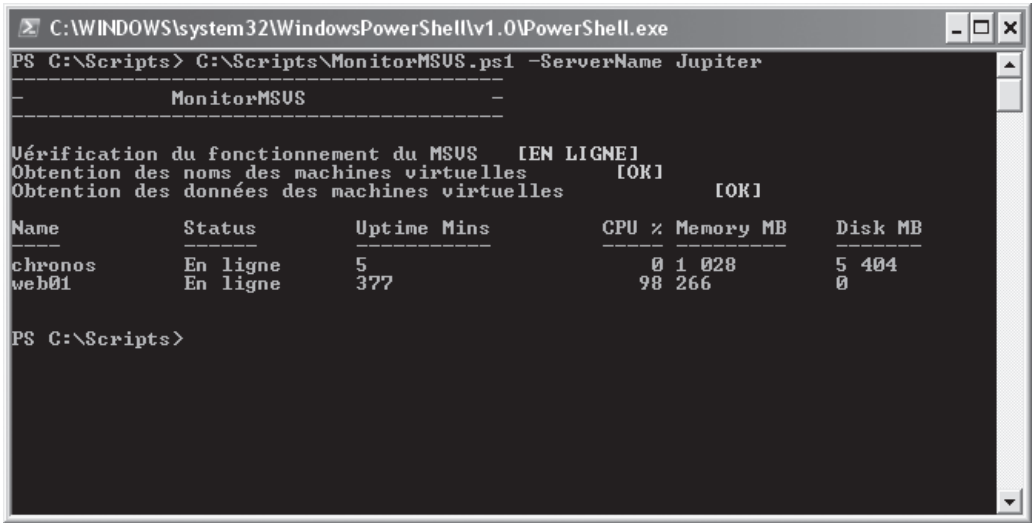


Figure 8.2

Exécution du script MonitorMSVS.ps1.

INFO

Dans la commande d'exécution du script `MonitorMSVS.ps1`, le nom du paramètre `ServerName` est indiqué dans la chaîne de commande, contrairement à l'exemple du Chapitre 6, "PowerShell et le système de fichiers". Dans PowerShell, vous pouvez passer le nom complet ou partiel des paramètres :

```
\MonitorMSVS.ps1 -S Jupiter
```

Si les arguments sont définis dans l'ordre exact des paramètres du script, il n'est pas nécessaire de les indiquer sur la ligne d'exécution du script :

```
\MonitorMSVS.ps1 Jupiter
```

Voici les opérations effectuées par le script `MonitorMSVS.ps1` :

1. Le script envoie un ping vers le MSVS (*Microsoft Virtual Server*) indiqué afin de vérifier que le serveur est opérationnel.
2. Puis, il se connecte au site Web d'administration de MSVS afin d'obtenir une liste des machines virtuelles hébergées par cet hôte. Cette liste est placée dans la variable `$Servers`.
3. Il utilise l'applet de commande `Get-WmiObject` pour obtenir une collection d'instances de la classe `VirtualMachine`, qu'il place dans la variable `$VirtualMachines`.
4. Pour chaque objet de machine virtuelle présent dans la variable `$Servers`, il ajoute l'état actuel de la machine virtuelle comme membre de cet objet. Si la machine virtuelle est active (présente dans la collection `$VirtualMachines`), le script ajoute également les valeurs des propriétés `Uptime`, `CpuUtilization`, `PhysicalMemoryAllocated` et `DiskSpaceUsed` comme membres de l'objet de machine virtuelle.
5. Enfin, il affiche les informations sur la console de PowerShell en utilisant l'applet de commande `Format-Table`.

Le premier extrait de code contient l'en-tête du script `MonitorMSVS.ps1`. Il fournit des informations sur le rôle du script, sa date de dernière mise à jour et son auteur. Juste après l'en-tête, nous trouvons l'unique paramètre du script (`$ServerName`) :

```
##################################################
# MonitorMSVS.ps1
# Ce script surveille Microsoft Virtual Server 2005.
#
# Créé le : 01/12/2006
# Auteur : Tyson Kopczynski
##################################################
param([string] $ServerName = $(throw write-host `
    "Veuillez indiquer le nom de l'hôte MSVS à surveiller !" `
    -Foregroundcolor Red))
```

Le code suivant contient le début de la partie automation du script. Tout d'abord, la variable `$URL` se voit affecter l'URL du site Web d'administration de Virtual Server pour l'hôte MSVS. Ensuite, comme le script `MonitorMSVS.wsf`, `MonitorMSVS.ps1` se sert d'un ping ICMP pour vérifier que l'hôte MSVS est opérationnel. Cependant, il utilise pour cela la classe .NET `Net.NetworkInformation.Ping` à la place de WMI. D'autres méthodes, y compris `ping.exe`,

auraient pu être choisies, mais la classe `Net.NetworkInformation.Ping` demande moins de travail et de code. Quelle que soit la méthode retenue, l'important est de prévoir les possibilités d'échec du script et de gérer correctement les erreurs :

```
##################################################
# Code principal.
##################################################
$URL = "http://$($ServerName):1024/VirtualServer/VSWebApp.exe?view=1"

#--------------------
# Début du script.
#--------------------
write-host "----------------------------------------"
write-host "-            MonitorMSVS               -"
write-host "----------------------------------------"
write-host
write-host "Vérification du fonctionnement du MSVS" -NoNewLine

.{
    trap{write-host `t "[ERREUR]" -Foregroundcolor Red;
        throw write-host $_ -Foregroundcolor Red;
        Break}

    $Ping = new-object Net.NetworkInformation.Ping
    $Result = $Ping.Send($ServerName)

    if ($Result.Status -eq "Success"){
        write-host `t "[EN LIGNE]" -Foregroundcolor Green
        }
    else{
        write-host `t "[HORS LIGNE]" -Foregroundcolor Red
        write-host
        Break
        }
}
```

Si l'hôte MSVS fonctionne parfaitement, le script affiche "[EN LIGNE]" sur la console et poursuit son exécution. En revanche, s'il n'est pas opérationnel, il affiche "[HORS LIGNE]" et se termine.

Après la vérification du fonctionnement de l'hôte MSVS, l'étape suivante consiste à s'y connecter et à obtenir une liste des machines virtuelles qu'il héberge. L'extrait de code suivant réalise cette tâche en améliorant la logique du script `MonitorMSVS.wsf` originel et en illustrant l'une des possibilités les plus impressionnantes de PowerShell :

```
#--------------------
# Obtenir la liste des machines virtuelles.
#--------------------
$Webclient = new-object Net.WebClient
$Webclient.UseDefaultCredentials = $True

write-host "Obtention des noms des machines virtuelles" -NoNewLine

.{
    trap{write-host `t "[ERREUR]" -Foregroundcolor Red;
        throw write-host $_ -Foregroundcolor Red;
        Break}

    $Data = $Webclient.DownloadString("$URL")

    write-host `t "[OK]" -Foregroundcolor Green
}

# Cette expression régulière obtient une liste des entrées de serveur
# à partir de données reçues.
$Servers = [Regex]::Matches($Data, '(?<=&amp;vm=)[^"\r\n]*(?=" )')

# Les doublons sont nombreux et doivent donc être regroupés.
# Par ailleurs, cela donne un meilleur nom à la propriété.
$Servers = $Servers | group Value | select Name
```

Le script `MonitorMSVS.wsf` présente un inconvénient majeur : la requête WMI retourne des informations uniquement sur les machines virtuelles qui sont actives au moment de la demande. Si une machine virtuelle est arrêtée à ce moment-là, il est impossible d'afficher cet état aux utilisateurs. Pourtant, disposer de la liste complète des machines virtuelles et de leur état courant est une information utile pour un outil de surveillance.

Pour accéder à ces informations, le script doit créer une liste de toutes les machines virtuelles présentes sur l'hôte MSVS. Cette liste existe sur le site Web d'administration de Microsoft Virtual Server. Pour l'obtenir, le script utilise la classe .NET `Net.WebClient`, grâce à laquelle il se connecte au site d'administration et télécharge le contenu HTML depuis la page d'état principale.

INFO

Puisque PowerShell est compatible avec .NET Framework, il peut accéder aux services Web et s'en servir comme source de données externe ou comme applications. Par exemple, PowerShell peut être employé pour poster des billets sur des blogs ou lire leur contenu, pour vérifier la disponibilité des consoles Wii sur amazon.fr ou pour effectuer d'autres tâches d'automation s'appuyant sur des données ou des applications fournies par les services Web de votre entreprise. Les possibilités sont infinies.

Dans le contenu HTML téléchargé, le nom de chaque machine virtuelle est répété plusieurs fois. Pour construire la liste, le script se sert du type abrégé des expressions régulières, `[Regex]`, afin d'extraire ces noms et de les placer dans la variable `$Servers`. La liste contient donc le nom de chaque machine virtuelle, mais répété plusieurs fois. Pour ne garder qu'un exemplaire de chaque nom, le script invoque l'applet de commande `Group-Object`. La liste finale, qui contient les noms de toutes les machines virtuelles hébergées par l'hôte MSVS indiqué, est réaffectée à la variable `$Servers`.

Ensuite, le script récupère les informations de performances des machines virtuelles à partir d'instances de la classe WMI `VirtualMachine` obtenues à l'aide de l'applet de commande `Get-WmiOjbect`. Dans l'étape suivante, les deux jeux de données sont fusionnés : les informations concernant les machines virtuelles (`$VirtualMachines`) et la liste des machines virtuelles (`$Servers`). Pour cela, le script prend chaque objet de machine virtuelle contenu dans la variable `$Servers`. Si le nom de la machine virtuelle se trouve dans les deux collections d'objets, l'applet de commande `Add-Member` est invoquée afin de compléter l'objet de machine virtuelle courant avec les informations de performances données par la variable `$VirtualMachines`.

Cette extension de l'objet inclut un indicateur d'état d'activité et les informations de propriétés associées. Si la machine virtuelle est hors ligne (absente des deux collections), le script insère uniquement l'indicateur d'état d'inactivité. Le concept de modification dynamique d'un objet a été présenté au Chapitre 3, "Présentation avancée de PowerShell", mais cet exemple illustre la puissance de cette caractéristique dans un script d'automation.

Voici le code de toute cette procédure :

```
#--------------------
# Obtenir les données concernant les machines virtuelles.
#--------------------
write-host "Obtention des données des machines virtuelles" -NoNewLine

.{
    trap{write-host `t`t "[ERREUR]" -Foregroundcolor Red;
        throw write-host $_ -Foregroundcolor Red;
        Break}

    $VSMachines = get-wmiobject -namespace "root/vm/virtualserver" `
        -class VirtualMachine -computername $ServerName
        -ErrorAction Stop

    write-host `t`t "[OK]" -Foregroundcolor Green
}

foreach ($Server in $Servers){
    &{
        $VSMachine = $VSMachines | where {$_.Name -eq $Server.Name}

        if($VSMachine){
            $Uptime = $VSMachine.Uptime / 60
            $Memory =  ($VSMachine.PhysicalMemoryAllocated / 1024) / 1000
            $Disk = ($VSMachine.DiskSpaceUsed / 1024) /1000

            add-member -inputObject $Server -membertype noteProperty `
                -name "Status" -value "En ligne"
            add-member -inputObject $Server -membertype noteProperty `
                -name "Uptime" -value $Uptime
            add-member -inputObject $Server -membertype noteProperty `
                -name "CPU" -value $VSMachine.CpuUtilization
            add-member -inputObject $Server -membertype noteProperty `
                -name "Memory" -value $Memory
            add-member -inputObject $Server -membertype noteProperty `
                -name "Disk" -value $Disk
            }
        else{
            add-member -inputObject $Server -membertype noteProperty `
                -name "Status" -value "Hors ligne"
            }
    }
    }
```

La dernière étape consiste à afficher les informations présentes dans la variable `$Servers` sur la console PowerShell à l'aide de l'applet de commande `Format-Table`. Cette applet permet d'ajouter des propriétés calculées. Dans notre exemple, elle modifie les étiquettes des propriétés provenant de `$Servers`. L'opérateur de format (`-f`) fixe la mise en forme de ces propriétés, comme le montre l'extrait de code suivant :

INFO

Pour plus d'informations sur l'opérateur `-f`, consultez la documentation de la méthode `Format` de la classe .NET `System.String` sur la page http://msdn2.microsoft.com/fr-fr/library/system.string.format(VS.80).aspx.

```
$Servers | format-table Name, Status `
    ,@{label="Uptime Mins"; expression={"{0:N0}" -f $_.Uptime}} `
    ,@{label="CPU %"; expression={$_.CPU}} `
    ,@{label="Memory MB"; expression={"{0:N0}" -f $_.Memory}} `
    ,@{label="Disk MB"; expression={"{0:N0}" -f $_.Disk}} `
    -wrap
```

En résumé

Ce chapitre s'est attaché à montrer l'utilisation de WMI conjointement à WSH et à PowerShell pour réaliser des tâches d'automation. Les exemples et les scripts présentés ne sont en aucun cas les seules tâches pouvant être menées avec WMI. Par ailleurs, vous avez également découvert la simplicité d'utilisation de WMI avec PowerShell. Armé de ces connaissances, il vous sera assez difficile d'atteindre les limites des possibilités offertes par ces deux technologies.

Lors de la présentation des scripts d'automation opérationnels, une caractéristique très puissante de PowerShell a été dévoilée. Comme nous l'avons expliqué, la compatibilité de PowerShell avec .NET Framework lui permet d'interagir avec des services Web et d'en obtenir des données. Cette caractéristique a été utilisée dans `MonitorMSVS.ps1` pour accéder aux informations disponibles dans Microsoft Virtual Server, que la solution VBScript ne permettait d'obtenir qu'avec grandes difficultés. Nous ne répéterons jamais assez que l'exemple de ce chapitre ne fait qu'aborder l'ensemble des possibilités offertes par cette caractéristique.

9

PowerShell et Active Directory

Dans ce chapitre

- Introduction
- Comparer l'utilisation d'ADSI dans WSH et dans PowerShell
- De VBScript à PowerShell

Introduction

Ce chapitre présente les interfaces des services Active Directory (ADSI, *Active Directory Services Interfaces*) et décrit les méthodes WSH (*Windows Script Host*) et PowerShell pour les tâches de gestion d'Active Directory. Pour comprendre ces concepts, nous comparons des exemples fondés sur WSH et sur PowerShell. Enfin, nous verrons la conversion VBScript vers PowerShell d'un script qui utilise ADSI pour réaliser une tâche d'automation Active Directory. L'objectif est de donner au lecteur la possibilité d'appliquer les techniques de scripts PowerShell à des besoins d'automation réels.

Comparer l'utilisation d'ADSI dans WSH et dans PowerShell

Avant de voir comment l'administration d'Active Directory peut se faire depuis PowerShell, vous devez savoir qu'ADSI représente la première interface de programmation pour la gestion d'Active Directory. La grande majorité des outils de gestion s'appuie sur ADSI pour interagir avec Active Directory. De même, la gestion d'Active Directory depuis un script se fait généralement avec ADSI.

Pour utiliser ADSI dans vos scripts, vous devez comprendre quelques concepts essentiels. Premièrement, ADSI est constitué d'un ensemble de fournisseurs : LDAP (*Lightweight Directory Access Protocol*), NDS (*Novell Directory Services*), NWCOMPAT (*Novell NetWare 3.x*) et WinNT (*Windows NT*). Ces fournisseurs permettent aux programmes externes et aux scripts de gérer différents annuaires réseau et référentiels de données, comme Active Directory, Novell NetWare 4.x NDS et NetWare 3.x Bindery, ainsi que toute infrastructure de service d'annuaire compatible LDAP (LDAP V2 et ultérieure). Cependant, il est possible de développer des fournisseurs ADSI supplémentaires afin de prendre en charge d'autres types de référentiels de données. Par exemple, Microsoft propose un fournisseur ADSI pour la gestion d'IIS (*Internet Information Services*).

Deuxièmement, un fournisseur ADSI implémente un groupe d'objets COM pour la gestion des annuaires réseau et des dépôts de données. Par exemple, un administrateur peut se servir du fournisseur ADSI WinNT pour se lier aux ressources d'un domaine Windows et les gérer car il fournit des objets pour, entre autres, les utilisateurs, les ordinateurs, les groupes et les domaines. Les objets mis à disposition par un fournisseur ADSI résident généralement dans la ressource à gérer. En accédant au fournisseur ADSI concerné, un programme ou un script peut se lier à un objet et l'administrer grâce aux méthodes et propriétés définies pour cet objet.

Troisièmement, ADSI fournit une couche d'abstraction pour que la gestion des objets puisse se faire au travers de différents services d'annuaire et référentiels de données. Cette couche d'attraction, appelée interface IADs, définit des propriétés et des méthodes communes à tous les objets ADSI. Par exemple, un objet ADSI auquel on a accédé au travers de l'interface IADs présente les caractéristiques suivantes :

- Un objet peut être identifié par un nom, une classe ou un `ADsPath`.
- Le conteneur d'un objet peut gérer la création et la suppression de cet objet.
- La définition du schéma d'un objet peut être obtenue.
- Les attributs d'un objet peuvent être chargés dans le cache de propriétés ADSI et les modifications peuvent être transmises à la source de données originelles.
- Les attributs d'un objet chargés dans le cache de propriétés ADSI peuvent être modifiés.

Quatrièmement, ADSI offre une interface supplémentaire (`IADsContainer`) pour les objets qui sont des conteneurs (comme les unités d'organisation, ou UO). Lorsqu'elle est liée à un objet conteneur, cette interface propose des méthodes communes pour créer, supprimer, déplacer, énumérer et gérer des objets enfants.

Cinquièmement, ADSI conserve un cache côté client des propriétés pour chaque objet ADSI lié ou créé. Ce cache local des informations d'un objet améliore les performances de lecture et d'écriture dans une source de données car un programme ou un script accède moins souvent

à la source de données. Il est important de comprendre que les informations d'objet contenues dans le cache des propriétés doivent être transmises à la source de données originelles. Si les modifications d'un objet ne sont pas validées, elles ne sont pas répercutées dans la source de données d'origine.

Après cette présentation de l'interaction entre ADSI et les objets d'Active Directory, nous pouvons comparer son utilisation dans WSH et PowerShell.

Utiliser ADSI dans WSH

Dans WSH, il existe deux manières d'utiliser ADSI. La première consiste à employer une méthode (comme `GetObject()` de WSH) ou une fonction (comme `GetObject()` de VBScript) pour se connecter (se lier) à un objet Active Directory. Pour cela, nous utilisons le fournisseur LDAP ou WinNT en précisant le chemin ADSI de l'objet :

```
Set objUser = GetObject("LDAP://CN=Garett Kopczynski,OU=Accounts,OU=Managed
Objects,DC=companyabc,DC=com")
```

```
Set objUser = GetObject("WinNT://companyabc.com/garett")
```

La seconde méthode passe par ADO (*ActiveX Data Objects*). ADO permet aux applications et aux scripts d'accéder à des données provenant de différentes sources en utilisant des fournisseurs OLE DB (*Object Linking and Embedding Database*). L'un d'eux est un fournisseur ADODB (*ADSI OLE DB*), qui permet d'utiliser ADO et sa prise en charge de SQL (*Structured Query Language*) ou de LDAP pour effectuer des recherches rapides dans Active Directory. L'exemple suivant montre comment rechercher un compte d'utilisateur dans Active Directory en utilisant LDAP :

```
Set objConnection = CreateObject("ADODB.Connection")
Set objCommand = CreateObject("ADODB.Command")
objConnection.Provider = "ADsDSOObject"
objConnection.Open("Active Directory Provider")
objCommand.ActiveConnection = objConnection
objCommand.Properties("Page Size") = 1000
objCommand.CommandText = _
        "<LDAP://companyabc.com>;(&(objectCategory=user)" _
        & "(sAMAccountName=tyson));sAMAccountName,distinguishedName;subtree"
Set objRecordSet = objCommand.Execute
```

Si l'utilisateur existe, le jeu d'enregistrements ADO obtenu est constitué de son `sAMAccountName` et de son `distinguishedName`. Cependant, cet exemple ne montre que la partie visible de l'iceberg. En utilisant SQL ou LDAP, nous pouvons écrire des recherches plus élaborées et retrouver des informations complexes filtrées concernant des objets Active Directory. Grâce à ADO, nos scripts Active Directory vont être plus puissants. Cependant, cette puissance a un prix. Le fournisseur ADSI OLE DB n'autorise qu'un accès en lecture seule à Active Directory et, pour interagir avec des objets, nous devons passer par ADSI.

Utiliser ADSI dans PowerShell

Dans PowerShell, il existe également deux manières de travailler avec Active Directory. La première, et la plus simple, consiste à utiliser le type abrégé `[ADSI]`. Il est analogue à `[WMI]` car le chemin de l'objet auquel nous nous connectons doit être précisé. En revanche, ce chemin est donné sous la forme d'un chemin ADSI :

```
PS C:\> $Utilisateur = [ADSI]"LDAP://CN=Garett Kopczynski,OU=Accounts,
OU=Managed Objects,DC=companyabc,DC=com"
```

Cet exemple utilise un chemin ADSI LDAP, mais d'autres fournisseurs ADSI sont disponibles avec le type abrégé `[ADSI]`. Comme nous l'avons expliqué au Chapitre 8, "PowerShell et WMI", le type abrégé `[ADSI]` de PowerShell est un alias de la classe .NET `System.DirectoryServices.DirectoryEntry`, qui peut s'interfacer avec différents fournisseurs ADSI : IIS, LDAP, NDS et WinNT. Par exemple, pour accéder au même compte d'utilisateur, mais au travers du fournisseur ADSI WinNT, nous utilisons la commande suivante :

```
PS C:\> $Utilisateur = [ADSI]"WinNT://companyabc.com/garett"
```

La deuxième méthode consiste à utiliser l'espace de noms .NET `System.DirectoryServices` par le biais de l'applet de commande `New-Object`. Dans ce cas, deux classes de composants nous permettent de gérer Active Directory. La première, `System.DirectoryServices.DirectoryEntry`, est la même classe que celle employée par le type abrégé `[ADSI]`. En voici un exemple :

```
PS C:\> $Utilisateur = new-object DirectoryServices.DirectoryEntry ("LDAP://
CN=Garett Kopczynski,OU=Accounts,OU=Managed Objects,DC=companyabc,DC=com")
```

La seconde, `System.DirectoryServices.DirectorySearcher`, est une classe qui permet d'effectuer des recherches LDAP :

```
PS C:\> $Chercheur = new-object DirectoryServices.DirectorySearcher
PS C:\> $Chercheur.Filter = "(&(objectCategory=person)(objectClass=user)
(samAccountName=garett))"
PS C:\> $Utilisateur = $Chercheur.FindOne().GetDirectoryEntry()
```

Les méthodes d'utilisation d'ADSI dans PowerShell sont comparables à celles dans WSH. Comme WSH, PowerShell dispose d'une méthode directe impliquant la classe `System.DirectoryServices.DirectoryEntry` ou le type abrégé `[ADSI]` pour se connecter à des objets Active Directory et les gérer. Par ailleurs, comme WSH, PowerShell propose une deuxième méthode qui s'appuie sur la classe `System.DirectoryServices.DirectorySearcher` pour effectuer des recherches dans Active Directory et obtenir des informations en lecture seule à propos des objets.

Par conséquent, la gestion d'Active Directory est pratiquement identique dans PowerShell et dans WSH. Même si PowerShell utilise pour cela .NET Framework, les classes `System.DirectoryServices.DirectoryEntry` et `System.DirectoryServices.DirectorySearcher` ne sont que des interfaces .NET pour ADSI. Les différences entre WSH et PowerShell se situent uniquement dans les fonctions et les méthodes de gestion d'Active Directory, ainsi que dans leur syntaxe. Les deux sections suivantes examinent ces similitudes en expliquant comment retrouver des informations sur un objet et comment créer un objet en utilisant VBScript et PowerShell.

Obtenir des informations sur un objet

L'exemple VBScript suivant se lie à l'objet d'utilisateur indiqué à l'aide de la méthode VBScript `GetObject()` et un fournisseur ADSI LDAP. Le script récupère ensuite les attributs `Name`, `userPrincipalName`, `description` et `physicalDeliveryOfficeName` de l'objet d'utilisateur. Enfin, il les affiche par l'intermédiaire d'une boîte de message ou sur la console :

```
Set objUser = GetObject("LDAP://CN=Garett Kopczynski,OU=Accounts,OU=Managed Obje
cts,DC=companyabc,DC=com")
WScript.Echo objUser.Name
WScript.Echo objUser.userPrincipalName
WScript.Echo objUser.description
WScript.Echo objUser.physicalDeliveryOfficeName
```

L'enregistrement de ce script dans le fichier `getuserinfo.vbs` et son exécution avec `cscript` produisent les résultats suivants :

```
C:\>cscript getuserinfo.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

CN=Garett Kopczynski
Garett@companyabc.com
Marketing Manager
Dallas

C:\>
```

Pour effectuer la même tâche dans PowerShell, nous utilisons le type abrégé `[ADSI]` pour établir une liaison avec l'objet d'utilisateur indiqué. La méthode ADSI `Get()` retrouve ses attributs :

```
PS C:\> $Utilisateur = [ADSI]"LDAP://CN=Garett Kopczynski,OU=Accounts,OU=Man
aged Objects,DC=companyabc,DC=com"
PS C:\> $Utilisateur.Get("Name")
Garett Kopczynski
PS C:\>
```

Une fois que nous sommes liés à l'objet d'utilisateur, nous pouvons accéder directement à ses attributs depuis PowerShell en utilisant n'importe quelle applet de mise en forme ou de manipulation de l'objet. Par exemple, pour accéder aux mêmes attributs que dans l'exemple VBScript et les afficher, nous invoquons l'applet de commande `Format-List` :

```
PS C:\> $Utilisateur | format-list Name, userPrincipalName, description,
physicalDeliveryOfficeName


name                       : {Garett Kopczynski}
userPrincipalName          : {Garett@taosage.net}
description                : {Marketing Manager}
physicalDeliveryOfficeName : {Dallas}


PS C:\>
```

Créer un objet

L'exemple VBScript suivant se lie à l'UO Accounts en utilisant la méthode VBScript `GetObject()` avec un fournisseur ADSI LDAP. Ensuite, le script invoque la méthode ADSI `Create()` pour créer un objet nommé David Lightman dans l'UO Accounts, puis il définit les attributs du nouvel objet d'utilisateur à l'aide de la méthode ADSI `Put()`. Enfin, ce nouvel objet est enregistré dans Active Directory à l'aide de la méthode ADSI `SetInfo()`. Un message d'état concernant la création de l'objet est affiché dans une boîte de message ou sur la console :

```
Set objOU = GetObject("LDAP://OU=Accounts,OU=Managed Objects,DC=companyabc,DC=com")
Set objUser = objOU.Create("user", "CN=David Lightman")
objUser.Put "sAMAccountName", "dlightman"
objUser.Put "sn", "Lightman"
objUser.Put "givenName", "David"
objUser.Put "userPrincipalName", "dlightman@norad.gov"
objUser.SetInfo
Wscript.Echo "Le compte d'utilisateur " & objUser.Get("sAMAccountName") & " a été créé."
```

L'enregistrement de ce script dans le fichier `createuserinfo.vbs` et son exécution avec `cscript` produisent les résultats suivants :

```
C:\>cscript createuserinfo.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

Le compte d'utilisateur dlightman a été créé.
C:\>
```

Pour réaliser la même tâche dans PowerShell, nous utilisons le type abrégé `[ADSI]`. Les commandes résultantes ont une logique et une syntaxe analogues à celles de l'exemple VBScript. Par exemple, pour créer l'objet d'utilisateur, nous établissons une liaison avec l'UO Accounts, puis nous créons un nouvel objet nommé David Lightman en invoquant la méthode ADSI `Create()`. Ensuite, la méthode ADSI `Put()` nous permet de définir les attributs de l'objet d'utilisateur, puis celui-ci est enregistré dans Active Directory grâce à la méthode ADSI `SetInfo()`. Enfin, pour vérifier la création du compte, nous nous lions à l'objet d'utilisateur *via* le type abrégé `[ADSI]`.

Cette procédure est mise en œuvre par le code suivant :

```
PS C:\> $UO = [ADSI]"LDAP://OU=Accounts,OU=Managed Objects,DC=companyabc,DC=com"
PS C:\> $NouvUtil = $UO.Create("user", "CN=David Lightman")
PS C:\> $NouvUtil.Put("sAMAccountName", "dlightman")
PS C:\> $NouvUtil.Put("sn", "Lightman")
PS C:\> $NouvUtil.Put("givenName", "David")
PS C:\> $NouvUtil.Put("userPrincipalName", "dlightman@norad.gov")
PS C:\> $NouvUtil.SetInfo()
PS C:\> [ADSI]"LDAP://CN=David Lightman,OU=Accounts,OU=Managed Objects,
        DC=companyabc,DC=com"

distinguishedName
-----------------
{CN=David Lightman,OU=Accounts,OU=Managed Objects,DC=companyabc,DC=com}


PS C:\
```

INFO

Si vous essayez cet exemple dans votre environnement, vous remarquerez que l'objet d'utilisateur résultant est initialement désactivé, car la valeur par défaut de `userAccountControl` est 514, autrement dit le compte est désactivé. Pour que cet exemple fonctionne, nous devons définir des paramètres supplémentaires, comme le mot de passe de l'utilisateur, l'état de son compte, son groupe, etc.

De VBScript à PowerShell

Cette section présente la conversion VBScript vers PowerShell d'un script qui détermine si des utilisateurs sont membres d'un groupe précis.

Au moment du développement de ce script, companyabc.com était en pleine migration des utilisateurs de l'ancienne application de gestion vers la nouvelle. Pour rationaliser le processus et limiter les interruptions dans le travail des employés, la migration se faisait en plusieurs étapes. L'une d'elles consistait à produire la liste des utilisateurs à passer de l'ancienne application à la nouvelle. Chaque utilisateur de la liste devait arriver dans la nouvelle application avec une configuration établie sur une appartenance à un groupe Active Directory.

Cependant, face aux milliers d'utilisateurs et de groupes, la vérification manuelle des groupes de chaque utilisateur inscrit sur la liste de migration demandait beaucoup de temps et la génération des rapports beaucoup d'efforts. companyabc.com avait donc besoin d'automatiser le processus de vérification des groupes pour que la migration puisse se faire sans interruption. Pour cela, cette entreprise a demandé un script qui prenne la liste des utilisateurs à faire migrer et produise un rapport précisant les appartenances aux groupes de ces utilisateurs.

Le script IsGroupMember.wsf

`IsGroupMember.wsf` est un fichier VBScript développé pour s'occuper du processus de vérification des groupes de la société companyabc.com. Vous en trouverez une copie dans le dossier `Scripts\Chapitre 9\IsGroupMember` et en téléchargement depuis le site **www.pearson-education.fr**. Pour exécuter ce script, il est nécessaire de définir deux paramètres. L'argument de `groupname` doit indiquer le `sAMAccountName` du groupe pour lequel nous devons vérifier l'appartenance de l'utilisateur. L'argument d'`importfile` doit désigner le nom du fichier CSV importé qui contient les utilisateurs devant être vérifiés. Le paramètre facultatif `exportfile` doit préciser le nom du fichier d'exportation dans lequel le script placera son rapport.

INFO

Le fichier CSV importé doit contenir une seule colonne (`sAMAccountName`). Le fichier `users.csv`, qui se trouve dans le dossier `Scripts\Chapitre 9\IsGroupMember` et que vous pouvez télécharger depuis le site **www.pearsoneducation.fr**, en est un exemple.

Voici la commande qui permet d'exécuter le script `IsGroupMember.wsf` et dont le résultat est présenté à la Figure 9.1 :

```
D:\Scripts>cscript IsGroupMember.wsf /groupname: "TAO-D-RA-LS-LocalWorkstation
PowerUsers" /importfile:".\users.csv" /exportfile:"export.csv"
```

Voici la suite des opérations réalisées par `IsGroupMember.wsf` :

1. Le script teste la connexion avec le domaine actuel en obtenant son `DefaultNamingContext`, qui sera utilisé ensuite pour interroger Active Directory. Si la connexion échoue, le script s'arrête.
2. Il crée un objet de connexion ADO, utilisé ensuite pour effectuer une recherche Active Directory à l'aide du fournisseur ADSI OLE DB.

```
C:\WINDOWS\system32\cmd.exe
C:\Scripts>cscript IsGroupMember.wsf /groupname:"TAO-D-RA-LS-LocalWorkstationPow
erUsers" /importfile:".\users.csv" /exportfile:"export.csv"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

##########################################
#            IsGroupMember               #
##########################################

Vérification de la connexion au domaine          [OK]
Établissement de la connexion ADODB              [OK]
importation du fichier CSV               [OK]
Obtention des informations d'appartenance à un groupe   [OK]
Obtention des informations sur l'utilisateur            [OK]

[Nom],[Est membre]
tyson,Oui
maiko,Oui
fujo,Non
dawn,Inexistant
kiyomi,Oui
med,Inexistant
erica,Oui
bill,Inexistant
miro,Inexistant
yelena,Inexistant

Création du fichier d'exportation[OK]

C:\Scripts>
```

Figure 9.1

Exécution du script IsGroupMember.wsf.

3. Ensuite, la fonction `ParseFile` ouvre le fichier CSV et lit les informations d'utilisateurs dans le tableau indiqué (`arrUsers`). Si le fichier précisé est invalide, la fonction échoue et le script s'arrête.

4. Le script interroge Active Directory à propos du groupe indiqué en passant par l'objet ADO. Si le groupe n'est pas valide, le script s'arrête. Dans le cas contraire, il se connecte au groupe en utilisant ADSI, récupère ses membres et les ajoute à l'objet `Dictionary` du groupe (`dictGroup`).

5. Puis, le script parcourt chaque utilisateur du tableau `arrUsers`, se connecte à chaque objet d'utilisateur avec ADSI et récupère son `distinguishedName`. Les utilisateurs invalides sont ajoutés à l'objet `Dictionary` d'utilisateur (`dictUsers`) avec la valeur "Inexistant". Si l'utilisateur est valide, le script vérifie l'existence de son `distinguishedName` dans l'objet `dictGroup`. Les utilisateurs membres du groupe sont ajoutés à l'objet `dictUsers` avec la valeur "Oui". Ceux qui ne font pas partie du groupe sont ajoutés à `dictUsers` avec la valeur "Non".

6. Enfin, le script affiche les informations de l'objet `dictUsers` sur la console. Si un fichier d'exportation a été précisé, ces mêmes informations y sont écrites.

Le premier extrait de code donne des éléments XML initiaux d'un fichier WSF. Ils définissent les paramètres acceptés, décrivent le script, donnent des exemples d'utilisation et précisent le langage employé :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<package>
<job id="IsGroupMember">
    <runtime>
        <description>
************************************************************
Ce script vérifie si des utilisateurs sont membres du groupe indiqué.
************************************************************
        </description>
        <named name="groupname" helpstring="Nom du groupe à vérifier."
type="string" required="1" />
        <named name="importfile" helpstring="Fichier CSV à importer."
type="string" required="1" />
        <named name="exportfile" helpstring="Fichier CSV à exporter."
type="string" required="0" />
        <example>
Exemple :
cscript ISGroupMember.wsf /groupname:"monGroupe" /importfile:"users.csv"
        </example>
    </runtime>
<script language="VBScript">
<![CDATA[
```

Le script vérifie ensuite si des arguments ont été définis pour les paramètres obligatoires `groupname` et `importfile`. Si ce n'est pas le cas, il affiche les informations d'utilisation (définies dans le code précédent) sur la console et se termine. Lorsque les arguments sont précisés, le script configure son environnement en définissant les variables utilisées par la suite.

Puisque les tableaux VBScript ne simplifient pas l'enregistrement et l'accès aux données, ce script utilise l'objet `Dictionary` de la bibliothèque Windows Scripting Runtime Library (`dictGroup` et `dictUsers`). Contrairement aux tableaux habituels, l'objet `Dictionary` enregistre les données sous forme de paires clé/valeur. Grâce à cette méthode, nous pouvons accéder aux données du tableau en précisant la clé, utiliser les méthodes et les propriétés de l'objet

`Dictionary` sur les données du tableau et ajouter ou retirer des données dynamiquement du tableau sans devoir le redimensionner :

```
'On Error Resume Next

'====================================================================
' Vérifier les arguments obligatoires.
'====================================================================
If WScript.Arguments.Named.Exists("groupname") = FALSE Then
    WScript.Arguments.ShowUsage()
    WScript.Quit
End If

If WScript.Arguments.Named.Exists("importfile") = FALSE Then
    WScript.Arguments.ShowUsage()
    WScript.Quit
End If

'====================================================================
' Définir l'environnement de travail.
'====================================================================
Const ForReading = 1
Const ForWriting = 2
ReDim arrUsers(0)
Dim arrMemberOf
Dim StdOut
Dim FSO
Dim strGroupName, strImportFile, strExportFile
Dim strDNSDomain, dictGroup, dictUsers

Set StdOut = WScript.StdOut
Set FSO = CreateObject("Scripting.FileSystemObject")
Set dictGroup = CreateObject("Scripting.Dictionary")
Set dictUsers = CreateObject("Scripting.Dictionary")

strGroupName = WScript.Arguments.Named("groupname")
strImportFile = WScript.Arguments.Named("importfile")
strExportFile = WScript.Arguments.Named("exportfile")
```

Le prochain extrait commence le code d'automation. Tout d'abord, le script affiche son en-tête sur la console, se lie à l'objet `RootDSE` et récupère le `DefaultNamingContext`. Nous procédons ainsi pour deux raisons. D'une part, le script vérifie la validité d'une connexion au domaine Active Directory. Ce test est réalisé car si le script ne peut se connecter au domaine Active

Directory à ce stade de son exécution, il échouera ensuite lorsqu'il tentera d'extraire des informations d'Active Directory. Comme au Chapitre 8, il s'agit d'une forme élaborée de gestion des erreurs, qui détermine les risques d'échec d'un script inclut une méthode qui les évite.

D'autre part, le script doit obtenir le nom de domaine d'ouverture de session courant pour plus tard. Sans cette information, il devrait être modifié afin de demander aux utilisateurs à quel domaine doivent être réclamées les informations d'appartenance au groupe. Dans les environnements constitués de multiples domaines, cette fonctionnalité pourrait être ajoutée au script. Cependant, dans notre exemple, elle est inutile et le script obtient le nom de domaine depuis l'objet `RootDSE` et le place dans la variable `strDNSDomain` :

```
'=====================================================================
' Commencer le travail.
'=====================================================================
Mess "#######################################"
Mess "#            IsGroupMember             #"
Mess "#######################################"
Mess vbNullString

'--------------------
' Tester la connexion au domaine.
'--------------------
StatStart "Vérification de la connexion au domaine"
    Set objRootDSE = GetObject("LDAP://RootDSE")
    strDNSDomain = objRootDSE.Get("DefaultNamingContext")

    Xerror
StatDone
```

Le code suivant crée un objet ADO (`objConnection`), qui sera utilisé par la suite. La fonction `ParseFile` importe les informations d'utilisateur depuis le fichier CSV et les place dans le tableau `arrUsers` :

```
'--------------------
' Établir une connexion ADODB.
'--------------------
StatStart "Établissement de la connexion ADODB"
    Dim objConnection
    Dim objCommand
    Dim objRecordSet
```

```
    Set objConnection = CreateObject("ADODB.Connection")
    Set objCommand = CreateObject("ADODB.Command")
    objConnection.Provider = "ADsDSOObject"
    objConnection.Open("Active Directory Provider")
    objCommand.ActiveConnection = objConnection
    objCommand.Properties("Page Size") = 1000

    Xerror
StatDone

'--------------------
' Importer le fichier CSV.
'--------------------
StatStart "importation du fichier CSV"
    ParseFile strImportFile, arrUsers
StatDon
```

Ensuite, le script utilise l'objet ADO créé pour effectuer une recherche LDAP sur le groupe indiqué dans le domaine Active Directory courant. En fonction des informations obtenues, il détermine si le groupe existe et son `distinguishedName`. Puis, grâce à ce `distinguishedName`, il se lie directement à l'objet de groupe dans Active Directory et obtient les membres du groupe, lesquels sont placés dans le tableau `arrMemberOf`. Une boucle `For` parcourt ce tableau et ajoute chaque membre du groupe à l'objet `dictGroup` avec la valeur temporaire "A fixer" (qui peut être quelconque, tant que la paire clé/valeur est complète) :

```
'--------------------
' Obtenir les informations d'appartenance à un groupe.
'--------------------
StatStart "Obtention des informations d'appartenance à un groupe"
    objCommand.CommandText = _
        "<LDAP://" & strDNSDomain & ">;(&(objectCategory=group)" _
        & "(sAMAccountName=" & strGroupName & "));distinguishedName;subtree"

    Set objRecordSet = objCommand.Execute

    If objRecordset.RecordCount = 0 Then
        StdOut.Write(vbTab)
        StdOut.WriteLine("Groupe invalide !")
        WScript.Quit()
    Else
```

```
        Set objGroup = GetObject _
            ("LDAP://" & objRecordSet.Fields("distinguishedName"))
        objGroup.getInfo

        arrMemberOf = objGroup.GetEx("member")

        For Each Member in arrMemberOf
            dictGroup.Add Member, "A fixer"
        Next
    End If

    Set objGroup = Nothing
StdOut.Write(vbTab)
StdOut.WriteLine("[OK]")
```

Le code suivant parcourt le tableau `arrUsers` créé lors de l'importation du fichier CSV :

```
'--------------------
' Obtenir les informations sur l'utilisateur.
'--------------------
StatStart "Obtention des informations sur l'utilisateur"

For Each User In arrUsers
    Err.Clear

    objCommand.CommandText = _
        "<LDAP://" & strDNSDomain & ">;(&(objectCategory=user)" _
        & "(sAMAccountName=" & User & "));distinguishedName;subtree"

    Set objRecordSet = objCommand.Execute

    If objRecordset.RecordCount = 0 Then
        dictUsers.Add User, "Inexistant "
    Else
        strUserDN = objRecordSet.Fields("distinguishedName")

        If (dictGroup.Exists(strUserDN) = True) Then
            dictUsers.Add User, "Oui"
        Else
            dictUsers.Add User, "Non"
        End If
    End If
```

```
    Set objRecordset = Nothing
    strUserDN = vbNullString
Next
StatDone
```

Pour chaque utilisateur, le script lance une recherche LDAP dans le domaine d'ouverture de session courant en utilisant l'objet ADO. Les utilisateurs qui n'existent pas sont ajoutés à l'objet `dictUsers` avec la valeur "Inexistant". En revanche, lorsqu'un utilisateur existe, le script prend son `distinguishedName` dans le jeu d'enregistrements retourné par la recherche LDAP et, par une comparaison, détermine si cet utilisateur existe dans l'objet `dictGroup`.

Pour effectuer cette comparaison, le script utilise la méthode `Exists()` de l'objet `Dictionary`. Elle permet de vérifier l'existence d'une clé dans l'objet `Dictionary`. Cette possibilité est la principale raison d'utiliser l'objet `Dictionary` à la place d'un tableau VBScript. Ensuite, en fonction de la valeur retournée par la méthode `Exists()`, le script ajoute l'utilisateur à l'objet `dictUsers` avec la valeur "Oui", pour indiquer qu'il appartient au groupe, ou avec la valeur "Non", s'il n'est pas membre du groupe.

Nous obtenons une collection d'informations d'utilisateur dans l'objet `dictUsers`. Le script prend ensuite chaque utilisateur présent dans l'objet `dictUsers` et affiche ses informations sur la console :

```
Mess vbNullString
Mess "[Nom],[Est membre]"

For Each User In dictUsers
    StdOut.Write User & ","
    StdOut.WriteLine dictUsers.Item(User)
Next
```

Si la variable `exportfile` a été définie lors de l'exécution du script, le fichier d'exportation est créé à l'aide de l'objet FSO. Puis, le script parcourt à nouveau l'objet `dictUsers` et écrit les informations des utilisateurs dans ce fichier.

Le code suivant implémente cette procédure :

```
'--------------------
' Créer un fichier d'exportation.
'--------------------
Mess vbNullString
StdOut.Write "Création du fichier d'exportation"

If strExportFile <> "" Then
    Set objExportFile = FSO.OpenTextFile(strExportFile, ForWriting, TRUE)

    For Each User In dictUsers
        objExportFile.Write User & ","
        objExportFile.WriteLine dictUsers.Item(User)
    Next

    objExportFile.Close()
    Set objExportFile = Nothing

    StdOut.WriteLine "[OK]"
End If
```

Le dernier extrait de code est constitué des procédures et fonctions utilisées par le script, ainsi que des éléments XML qui le terminent. Il est inutile de revenir sur cette partie du script car ces procédures et fonctions sont suffisamment explicites ou ont déjà été présentées :

```
'====================================================================
' Procédures.
'====================================================================
'--------------------
' Procédure générale pour les messages.
'--------------------
Sub Mess(Message)
    ' Écrire sur la console.
    StdOut.WriteLine(Message)
End Sub

'--------------------
' Procédure générale pour le début d'un message.
'--------------------
Sub StatStart(Message)
    ' Écrire sur la console.
    StdOut.Write(Message)
```

```
End Sub

'-------------------
' Procédure générale pour la fin d'un message.
'-------------------
Sub StatDone
    ' Écrire sur la console.
    StdOut.Write(vbTab & vbTab)
    StdOut.WriteLine("[OK]")
End Sub

'-------------------
' Procédure générale pour Xerror.
'-------------------
Sub Xerror
    If Err.Number <> 0 Then
        ' Écrire sur la console.
        StdOut.WriteLine(" Erreur fatale : " & CStr(Err.Number) _
            & " " & Err.Description)

        WScript.Quit()
    End If
End Sub

'=================================================================
' Fonctions.
'=================================================================
Function ParseFile(file, arrname)
    ' Analyser un fichier et retourner son contenu dans un tableau
    ' (la première ligne est sautée !!!)
    On Error Resume Next
    count = -1

    ' Ouvrir le fichier en lecture.
    Set objFile = FSO.OpenTextFile(file, ForReading)
    objFile.SkipLine ' Note : cette ligne contient les en-têtes de colonnes.
    Xerror

    ' Lire chaque ligne du fichier et la placer dans un tableau.
    Do While objFile.AtEndOfStream <> True
        count = count + 1
        If count > UBound(arrname) Then ReDim Preserve arrname(count)
        arrname(count) = objFile.Readline
```

```
    Loop
    Xerror

    ' Fermer le fichier.
    objFile.Close()
    Set objFile = Nothing
    count = 0
End Function
]]>
  </script>
</job>
</package>
```

Le script IsGroupMember.ps1

`IsGroupMember.ps1` est une version PowerShell du script `IsGroupMember.wsf`. Vous en trouverez une copie dans le dossier `Scripts\Chapitre 9\IsGroupMember` et en téléchargement depuis le site **www.pearsoneducation.fr**. Pour exécuter ce script, il est nécessaire de définir deux paramètres. L'argument de `GroupName` doit indiquer le `sAMAccountName` du groupe pour lequel nous devons vérifier l'appartenance de l'utilisateur. L'argument d'`ImportFile` doit désigner le nom du fichier CSV importé qui contient les utilisateurs à vérifier. Le paramètre facultatif `ExportFile` doit préciser le nom du fichier d'exportation dans lequel le script pourra écrire son rapport. Voici la commande d'exécution du script `IsGroupMember.ps1` :

```
PS D:\Scripts> .\IsGroupMember.ps1 "TAO-D-RA-LS-LocalWorkstationPowerUsers" ".\
users.csv" "export.csv"
```

La Figure 9.2 montre l'exécution du script `IsGroupMember.ps1` sans qu'un fichier d'exportation soit précisé et la Figure 9.3 montre son exécution avec un fichier d'exportation.

Voici la suite des opérations réalisées par le script `IsGroupMember.ps1` :

1. Il se connecte au domaine d'ouverture de session actuel en utilisant la classe .NET `System.DirectoryServices.ActiveDirectory.Domain`, puis il obtient le nom du domaine et l'affiche sur la console PowerShell. Si la connexion échoue, le script s'arrête.
2. Il vérifie que le groupe indiqué existe dans le domaine actuel en utilisant la fonction `Get-ADObject`. Si c'est le cas, la fonction retourne un objet lié à l'objet de groupe dans Active Directory (`$Group`). Sinon, le script se termine.
3. Il utilise l'applet de commande `Test-Path` pour vérifier que le fichier CSV à importer est valide. Si ce n'est pas le cas, le script se termine.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\IsGroupMember.ps1 "TAO-D-RA-LS-LocalWorkstationPowerUsers" ".\u
sers.csv"

#########################################
# Script          IsGroupMember
# Usage :         Ce script vérifie si des utilisateurs sont membres du groupe i
ndiqué.
# Utilisateur :   tyson
# Date :          Tue Oct 30 11:27:10     2007
#########################################

Vérification de la connexion au domaine  taosage.internal
Vérification du nom du groupe    [OK]
Vérification du fichier à importer       [OK]

sAMAccountName                           IsMember
--------------                           --------
tyson                                    Oui
maiko                                    Oui
fujio                                    Non
dawn                                     Inexistant
kiyomi                                   Non
med                                      Inexistant
erica                                    Oui
bill                                     Inexistant
miro                                     Inexistant
yelena                                   Inexistant


PS C:\Scripts>
```

Figure 9.2

Exécution du script IsGroupMember.ps1 sans fichier d'exportation.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\IsGroupMember.ps1 "TAO-D-RA-LS-LocalWorkstationPowerUsers" ".\u
sers.csv" "export.csv"

#########################################
# Script          IsGroupMember
# Usage :         Ce script vérifie si des utilisateurs sont membres du groupe i
ndiqué.
# Utilisateur :   tyson
# Date :          Tue Oct 30 11:27:54     2007
#########################################

Vérification de la connexion au domaine  taosage.internal
Vérification du nom du groupe    [OK]
Vérification du fichier à importer       [OK]

Exportation des données dans : export.csv

PS C:\Scripts>
```

Figure 9.3

Exécution du script IsGroupMember.ps1 avec fichier d'exportation.

4. Il invoque l'applet de commande `Import-Csv` pour placer le contenu du fichier CSV dans la variable `$Users`.
5. Il appelle la fonction `Get-ADObject` pour vérifier que chaque utilisateur de la collection `$Users` existe dans le domaine actuel et pour se lier à l'objet d'utilisateur correspondant dans Active Directory.
6. Si l'utilisateur existe, le script compare son `distinguishedName` aux noms distinctifs dans l'attribut de membre du groupe indiqué (`$Group`). Lorsqu'une correspondance est trouvée, l'objet d'utilisateur est étendu à l'aide de l'applet de commande `Add-Member` pour indiquer que l'utilisateur est membre du groupe ("Oui"). Dans le cas contraire, il est étendu de manière à indiquer que l'utilisateur n'est pas membre du groupe ("Non"). Si l'utilisateur n'existe pas dans le domaine actuel, l'objet d'utilisateur est étendu pour indiquer ce fait ("Inexistant").
7. Si un fichier d'exportation a été précisé, le script utilise l'applet de commande `Export-Csv` pour créer un fichier CSV à partir du contenu de la variable `$Users`. Si ce fichier n'est pas demandé, il affiche ce contenu sur la console de PowerShell.

Le premier extrait de code contient l'en-tête du script `IsGroupMember.ps1`. Il fournit des informations sur le rôle du script, sa date de dernière mise à jour et son auteur. Juste après l'en-tête, nous trouvons les paramètres du script :

```
##################################################
# IsGroupMember.ps1
# Ce script vérifie si des utilisateurs sont membres du groupe indiqué.
#
# Créé le: 21/10/2006
# Auteur : Tyson Kopczynski
##################################################
param([string] $GroupName, [string] $ImportFile, [string] $ExportFile)
```

Ensuite, le script contient les fonctions `Get-ScriptHeader` et `Show-ScriptUsage` :

```
##################################################
# Fonctions.
##################################################
#-------------------------------------------------
# Get-ScriptHeader
#-------------------------------------------------
# Usage :        Générer l'instruction d'en-tête du script.
```

```
# $Name :        Nom du script.
# $Usage :       Rôle du script.

function Get-ScriptHeader{
    param ($Name, $Usage)

    $Date = Date

    $Text = „##################################### `n"
    $Text += „# Script         $Name `n"
    $Text += "# Usage :        $Usage `n"
    $Text += "# Utilisateur :  $Env:username `n"
    $Text += "# Date :         $Date `n"
    $Text += "#####################################"

    $Text
    }

#-------------------------------------------------
# Show-ScriptUsage
#-------------------------------------------------
# Usage :        Afficher les informations d'utilisation de script.

function Show-ScriptUsage{
    write-host
    write-host "Usage : ISGroupMember -GroupName valeur" `
                    "-ImportFile valeur -ExportFile valeur"
    write-host
    write-host "Options :"
    write-host
    write-host "GroupName `t : nom du groupe à vérifier."
    write-host "ImportFile `t : fichier CSV à importer."
    write-host "ExportFile `t : [facultatif] fichier CSV à exporter."
    write-host
    write-host "Format CSV :"
    write-host "sAMAccountName"
    write-host
    write-host "Exemple :"
    write-host 'ISGroupMember.ps1 "monGroupe"' `
                    '"utilisateurs.csv" "resultats.csv"'
    write-host
    }
```

Ces fonctions sont utilisées pour afficher les informations d'utilisation du script, de manière analogue à celles affichées par un script WSF :

```
PS D:\Scripts> .\IsGroupMember.ps1

Veuillez préciser le nom du groupe !

#####################################
# Script          IsGroupMember
# Usage :         Ce script vérifie si des utilisateurs sont membres du groupe
                   indiqué.
# Utilisateur :   tyson
# Date :          Mon Oct 22 17:37:19     2007
#####################################

Usage : ISGroupMember -GroupName valeur -ImportFile valeur -ExportFile valeur

Options :

GroupName         : nom du groupe à vérifier.
ImportFile        : fichier CSV à importer.
ExportFile        : [facultatif] fichier CSV à exporter.

Format CSV :
sAMAccountName

Exemple :
ISGroupMember.ps1 "monGroupe" "utilisateurs.csv" "resultats.csv"

PS D:\Scripts>
```

Un fichier WSF a cela d'intéressant qu'il peut fournir aux utilisateurs des informations sur l'objectif du script, ses paramètres et des exemples d'utilisation. Ils n'ont donc pas besoin de lire les commentaires ou de se référer à une documentation externe pour comprendre le rôle du script et son usage. Cette caractéristique améliore l'expérience de l'utilisateur avec un script d'automation et augmente donc les chances qu'il soit considéré comme indispensable.

Malheureusement, cette fonctionnalité n'existe pas dans PowerShell. Au mieux, nous pouvons définir les paramètres requis et afficher des informations sur leur utilisation au moyen du mot clé `throw`. Ce mot clé a été employé dans les scripts précédents, mais il n'affichait aucune information dans un format aussi convivial que celui des scripts WSF. Pour obtenir le même niveau d'interface, nous avons développé les fonctions `Show-ScriptUsage` et `Get-ScriptHeader`. La première, `Show-ScriptUsage`, définit le rôle du script, ses paramètres et son utilisation.

Même si nous pouvons réemployer la structure de cette fonction dans d'autres scripts, son contenu est statique et doit être modifié chaque fois. La seconde, `Get-ScriptHeader`, affiche simplement le titre du script. Elle peut servir dans d'autres scripts sans grandes modifications, car les paramètres `$Name` et `$Usage` définissent les informations affichées.

Au final, les informations d'utilisation du script affichées dans l'exemple précédent grâce à ces fonctions sont comparables à ce que peut produire un script WSF. Bien que la modification de `Show-ScriptUsage` dans chaque nouveau script soit un peu pénible, ces fonctions génériques simples ont l'avantage de montrer qu'un script a été écrit pour ses utilisateurs et non pour des développeurs. Nous utiliserons ces fonctions dans la suite de cet ouvrage.

INFO

Nous pouvons améliorer la fonction `Show-ScriptUsage` en la rendant plus générique et éviter ainsi sa modification dans les autres scripts. Par exemple, les informations retournées par cette fonction peuvent s'appuyer sur une chaîne XML dont la structure ressemble à celle d'un fichier WSF.

Après ces fonctions outils, les deux suivantes mettent en œuvre les interactions avec Active Directory :

```
#-----------------------------------------------
# Get-CurrentDomain
#-----------------------------------------------
# Usage :        Obtenir le domaine actuel.

function Get-CurrentDomain{
    [System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()
    }

#-----------------------------------------------
# Get-ADObject
#-----------------------------------------------
# Usage :        Retrouver un objet depuis Active Directory.
# $Item :        Élément de l'objet (sAMAccountName ou distinguishedName).
# $Name :        Nom de l'objet (sAMAccountName ou distinguishedName).
# $Cat :         Catégorie de l'objet.

function Get-ADObject{
    param ($Item, $Name, $Cat)
```

```
trap{Continue}

$Searcher = new-object DirectoryServices.DirectorySearcher
$SearchItem = "$Item"
$SearchValue = "$Name"
$SearchCat = "$Cat"

$Searcher.Filter = `
    "(&($($SearchItem)=$($SearchValue))(objectCategory=$($SearchCat)))"

$Searcher.FindOne().GetDirectoryEntry()
}
```

La première fonction, `Get-CurrentDomain`, est simple et se contente d'établir une liaison avec l'objet du domaine d'ouverture de session courant. Pour cela, elle s'appuie sur une référence .NET Framework à la classe `System.DirectoryServices.ActiveDirectory.Domain` (voir le Chapitre 3, "Présentation avancée de PowerShell") avec la méthode `GetCurrentDomain()`. Ensuite, elle retourne l'objet de domaine résultant, qui vérifie la connexion au domaine et fournit une méthode permettant d'afficher le nom DNS du domaine aux utilisateurs de script (un rappel visuel indiquant que les informations du domaine sont en cours de consultation).

La deuxième fonction, `Get-ADObject`, vérifie l'existence d'un objet dans Active Directory en fonction d'un identifiant unique, comme l'attribut `sAMAccountName` ou `distinguishedName`. Ensuite, elle se connecte à cet objet à l'aide de la classe `System.DirectoryServices.DirectorySearcher`, qui est une méthode .NET pour effectuer des recherches Active Directory. Lors de l'appel à `Get-ADObject`, nous devons fournir l'identifiant unique de l'objet (`$Name`), le type (`$Item`) de cet identifiant unique (`sAMAccountName` ou `distinguishedName`) et le type (Utilisateur, Ordinateur ou Groupe) de la catégorie (`$Cat`) de l'objet. À partir de ces valeurs, `Get-ADObject` crée un objet `$Searcher`, dont il fixe la propriété `Filter` à une chaîne de recherche LDAP fondée sur les valeurs fournies. Ensuite cette fonction utilise la méthode `FindOne()` de l'objet `$Searcher` pour effectuer la recherche et obtenir uniquement la première entrée trouvée. Enfin, la méthode `GetDirectoryEntry()` est invoquée sur l'entrée obtenue afin d'établir une liaison avec l'objet Active Directory référencé. À ce stade, soit nous avons vérifié qu'un objet existe, soit nous pouvons interroger l'objet retourné par la fonction pour plus d'informations.

Dans l'extrait de code suivant, les variables `$ScriptName` et `$ScriptUsage` sont définies. Elles seront employées plus loin pour afficher les informations d'utilisation du script :

```
##################################################
# Code principal.
##################################################
#--------------------
# Définir les variables de configuration.
#--------------------
$ScriptName = "IsGroupMember"
$ScriptUsage = "Ce script vérifie si des utilisateurs sont membres du groupe
indiqué."
```

Outre l'affichage des informations sur l'utilisation du script, les fonctions `Get-ScriptHeader` et `Show-ScriptUsage` présentent également une aide lorsque les utilisateurs indiquent en premier argument l'une des chaînes suivantes : `-?`, `-h` et `-help`. Pour cela, le script se sert de l'opérateur de comparaison de correspondance : `$args[0] -match '-(\?|(h|(help)))'` :

```
#--------------------
# Vérifier les paramètres obligatoires.
#--------------------
if ($args[0] -match '-(\?|(h|(help)))'){
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage
    Return
    }
```

Les deux portions de code suivantes sont des méthodes de vérification des paramètres obligatoires du script. Les exemples précédents s'appuyaient sur le mot clé `throw` lors de la définition d'un paramètre (avec le mot clé `param`) et indiquaient ainsi les paramètres requis. Dans ce script, à la place du mot clé `throw`, nous vérifions la présence du paramètre obligatoire et, s'il est absent, nous affichons aux utilisateurs un message d'aide leur indiquant qu'ils ont oublié de préciser un argument. Nous pouvons également leur afficher des informations à propos des paramètres, de leur utilisation et des exemples d'arguments.

Enfin, `Get-ScriptHeader` affiche l'en-tête du script. Les utilisateurs ont ainsi la confirmation qu'ils exécutent le bon script. Par ailleurs, si le script demande plusieurs heures avant de se terminer, l'en-tête contient la date et l'heure de lancement.

L'objectif de cette fonction est d'améliorer la convivialité du script, une qualité souvent négligée dans les scripts et les interfaces en ligne de commande. L'absence de facilité d'utilisation fait partie des raisons qui ont maintenu les administrateurs de systèmes Windows à l'écart des scripts et des CLI pour la gestion de leurs environnements Windows. L'équipe de développement de PowerShell a reconnu les problèmes de convivialité des langages de scripts précédents et a fait un effort particulier pour créer un shell et un langage pas seulement destiné aux développeurs, mais également aux utilisateurs. Lorsque vous développez des scripts, gardez les utilisateurs en perspective. Comme nous l'avons expliqué au Chapitre 5, "Suivre les bonnes pratiques", l'automation n'est qu'une partie du développement d'un script.

```
if (!$GroupName){
    write-host
    write-host "Veuillez préciser le nom du groupe !" -Foregroundcolor Red
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage
    Return
    }

if (!$ImportFile){
    write-host
    write-host "Veuillez préciser le nom du fichier CSV à importer !" `
    -Foregroundcolor Red
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage
    Return
```

Dans le code suivant, la fonction `Get-ScriptHeader` indique à l'utilisateur du script que la partie automation vient de démarrer :

```
#--------------------
# Début du script.
#--------------------
write-host
Get-ScriptHeader $ScriptName $ScriptUsage
write-host
```

Le script doit ensuite vérifier qu'il existe une connexion valide au domaine. Pour cela, il invoque `Get-CurrentDomain`. S'il n'existe aucune connexion valide, le script se termine et retourne son code d'état. Dans le cas contraire, il poursuit son exécution et affiche le nom de domaine sur la console :

```
.{
    trap{write-host `t "[ERREUR]" -Foregroundcolor Red;
        throw write-host $_ -Foregroundcolor Red;
        Break}

    write-host "Vérification de la connexion au domaine" -NoNewLine

    # Tester la connexion au domaine.
    $Domain = Get-CurrentDomain

    # Afficher le nom du domaine.
    write-host `t $Domain.Name -Foregroundcolor Green
}
```

Nous vérifions ensuite le nom du groupe dans la variable `$GroupName`. Pour cela, le script se sert de la fonction `Get-ADObject`. Elle se connecte à Active Directory et recherche le groupe par son nom. Si la fonction retourne un objet, le nom du groupe est valide. Sinon, le nom du groupe est considéré comme invalide et le script se termine :

```
write-host "Vérification du nom du groupe" -NoNewLine

# Obtenir le groupe.
$Group = Get-ADObject "sAMAccountName" $GroupName "Group"

if (!$Group){
    write-host `t "Invalide !" -Foregroundcolor Red
    write-host
    Break
    }
else{
    write-host `t "[OK]" -Foregroundcolor Green
    }
```

La dernière vérification concerne la validité du nom du fichier à importer. Pour cela, nous utilisons l'applet de commande `Test-Path` avec la variable `$ImportFile` :

```
write-host "Vérification du fichier à importer" -NoNewLine

if (!(test-path $ImportFile -pathType leaf)){
    write-host `t "Fichier invalide !" -Foregroundcolor Red
    write-host
    Break
    }
else{
    write-host `t "[OK]" -Foregroundcolor Green
    }
```

Dans le code suivant, le script termine la vérification d'appartenance de l'utilisateur à un groupe. Comme nous l'avons expliqué précédemment, en fonction de la validité de l'utilisateur dans Active Directory et de son appartenance au groupe indiqué (`$GroupName`), le script complète l'objet d'utilisateur dans la collection `$Users` :

```
#--------------------
# Vérifier l'appartenance de chaque utilisateur au groupe.
#--------------------
$Users = import-csv $ImportFile

foreach ($User in $Users){
    &{
        $sAMAccountName = $User.sAMAccountName

        $ADUser = Get-ADObject "sAMAccountName" $sAMAccountName "User"

        if ($ADUser){
            [string]$DN = $ADUser.distinguishedName

            $IsMember = $Group.Member | `
                where {$_ -eq $DN}
            if ($IsMember){
                add-member -inputObject $User -membertype noteProperty `
                    -name "IsMember" -value "Oui"
                }
```

```
            else{
                add-member -inputObject $User -membertype noteProperty `
                    -name "IsMember" -value "Non"
                }
            }
        else{
            # Et si utilisateur n'existe pas ?
            add-member -inputObject $User -membertype noteProperty `
                -name "IsMember" -value "Inexistant"
            }
    }
    }
```

Au cours des chapitres précédents, nous avons expliqué que l'opérateur d'appel `&` exécute un bloc de code du script dans sa propre portée. Lorsqu'il se termine, sa portée est détruite avec tout ce qu'elle contient.

Dans l'exemple de code précédent, l'opérateur `&` nous permet de réutiliser les noms de variables sans nous préoccuper des anciennes données. Par exemple, lorsque nous sortons de la boucle `For`, les noms de variables `$sAMAccountName` et `$ADUser` restent des objets valides. Pour ne pas nous servir par mégarde ces anciens objets, nous utilisons l'opérateur d'appel `&` afin de détruire l'objet après l'exécution du bloc de code.

L'extrait de code suivant affiche le contenu de la collection `$Users` sur la console. Si un fichier d'exportation a été précisé, le contenu y est alors écrit au format CSV en utilisant l'applet de commande `Export-CSV`.

```
if (!$ExportFile){
    $Users
    }
else{
    write-host
    write-host "Exportation des données dans : " -NoNewLine
    $Users | export-csv $ExportFile
    write-host "$ExportFile" -Foregroundcolor Green
    write-host
    }
```

En résumé

Au long de ce chapitre, nous avons examiné l'interaction entre PowerShell et ADSI, ainsi que l'accès à des objets Active Directory. Nous avons vu que PowerShell offre les mêmes interfaces de gestion d'Active Directory que WSH, si ce n'est plus, grâce à sa compatibilité .NET Framework. Par ailleurs, nous avons étudié un script opérationnel qui détermine si des utilisateurs sont membres d'un groupe. Comme avec WSH, il s'agit seulement de l'un des nombreux types de scripts de gestion d'Active Directory que nous pouvons développer avec PowerShell.

III

Utiliser PowerShell pour les besoins d’automation

10

Utiliser PowerShell en situation réelle

Dans ce chapitre

- Le script PSShell.ps1
- Le script ChangeLocalAdminPassword.ps1

Ce chapitre illustre la puissance de PowerShell dans la gestion des environnements Windows. Nous passons en revue deux scripts PowerShell utilisés pour la gestion des systèmes. Le premier, `PSShell.ps1`, se charge des interactions entre l'utilisateur et le bureau de Windows en créant un remplaçant contrôlé, sécurisé et séduisant du bureau. Le second, `ChangeLocalAdminPassword.ps1`, gère les mots de passe de l'administrateur local sur des serveurs d'un domaine Active Directory.

Ces scripts montrent comment satisfaire les besoins de gestion des systèmes dans une entreprise. Lors de l'étude de chaque script, vous apprendrez de nouveaux concepts PowerShell et verrez comment les appliquer à vos propres besoins.

Le script PSShell.ps1

`PSShell.ps1` peut être employé comme un shell sécurisé pour les kiosques informatiques. Vous en trouverez une copie dans le dossier `Scripts\Chapitre 10\PSShell` et en téléchargement sur le site **www.pearsoneducation.fr**. Ce script exige que vous compreniez comment se passe le remplacement du shell de Windows. Lisez bien les sections suivantes à propos des composants du script afin que vous sachiez comment le déployer et l'utiliser efficacement. Cependant, nous devons commencer par présenter les raisons de ce script.

companyabc.com fabrique des processeurs pour le grand public et le gouvernement américain. Le personnel qui travaille sur les processeurs destinés au gouvernement doit posséder certaines autorisations de sécurité, et toutes les données associées à la fabrication de ces processeurs doivent être sécurisées afin d'éviter leur diffusion auprès d'entités non autorisées, tant internes qu'externes à la société.

Ces contraintes sécuritaires posent un problème à companyabc.com. Son service informatique doit mettre en œuvre des procédures métier différentes pour les contrats grand public et ceux gouvernementaux. Par ailleurs, le directeur de companyabc.com a souhaité un système centralisé, autrement dit que tous les utilisateurs aient accès aux données et aux applications depuis n'importe quel ordinateur. Ce fonctionnement complexifie d'autant les mesures de sécurité.

Pour satisfaire ces contraintes, le service informatique a déployé des fermes de serveurs WTS (*Windows Terminal Services*). Les utilisateurs qui travaillent pour le grand public accèdent à des serveurs ayant un niveau de sécurité faible, tandis que ceux qui travaillent sur les contrats gouvernementaux accèdent à des serveurs WTS isolés des autres utilisateurs et avec un niveau de sécurité très élevé.

Il a également été décidé que les connexions aux fermes WTS se feraient *via* des clients légers, afin d'accélérer le déploiement et de maîtriser totalement la sécurité des accès et des données. Cependant, même si companyabc.com disposait du budget pour mettre en place les fermes WTS, les fonds permettant d'acheter les clients légers et les logiciels adéquats pour tous les utilisateurs n'étaient pas disponibles. Pour compliquer les choses, la société était récemment passée à des systèmes Windows XP. Par ailleurs, le matériel bureautique venait d'être acheté et devrait donc être utilisé encore quelques années avant d'être remplacé.

Pour ne pas dépasser le budget alloué, le service informatique a recherché une solution bon marché permettant de convertir les systèmes Windows XP existants en clients légers. L'un des administrateurs système a lu un article technique concernant le remplacement du shell de Windows afin de convertir un bureau Windows XP en un kiosque sécurisé, mais cela impliquait le remplacement de l'Explorateur Windows par Internet Explorer pour créer l'interface du kiosque. Si cette méthode convient parfaitement à un simple kiosque de navigation sur le Web, le service informatique avait quant à lui besoin d'un contrôle total sur le shell d'interface.

Pour résoudre ce problème, le service informatique a décidé d'utiliser PowerShell et sa compatibilité avec .NET Windows Forms pour offrir un remplacement personnalisable du shell pour l'Explorateur Windows. Après quelques développements et tests, la solution aux besoins de clients légers de companyabc.com a été un hybride entre plusieurs composants différents. Ces composants incluent un shell de remplacement de Windows, qui utilise `cmd.exe`

comme shell de base, et un script PowerShell, qui s'appuie sur Windows Forms pour présenter un bureau sécurisé de type Explorateur Windows aux utilisateurs. Les sections suivantes détaillent les composants de `PSShell.ps1` (nommé Kiosque PSShell).

Composant 1 : shell de remplacement

Le premier composant de Kiosque PSShell est le shell de remplacement. Par défaut, Windows utilise `explorer.exe` comme interface avec le système d'exploitation. Cependant, ce shell n'est absolument pas nécessaire au fonctionnement de Windows. Parfois, les utilisateurs souhaitent disposer de fonctionnalités non fournies par l'Explorateur Windows ou veulent réduire les fonctionnalités afin d'améliorer la sécurité, ce qui est le cas de companyabc.com.

Les utilisateurs et les administrateurs de Windows peuvent modifier `explorer.exe` ou le remplacer par un autre shell (ce qui n'est pas forcément pris en charge par Microsoft). Cette procédure se nomme "remplacement du shell de Windows" (*Windows Shell Replacement*). Différents shells peuvent être employés comme remplaçants. Cela va des shells graphiques, comme Internet Explorer (`iexplore.exe`), Geoshell et LiteStep, aux shells en ligne de commande, comme `cmd.exe`, `command.com` et même PowerShell.

Il existe deux méthodes pour remplacer `explorer.exe`. La première consiste à modifier le Registre de Windows et à préciser le shell de remplacement dans la valeur `Shell` de la clé `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon`.

Pour companyabc.com, la modification du Registre sur chaque machine Windows XP n'était pas envisageable. Par ailleurs, le remplacement du shell pour l'intégralité du système n'est pas prudent. Supposons que des techniciens doivent se connecter à des machines pour effectuer une opération de maintenance. Si le shell par défaut du système a été remplacé par le biais de la méthode du Registre, les techniciens n'ont d'autres choix que d'utiliser le shell de remplacement limité car la modification a été effectuée pour tous les utilisateurs. Même si le Registre permet d'activer un shell de remplacement en fonction des utilisateurs, cette solution n'est pas une méthode conviviale et efficace de déployer des shells, comme l'a découvert le service informatique de companyabc.com.

La seconde méthode permettant de remplacer `explorer.exe` passe par un paramètre de GPO (*Group Policy Object*) baptisé Interface utilisateur personnalisée. Il permet de préciser le shell des utilisateurs lorsqu'ils ouvrent une session sur une machine. L'utilisation des GPO a l'avantage d'offrir la centralisation et la facilité de gestion. Par ailleurs, nous pouvons définir différentes configurations de shells en fonction de l'utilisateur et non de la machine sur laquelle il se connecte. Puisque companyabc.com recherche ce type de contrôle, le service informatique a choisi la méthode GPO pour la gestion de Kiosque PSShell. Les étapes suivantes détaillent cette solution.

Étape 1 : créer le GPO du Kiosque PSShell sécurisé

Pour créer le GPO de configuration du shell de remplacement de Windows, procédez comme suit :

1. À l'aide de la console de gestion des stratégies de groupe (GPMC, *Group Policy Management Console*), créez un GPO nommé **GPO Bureau Kiosque PSShell**.
2. Ensuite, désactivez les paramètres de configuration ordinateur.
3. Retirez les utilisateurs authentifiés des paramètres de Filtrage de sécurité.
4. Dans la console Utilisateurs et ordinateurs Active Directory, créez un groupe de domaine local nommé **GPO Bureau Kiosque PSShell - Appliquer** et ajoutez un utilisateur de test au groupe.
5. Ajoutez le groupe GPO Bureau Kiosque PSShell - Appliquer aux paramètres de filtrage de sécurité du GPO Bureau Kiosque PSShell.
6. Enfin, liez le GPO Bureau Kiosque PSShell à l'unité d'organisation (UO) de premier niveau qui contient tous vos comptes d'utilisateurs et vérifiez que l'ordre des liaisons des autres GPO n'écrase pas le GPO Bureau Kiosque PSShell.

INFO

La liaison du GPO Bureau Kiosque PSShell à l'UO de premier niveau qui contient les comptes d'utilisateurs suppose qu'il n'existe aucun autre GPO lié aux UO enfants qui pourraient écraser celui-ci. Par ailleurs, le GPO est appliqué à un groupe d'utilisateurs et non à un groupe de machines afin d'empêcher que les utilisateurs ayant des autorisations de sécurité élevées aient un bureau non sécurisé.

Étape 2 : configurer le remplacement du shell de Windows

Ensuite, configurez les paramètres de remplacement du shell de Windows en procédant comme suit :

1. Dans la console gestion des stratégies de groupe, ouvrez le GPO Bureau Kiosque PSShell.
2. Développez Configuration utilisateur, Modèles d'administration et Système. Ensuite, sélectionnez le paramètre Interface utilisateur personnalisée.
3. Cliquez du bouton droit sur Interface utilisateur personnalisée et choisissez Propriétés.
4. Dans la boîte de dialogue Propriétés d'Interface utilisateur personnalisée, cochez la case Activé, saisissez **cmd /c "C:\PSShell\Launch.bat"** dans le champ de texte Nom du fichier d'interface (voir Figure 10.1) puis cliquez sur OK.

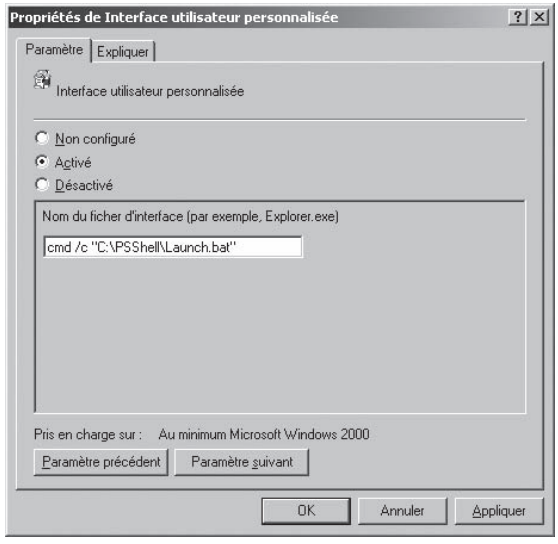


Figure 10.1

Boîte de dialogue Propriétés d'Interface utilisateur personnalisée.

En fixant le nom du fichier d'interface à `cmd`, nous obligeons Windows à utiliser `cmd.exe` comme shell de remplacement. L'option `/c` demande à `cmd` d'exécuter le fichier de commandes `C:\PSShell\Launch.bat` et de s'arrêter, ce qui ferme la fenêtre de `cmd` après la fin de l'exécution de `Launch.bat`.

INFO

En utilisant le chemin `C:\PSShell`, nous supposons que les fichiers de Kiosque PSShell ont été copiés dans ce dossier sur la machine du client. Cependant, ce choix n'est en rien figé et ils peuvent être placés ailleurs, par exemple sur un partage réseau Windows.

Composant 2 : PSShell.exe

Vous vous demandez peut-être pourquoi nous avons utilisé `cmd` comme shell de remplacement à la place de PowerShell. Lorsque nous exécutons un script PowerShell, il est impossible de le faire sans afficher la console PowerShell. Si `explorer.exe` est remplacé par PowerShell, le bureau résultant contient cette console.

Cependant, companyabc.com souhaite que les utilisateurs disposent d'un bureau analogue à celui de `explorer.exe` et non d'un bureau qui propose la console PowerShell. La solution implique le deuxième composant, nommé `PSShell.exe`. `PSShell.exe` est une application Windows écrite en C# qui masque la console PowerShell lors de l'exécution de `PSShell.ps1`.

L'extrait suivant montre le code source de cette application :

```
using System;
using System.Diagnostics;

namespace PSShell
{
    static class Program
    {
        static void Main()
        {
            Process Process = new Process ();

            Process.StartInfo.FileName = "powershell.exe ";
            Process.StartInfo.Arguments = "-Command \"C:\\PSShell\\PSShell.ps1\"";
            Process.StartInfo.CreateNoWindow = true;
            Process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
            Process.Start();
        }
    }
}
```

Pour masquer la console PowerShell, `PSShell.exe` se sert de la classe .NET `System.Diagnostics.Process`. En utilisant cette classe avec l'énumération .NET `ProcessWindowStyle`, nous pouvons préciser comment la fenêtre d'un processus apparaît au moment de son démarrage. Le style (apparence) peut être `Hidden`, `Normal`, `Minimized` ou `Maximized`. Dans notre exemple, nous voulons que le style de la fenêtre de PowerShell soit fixé à `Hidden`. Une fois lancé le processus PowerShell à l'aide de la méthode `Start()` et des arguments adéquats, Windows n'affiche pas la console PowerShell.

INFO

Une fois encore, le chemin `C:\PSShell` dans le code source de `PSShell.exe` n'est qu'une suggestion. Si vous modifiez le chemin de déploiement de Kiosque PSShell, vous devez actualiser le code et compiler un nouvel exécutable. Cependant, si le langage C# vous est familier, une meilleure solution consiste à modifier `PSShell.exe` afin qu'il prenne des arguments pour définir le chemin du script `PSShell.ps1`.

Pour bien comprendre pourquoi `cmd` est utilisé comme shell de remplacement, n'oubliez pas que `PSShell.exe` n'est pas un shell, mais une application dont le seul objectif est de supprimer la console PowerShell lors de l'exécution d'un script. Cet exécutable est

également nécessaire pour lancer PowerShell et exécuter `PSShell.ps1` avec la console masquée. Cependant, pour démarrer `PSShell.exe`, nous devons l'appeler depuis un autre shell, comme `cmd`. Le nom du fichier d'interface saisi dans le paramètre Interface utilisateur personnalisée précise le fichier de commandes nommé `Launch.bat`. Il est utilisé pour lancer `PSShell.exe`.

En résumé, `cmd` exécute `Launch.bat`, qui démarre `PSShell.exe`. À son tour, `PSShell.exe` lance PowerShell, qui exécute finalement le script `PSShell.ps1`. Tout cela est un peu compliqué mais indispensable pour répondre à une déficience de fonctionnalités dans PowerShell. Grâce à cette solution, nous pouvons générer un bureau sécurisé avec PowerShell.

Composant 3 : PSShell.ps1

Le dernier composant de Kiosque PSShell est le script `PSShell.ps1`. Il crée le bureau du kiosque pour les utilisateurs connectés. Pour cela, il utilise un formulaire, grâce à la compatibilité entre PowerShell et .NET Windows Forms. Le seul objectif de ce script est de donner aux utilisateurs l'illusion d'être devant le bureau par défaut de Windows, alors qu'ils sont en réalité dans un bureau personnalisé aux fonctionnalités limitées.

Kiosque PSShell détermine ce qui est présenté aux utilisateurs et les programmes qu'ils peuvent lancer depuis le bureau. companyabc.com souhaite que les utilisateurs puissent effectuer des tâches suivantes dans un bureau sécurisé :

- lancer le client Bureau à distance (RDP, *Microsoft Remote Desktop*), qui est configuré pour se connecter à une ferme WTS sécurisée ;
- démarrer une version limitée (par GPO) d'Internet Explorer qui va sur le site de courrier Web de la société ;
- se déconnecter de Kiosque PSShell lorsque leur travail est terminé.

Le premier extrait de code contient l'en-tête du script `PSShell.ps1`. Il fournit des informations sur le rôle du script, sa date de dernière mise à jour et son auteur :

```
##################################################
# PSShell.ps1
# Ce script est un shell de remplacement pour explorer.exe.
#
# Créé le : 17/10/2006
# Auteur  : Tyson Kopczynski
##################################################
```

Le code suivant contient deux longues instructions complexes qui font intervenir la classe .NET `System.Reflection.Assembly` :

```
$Null=[System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
$Null=[System.Reflection.Assembly]::LoadWithPartialName("System.Drawing")
```

Ces deux instructions sont indispensables car PowerShell charge uniquement quelques assemblages (*assembly*) dans son `AppDomain`. Par exemple, si nous essayons de créer un objet Windows Forms avec l'applet de commande `New-Object`, nous obtenons l'erreur suivante :

```
PS C:\> $Form = new-object System.Windows.Forms.Form
New-Object : Le type [System.Windows.Forms.Form] est introuvable : vérifiez que
 l'assembly dans lequel il se trouve est chargé.
Au niveau de ligne : 1 Caractère : 19
+ $Form = new-object  <<<< System.Windows.Forms.Form
PS C:\>
```

Pour pouvoir utiliser la classe `System.Windows.Forms.Form`, nous devons tout d'abord charger l'assemblage dans PowerShell à l'aide de la méthode `LoadWithPartialName()`. Les assemblages des DLL de type .NET incluses avec les SDK Microsoft, celles des fournisseurs tiers ou nos propres DLL doivent également être chargés dans PowerShell. Par exemple, supposons que nous ayons développé une DLL .NET pour gérer l'application xyz. Pour utiliser cette DLL dans PowerShell, nous utilisons la méthode `LoadFrom()` ou `LoadFile()` de la classe `System.Reflection.Assembly` :

```
PS C:\> [System.Reflection.Assembly]::LoadFrom("C:\Outils\maPremiere.dll")
0
PS C:\>
```

INFO

Microsoft a déclaré la méthode `LoadWithPartialName()` obsolète. Elle est remplacée par la méthode `Load()`, qui est conçue pour éviter des liaisons partielles lors du chargement d'assemblages .NET. L'utilisation de la méthode `Load()` demande un travail plus important. Cependant, si les implications d'une liaison partielle (comme l'échec de votre script) ne vous préoccupent pas, vous pouvez toujours utiliser `LoadWithPartialName()` jusqu'à ce qu'elle soit retirée de .NET Framework.

À présent que les assemblages nécessaires aux objets Windows Forms ont été chargés, l'étape suivante consiste à terminer la configuration de l'environnement d'exécution du script. Pour cela, nous définissons un ensemble de chaînes de commandes qui définissent les applications que les utilisateurs sont autorisés à lancer depuis le bureau de Kiosque PSShell. Nous reviendrons sur ces chaînes de commandes plus loin :

```
# Chaînes des commandes autorisées.
$LaunchIE = {$IE = new-object -com InternetExplorer.Application; `
    $IE.navigate("webmail.companyabc.com"); $IE.visible = $True; $IE}
$LaunchRemoteDesktop = {mstsc /v:earth.companyabc.com /f}
```

Ensuite, nous créons un espace d'exécution (Runspace) de PowerShell, comme le montre le code suivant :

```
#--------------------
# Créer un espace d'exécution.
#--------------------
# Pour plus d'informations sur les espaces d'exécution, voir
# http://msdn2.microsoft.com/en-us/library/ms714459.aspx

$Runspace =
    [System.Management.Automation.Runspaces.RunspaceFactory]::
CreateRunspace()
$RunspaceInvoke =
    new-object System.Management.Automation.RunspaceInvoke($Runspace)
$Runspace.Open()
```

Ce code montre un espace d'exécution PowerShell, qui est représenté par l'espace de noms `System.Management.Automation.Runspaces`. Un **espace d'exécution** est une abstraction de l'environnement d'exécution de PowerShell qui permet à une application hôte d'exécuter des commandes PowerShell. Même si `powershell.exe` est une application hôte qui utilise son propre espace d'exécution pour traiter les commandes, les espaces d'exécution sont d'autant plus intéressants lorsqu'ils sont employés dans des applications en dehors de PowerShell.

Les espaces d'exécution sont indispensables à PowerShell, mais ils ont été développés principalement pour donner aux autres applications un moyen simple d'appeler l'environnement d'exécution PowerShell et d'exécuter des commandes. En un sens, l'objet Windows Form créé par `PSShell.ps1` est une application, et son interaction avec un espace d'exécution PowerShell

pour effectuer des tâches est donc sensée. En exploitant les espaces d'exécution de PowerShell, nous n'avons pas besoin d'ajouter une logique à l'objet Windows Form pour qu'il réalise des tâches pour le compte des utilisateurs.

La création d'un espace d'exécution (`$Runspace`) pour l'objet Windows Form consiste simplement à invoquer la méthode `CreateRunspace()` de la classe PowerShell `System.Management.Automation.Runspaces.RunspaceFactory`. Ensuite, nous créons un objet `RunspaceInvoke` qui permet au Windows Form d'exécuter des commandes par le biais de l'espace d'exécution. Enfin, nous ouvrons cet espace à l'aide de la méthode `Open()` afin qu'il puisse être utilisé par l'objet Windows Form.

Une fois l'espace d'exécution défini, nous devons construire le formulaire (fenêtre) lui-même, comme le montre l'extrait de code suivant. La section "Définir les images" crée un ensemble d'objets `Drawing.Image`. Ils seront utilisés par la suite dans le formulaire pour représenter des éléments comme le menu Démarrer du Kiosque PSShell et les icônes des applications. Ensuite, à la section "Créer la fenêtre", l'objet formulaire est créé avec un ensemble de propriétés prédéfinies qui le font ressembler au bureau par défaut de Windows.

```
#--------------------
# Définir les images.
#--------------------
$ImagePath = Split-Path -Parent $MyInvocation.MyCommand.Path
$ImgStart = [Drawing.Image]::FromFile("$Imagepath\Images\Start.png")
$ImgRDP = [Drawing.Image]::FromFile("$Imagepath\Images\RDP.png")
$ImgIE = [Drawing.Image]::FromFile("$Imagepath\Images\IE.png")

#--------------------
# Créer la fenêtre.
#--------------------
$Form = new-object System.Windows.Forms.Form
$Form.Size = new-object System.Drawing.Size @(1,1)
$Form.DesktopLocation = new-object System.Drawing.Point @(0,0)
$Form.WindowState = "Maximized"
$Form.StartPosition = "CenterScreen"
$Form.ControlBox = $False
$Form.FormBorderStyle = "FixedSingle"
$Form.BackColor = "#647258"
```

La fenêtre est créée, mais avant de l'activer et de l'afficher à l'utilisateur, nous devons ajouter les éléments du menu. Le code suivant ajoute plusieurs `MenuItems` au `ToolStripMenu` qui joue le rôle du menu Démarrer pour le bureau de Kiosque PSShell :

```
#--------------------
# Construire le menu.
#--------------------
$MenuStrip = new-object System.Windows.Forms.MenuStrip
$MenuStrip.Dock = "Bottom"
$MenuStrip.BackColor = "#292929"

# Menu Démarrer.
$StartMenuItem = new-object System.Windows.Forms.ToolStripMenuItem("")
$StartMenuItem.Padding = 0
$StartMenuItem.Image = $ImgStart
$StartMenuItem.ImageScaling = "None"

# Élément de menu 1.
$MenuItem1 = new-object System.Windows.Forms.ToolStripMenuItem("&Webmail")
$MenuItem1.Image = $ImgIE
$MenuItem1.ImageScaling = "None"
$MenuItem1.add_Click({$RunspaceInvoke.Invoke($LaunchIE)})

$StartMenuItem.DropDownItems.Add($MenuItem1)

# Élément de menu 2
$MenuItem2 = new-object System.Windows.Forms.ToolStripMenuItem(`
   "&Bureau à distance")
$MenuItem2.Image = $ImgRDP
$MenuItem2.ImageScaling = "None"
$MenuItem2.add_Click({$RunspaceInvoke.invoke($LaunchRemoteDesktop)})

$StartMenuItem.DropDownItems.Add($MenuItem2)

# Élément de menu 3.
$MenuItem3 = new-object System.Windows.Forms.ToolStripMenuItem(`
   "&Fermer la session")
$MenuItem3.add_Click({`
    $RunspaceInvoke.invoke({Get-WmiObject Win32_OperatingSystem | `
    foreach-object {$_.Win32Shutdown(0)}})})

$StartMenuItem.DropDownItems.Add($MenuItem3)
```

Les éléments de menu ajoutés par le code précédent permettent aux utilisateurs de lancer des applications ou de fermer le bureau de Kiosque PSShell. À chaque élément est affecté un gestionnaire d'événements de clic qui utilise l'objet `$RunspaceInvoke` et sa méthode `invoke()` pour exécuter la commande PowerShell indiquée. La liste suivante décrit l'action réalisée par chaque élément de menu :

- `$MenuItem1`. Utilise la commande indiquée dans la variable `$LaunchIE` pour lancer Internet Explorer.
- `$MenuItem2`. Utilise la commande indiquée dans la variable `$LaunchRemoteDesktop` pour démarrer `mstsc.exe` (le client du bureau à distance de Microsoft).
- `$MenuItem3`. Utilise l'applet de commande `Get-WmiObject` pour fermer la session Windows.

Enfin, le script doit activer la fenêtre et l'afficher à l'utilisateur. Pour cela, nous utilisons la méthode `ShowDialog()` :

```
#--------------------
# Afficher la fenêtre.
#--------------------
$MenuStrip.Items.Add($StartMenuItem)
$Form.Controls.Add($MenuStrip)
$Form.Add_Shown({$Form.Activate()})
$Form.ShowDialog()
```

Tout réunir

Une fois le GPO Bureau Kiosque PSShell configuré et prêt à être appliqué aux utilisateurs, l'étape suivante consiste à déployer les fichiers de Kiosque PSShell sur les systèmes servant de clients légers sécurisés :

- `Launch.bat`. Le fichier de commandes qui permet de lancer `PSShell.exe`.
- `PSShell.exe`. L'application C# qui sert à exécuter le script `PSShell.ps1`.
- `PSShell.ps1`. Le script PowerShell qui crée le Kiosque PSShell.
- Dossier `Images`. Le dossier qui contient les images utilisées par le bureau de Kiosque PSShell.

Comme nous l'avons indiqué précédemment, Kiosque PSShell est configuré pour le chemin `C:\PSShell`. Par conséquent, après avoir déployé ces fichiers dans ce dossier sur chaque machine, vous pouvez placer les utilisateurs qui ont besoin d'un Bureau sécurisé dans le

groupe GPO Bureau Kiosque PSShell - Appliquer. La Figure 10.2 montre le Bureau de Kiosque PSShell avec trois éléments de menu.

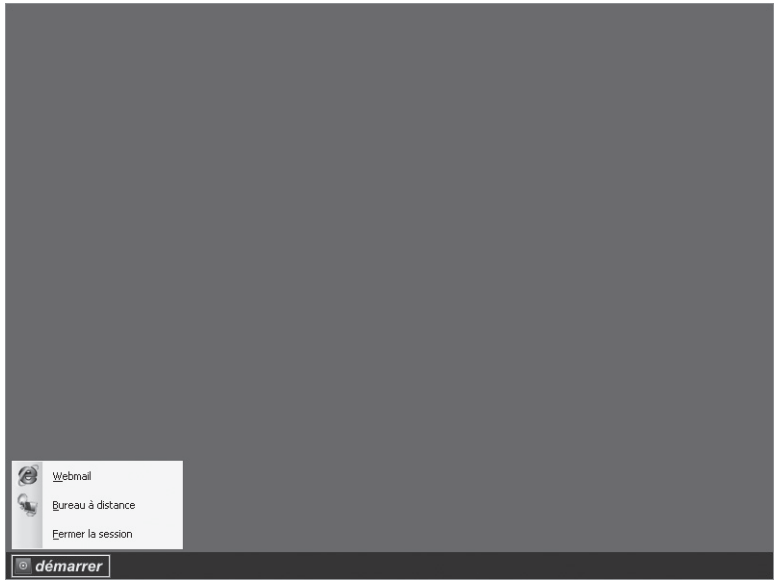


Figure 10.2
Le Bureau de Kiosque PSShell.

Le script ChangeLocalAdminPassword.ps1

Le script `ChangeLocalAdminPassword.ps1` a été développé pour s'occuper d'une tâche demandant beaucoup de temps aux administrateurs système. Il s'agit de la modification du mot de passe de l'administrateur local, que ce soit de manière routinière (administration planifiée) ou forcée (suite à une attaque du réseau). Cette modification fait partie des toutes premières activités de gestion des systèmes, et les administrateurs la négligent fréquemment car elle est fastidieuse.

companyabc.com dispose d'une ferme de cinq cents serveurs sous Windows Server 2003. Conformément aux pratiques sécuritaires de cette entreprise, le service informatique a essayé de modifier régulièrement le mot de passe de l'administrateur local sur les cinq cents serveurs, en général une fois par mois ou lorsqu'un administrateur système quittait la société. À cause du temps et des efforts demandés par ces changements, le service informatique a eu tendance

à ne pas respecter les dates prévues. Il a fini par ne plus modifier le mot de passe de l'administrateur local, ce qui a rapidement conduit à un incident de sécurité sérieux : une entité externe a exploité cette défaillance dans le respect des bonnes pratiques de gestion des mots de passe pour prendre le contrôle de quelques serveurs de companyabc.com et demander une rançon pour rendre l'accès à ces systèmes.

Suite à cet incident, le service informatique a recherché une manière de modifier rapidement et en masse le mot de passe de l'administrateur local. Il a été décidé d'employer un script d'automation qui crée une liste des serveurs dans une UO déterminée et qui se connecte à chaque serveur pour modifier le mot de passe de l'administrateur local. Le script `ChangeLocalAdminPassword.ps1` est le fruit de ce travail.

Vous trouverez une copie de ce script dans le dossier `Scripts\Chapitre 10\ChangeLocalAdminPassword` et en téléchargement sur le site **www.pearsoneducation.fr**. Pour l'exécuter, il faut définir le paramètre `OUDN`. Son argument doit être le `distinguishedName` de l'UO qui contient les serveurs dont le mot de passe de l'administrateur local doit être modifié. Voici la commande qui permet d'exécuter le script `ChangeLocalAdminPassword.ps1` :

```
PS D:\Scripts> .\ChangeLocalAdminPassword.ps1 "OU=Servers,OU=Managed Objects
,DC=companyabc,DC=com"
```

Les Figures 10.3 et 10.4 illustrent l'exécution de `ChangeLocalAdminPassword.ps1`.

Voici la suite des opérations réalisées par le script `ChangeLocalAdminPassword.ps1` :

1. Il charge le fichier de bibliothèque `LibraryCrypto.ps1`, qui propose une fonction permettant de générer des mots de passe aléatoires.
2. Il crée un nouvel objet `DataTable` (`$ServersTable`) à l'aide de la classe .NET `System.Data.DataSet`. Cet objet sera utilisé pour enregistrer des informations d'état à propos des machines de l'UO précisée.
3. Il crée un journal d'erreurs nommé `ChangeLocalAdminPassword_Errors.log` en utilisant l'applet de commande `Out-File`. Ce journal permet de présenter aux utilisateurs des informations détaillées concernant les erreurs.
4. Il se connecte au domaine d'ouverture de session actuel en invoquant la fonction `Get-CurrentDomain`. À partir de l'objet retourné par cette fonction, il affiche le nom du domaine sur la console PowerShell. Si la connexion échoue, le script se termine.
5. Ensuite, le script vérifie que l'UO indiquée existe dans le domaine courant en appelant la fonction `Get-ADObject`. Si l'UO n'est pas valide, il se termine.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\ChangeLocalAdminPassword.ps1 "OU=Servers,OU=Managed Objects,DC=
companyabc,DC=com"

 Modification du mot de passe - Administrateur
    Veuillez patienter...

r certains machines.
# Utilisateur :  tyson
# Date :         Tue Oct 30 13:43:16     2007
##########################################

Vérification de la connexion au domaine  taosage.internal
Vérification du nom de l'UO      [OK]

Question :
Dois-je générer un mot de passé aléatoire ?
[O] Oui  [N] Non  [?] Aide (la valeur par défaut est « O ») : N

Veuillez saisir le mot de passe: *********

Obtention des informations sur les serveurs      [OK]
Obtention des informations d'état        [OK]
Modification des mots de passe_
```

Figure 10.3

Modifier le mot de passe de l'administrateur local.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\ChangeLocalAdminPassword.ps1 "OU=Servers,OU=Managed Objects,DC=
companyabc,DC=com"

##########################################
# Script         ChangeLocalAdminPassword.ps1
# Usage :        Ce script modifie le mot de passe de l'administrateur local su
r certains machines.
# Utilisateur :  tyson
# Date :         Tue Oct 30 13:43:16     2007
##########################################

Vérification de la connexion au domaine  taosage.internal
Vérification du nom de l'UO      [OK]

Question :
Dois-je générer un mot de passé aléatoire ?
[O] Oui  [N] Non  [?] Aide (la valeur par défaut est « O ») : N

Veuillez saisir le mot de passe: *********

Obtention des informations sur les serveurs      [OK]
Obtention des informations d'état        [OK]
Modification des mots de passe   [OK]

Le script est terminé !
Consultez le fichier ChangeLocalAdminPassword.ps1_Errors.log en cas d'erreurs.
PS C:\Scripts>
```

Figure 10.4

Terminaison du script ChangeLocalAdminPassword.ps1.

6. Il utilise ensuite les fonctions `Set-ChoiceMesssage` et `New-PromptYesNo` pour demander à l'utilisateur s'il souhaite un mot de passe généré aléatoirement ou le préciser. Pour les mots de passe aléatoires, le script appelle la fonction `New-RandomPassword` de la bibliothèque `LibraryCrypto.ps1`, en précisant la longueur du mot de passe. Il est stocké sous forme de chaîne sécurisée (`$Password`) et retourné à l'utilisateur pour qu'il le vérifie. Pour les mots de passe indiqués par l'utilisateur, le script emploie l'applet de commande `Read-Host` avec le paramètre `AsSecureString` afin d'obtenir le mot de passe et de l'enregistrer dans une chaîne sécurisée (`$Password`).
7. Ensuite, le script utilise la classe .NET `DirectoryServices.DirectoryEntry` pour établir une liaison avec l'UO indiquée dans Active Directory, puis la classe .NET `Directory-Services.DirectorySearcher` pour créer un objet `$Searcher`. La propriété `SearchRoot` de cet objet est liée à l'objet d'UO lié. Une recherche LDAP est lancée afin de placer dans la variable `$Computers` tous les serveurs présents dans l'UO.
8. La classe `System.Net.NetworkInformation.Ping` permet ensuite au script de contacter chaque serveur indiqué dans la collection `$Servers`. Si un serveur répond, une nouvelle ligne est alors ajoutée dans l'objet `DataTable` (`$ServersTable`). Elle est constituée du nom du serveur et de l'état "En ligne". Si un serveur ne répond pas, une nouvelle ligne est également ajoutée dans cet objet `DataTable`, mais son état est fixé à "Hors ligne".
9. Le script utilise à nouveau la classe `System.Net.NetworkInformation.Ping` pour contacter chaque serveur indiqué dans la collection `$Computers`. Si un serveur répond, une nouvelle ligne est alors ajoutée dans l'objet `DataTable` (`$ServersTable`). Elle est constituée du nom du serveur et de l'état "En ligne". Si un serveur ne répond pas, une nouvelle ligne est également ajoutée dans cet objet `DataTable`, mais son état est fixé à "Hors ligne".
10. La liste des serveurs et de leur état est enregistrée dans le journal des erreurs du script afin de pouvoir y faire ensuite référence à l'aide de l'applet de commande `Out-File`.
11. Ensuite, le script utilise la classe .NET `System.Runtime.InteropServices.Marshal` pour convertir la chaîne sécurisée contenue dans la variable `$Password` en une chaîne normale qui sera utilisée plus loin dans le script.
12. Enfin, pour chaque serveur "En ligne" présent dans `$ServersTable`, l'applet de commande `Get-WmiObject` est utilisée pour se connecter et obtenir une liste des comptes d'utilisateurs. Le compte de l'administrateur local possède un identifiant de sécurité (SID, *Security ID*) qui se termine par "-500". Le script se lie à ce compte à l'aide du fournisseur ADSI WinNT et modifie son mot de passe en utilisant la chaîne qui se trouve à présent dans la variable `$Password`.

Voici le fichier de bibliothèque `LibraryCrypto.ps1` :

```
##################################################
# LibraryCrypto.ps1
# Les fonctions de ce fichier réalisent des opérations de chiffrement.
#
# Créé le : 3/11/2006
# Auteur : Tyson Kopczynski
##################################################
#-------------------------------------------------
# New-RandomPassword
#-------------------------------------------------
# Usage :        Générer un mot de passe aléatoire.
# $Size :        Longueur du mot de passe à générer.

function New-RandomPassword{
    param ([int] $Size)

    $Bytes = new-object System.Byte[] $Size
    $Chars = "abcdefghijklmnopqrstuvwxyz".ToCharArray()
    $Chars += "ABCDEFGHIJKLMNOPQRSTUVWXYZ".ToCharArray()
    $Chars += "0123456789``~!@#$^*()-_=+[]{}`\|;:`'`",./".ToCharArray()

    $Crypto =
        new-object System.Security.Cryptography.RNGCryptoServiceProvider

# Remplir un tableau d'octets avec une séquence de valeurs aléatoires
# différentes de zéro et présentant une qualité cryptographique élevée.
    $Crypto.GetNonZeroBytes($Bytes)

    foreach ($Byte in $Bytes){

        # Pour chaque octet, appliquer un modulo.
        $Password += $Chars[$Byte % ($Chars.Length - 1)]
        }

    # Retourner le mot de passe aléatoire sous forme de SecureString.
    ConvertTo-SecureString "$Password" -AsPlainText -Force
    }
```

Comme nous l'avons mentionné précédemment, `ChangeLocalAdminPassword.ps1` utilise la fonction `New-RandomPassword` du fichier `LibraryCrypto.ps1` pour générer des mots de passe aléatoires de la taille indiquée, à partir de l'ensemble des caractères autorisés. La fonction s'appuie sur la classe .NET `System.Security.Cryptography.RNGCryptoServiceProvider` pour générer des nombres aléatoires ayant une qualité cryptographique élevée.

Un générateur de nombres aléatoires améliore la robustesse des mots de passe, même ceux constitués de caractères et de nombres. La fonction `New-RandomPassword` utilise le générateur afin d'obtenir des caractères aléatoires. Elle prend tout d'abord la longueur indiquée et crée un tableau de `System.Byte` (`$Bytes`) de la même taille. Elle définit ensuite un tableau de caractères (`$Chars`) constitué de tous les caractères acceptés dans les mots de passe.

Puis, `New-RandomPassword` crée un générateur de nombres aléatoires (`$Crypto`) à l'aide de la classe `System.Security.Cryptography.RNGCryptoServiceProvider`. La méthode `GetNonZeroBytes()` utilise ensuite `$Crypto` pour remplir le tableau `$Bytes` avec une suite de valeurs aléatoires différentes de zéro, présentant une qualité cryptographique élevée. Pour chaque octet du tableau `$Bytes`, la fonction applique un modulo (le reste de la division d'un nombre par un autre) afin de déterminer le caractère du tableau `$Chars` qui doit être ajouté dans la variable `$Password`. Le résultat final est un mot de passe aléatoire retourné à l'appelant sous forme d'une chaîne sécurisée.

L'extrait de code suivant contient l'en-tête du script `ChangeLocalAdminPassword.ps1`. Il fournit des informations sur le rôle du script, sa date de dernière mise à jour et son auteur. Juste après l'en-tête, nous trouvons le paramètre `OUDN` du script :

```
##################################################
# ChangeLocalAdminPassword.ps1
# Ce script modifie le mot de passe de l'administrateur local
# sur les comptes d'ordinateurs dans.
#
# Créé le : 11/2/2006
# Auteur : Tyson Kopczynski
##################################################
param([string] $OUDN)
```

Le script définit ensuite les fonctions `Set-ChoiceMessage` et `New-PromptYesNo` :

```
##################################################
# Fonctions.
##################################################
#-------------------------------------------------
# Set-ChoiceMessage
#-------------------------------------------------
# Usage :        Proposer les options de choix Oui et Non.
# $No :          Le message Non.
# $Yes :         Le message Oui.

function Set-ChoiceMessage{
    param ($No, $Yes)

    $N = ([System.Management.Automation.Host.ChoiceDescription]"&Non")
    $N.HelpMessage = $No

    $Y = ([System.Management.Automation.Host.ChoiceDescription]"&Oui")
    $Y.HelpMessage = $Yes

    Return ($Y,$N)
    }

#-------------------------------------------------
# New-PromptYesNo
#-------------------------------------------------
# Usage :        Afficher une invite de choix.
# $Caption :     Intitulé de l'invite.
# $Message :     Message d'invite.
# $Choices :     Catégorie de l'objet.

function New-PromptYesNo{
    param ($Caption, $Message,
        [System.Management.Automation.Host.ChoiceDescription[]]$Choices)

    $Host.UI.PromptForChoice($Caption, $Message, $Choices, 0)
    }
```

PowerShell demande parfois d'effectuer un choix avant de poursuivre une commande. Par exemple, comme nous l'avons vu au Chapitre 4, "Signer du code", il peut demander une confirmation avant d'exécuter un script qui n'est pas signé par une entité approuvée, en fonction de la stratégie d'exécution en cours. Il peut également attendre une confirmation avant d'exécuter une commande lorsqu'une applet de commande est utilisée avec l'option `confirm` :

```
PS C:\> get-process | stop-process –confirm

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « Stop-Process » en cours sur la cible « alg (2116) ».
[O] Oui  [T] Oui pour tout  [N] Non  [U] Non pour tout  [S] Suspendre
[?] Aide(la valeur par défaut est « O ») :
```

Grâce aux fonctions `Set-ChoiceMessage` et `New-PromptYesNo`, nous pouvons construire un menu d'options Oui ou Non et le présenter aux utilisateurs sur la console PowerShell. La fonction `Set-ChoiceMessage` crée un ensemble d'objets de choix et est utilisée avec la fonction `New-PromptYesNo` pour générer le menu. Pour cela, `New-PromptYesNo` invoque la méthode `PromptForChoice()` de l'objet `$host.UI`, qui n'est qu'une implémentation de la classe `System.Management.Automation.Host.PSHostUserInterface`.

Dans l'extrait de code suivant, nous définissons les variables qui seront employées par la suite dans le script. Par ailleurs, deux fichiers de bibliothèque sont chargés dans la portée du script. Le premier, `LibraryGen.ps1`, est une bibliothèque générale qui contient les fonctions d'utilisation du script et les fonctions Active Directory développées au Chapitre 9, "PowerShell et Active Directory". Le second est la bibliothèque `LibraryCrypto.ps1` mentionnée précédemment. Elle fournit la fonction `New-RandomPassword` :

```
##################################################
# Code principal.
##################################################
#--------------------
# Charger des bibliothèques.
#--------------------
. .\LibraryGen.ps1
. .\LibraryCrypto.ps1
```

```
#--------------------
# Définir les variables de configuration.
#--------------------
$ScriptName = "ChangeLocalAdminPassword.ps1"
$ScriptUsage = "Ce script modifie le mot de passe de l'administrateur" `
    local sur certaines machines."
$ScriptCommand = "$ScriptName -OUDN valeur"
$ScriptParams = "OUDN = Le distinguishedName de l'UO où se trouvent" `
    + " les machines."
$ScriptExamples = "$ScriptName ""OU=Accounts,DC=companyabc,DC=com"""
$ErrorLogName = $ScriptName + "_Errors.log"
$Date = Date
```

L'étape suivant la définition des variables du script et le chargement des bibliothèques consiste à vérifier si l'utilisateur a besoin d'une aide et si le paramètre `OUDN` obligatoire a été défini :

```
#--------------------
# Vérifier les paramètres obligatoires.
#--------------------
if ($args[0] -match '-(\?|(h|(help)))'){
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }

if (!$OUDN){
    write-host
    write-host "Veuillez préciser dans quelle UO se trouvent les machines !" `
            -Foregroundcolor Red
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }
```

Ensuite, le script crée un objet `DataTable`. Il s'agit d'un concept nouveau qui s'appuie sur un objet .NET `DataTable` (il provient de la classe `System.Data.DataTable`, qui fait partie de l'architecture ADO.NET) :

```
#--------------------
# Définir le DataTable.
#--------------------
$ServersTable = new-object System.Data.DataTable
$ServersTable.TableName = "Servers"
[Void]$ServersTable.Columns.Add("Name")
[Void]$ServersTable.Columns.Add("Status")
```

Les objets `DataTable` sont équivalents aux tables d'une base de données, excepté qu'ils se trouvent en mémoire. Les scripts peuvent se servir de ces tables pour conserver les données obtenues depuis d'autres sources ou celles définies manuellement.

Dans ce script, un `DataTable` est utilisé pour enregistrer les informations d'état des serveurs récupérés dans Active Directory. Le script commence par créer un objet `DataTable` nommé `$ServersTable` en utilisant l'applet de commande `New-Object` et la classe `System.Data.DataTable`. Lorsqu'il est créé, le `DataTable` est vide et ne possède aucune structure. Nous devons donc la définir avant de pouvoir y placer des données. Pour la structure de `$ServersTable`, le script invoque la méthode `Add()` afin d'ajouter les colonnes `Name` et `Status` à sa collection `Columns`. Plus loin, cette même méthode `Add()` sera utilisée pour ajouter des lignes de données à la collection `Rows` de `$ServersTable`.

Dans l'extrait de code suivant, l'applet de commande `Out-File` crée un journal des erreurs et y écrit des informations d'en-tête. Ensuite, la fonction `Get-ScriptHeader` est utilisée pour signaler à l'opérateur du script que la partie automation a démarré :

```
#--------------------
# Début du script.
#--------------------
# Commencer le journal des erreurs.
$ScriptName + " Exécuté le : " + $Date | out-file $ErrorLogName

write-host
Get-ScriptHeader $ScriptName $ScriptUsage
write-host
```

Le script doit à présent vérifier si la connexion au domaine est valide. Pour cela, il se sert de la fonction `Get-CurrentDomain`. Si aucune connexion valide n'existe, le script se termine et retourne son code d'état. Dans le cas contraire, il poursuit son exécution et affiche le nom de domaine sur la console. Ensuite, il appelle la fonction `Get-ADObject` pour vérifier si la chaîne contenue dans la variable `$OUDN` est un nom distinctif valide. Si la fonction retourne un objet, la variable est alors valide. Sinon, elle est considérée comme invalide et le script se termine :

```
.{
    trap{write-host `t "[ERREUR]" -Foregroundcolor Red;
        throw write-host $_ -Foregroundcolor Red;
        Break}

    write-host "Vérification de la connexion au domaine" -NoNewLine

    # Tester la connexion au domaine.
    $Domain = Get-CurrentDomain

    # Afficher le nom du domaine.
    write-host `t $Domain.Name -Foregroundcolor Green
}

write-host "Vérification du nom de l'UO" -NoNewLine

if (!(Get-ADObject "distinguishedName" $OUDN "organizationalUnit")){
    write-host `t "Invalide !" -Foregroundcolor Red
    write-host
    Break
    }
else{
    write-host `t "[OK]" -Foregroundcolor Green
    }
```

Le code suivant correspond à la procédure de définition du mot de passe utilisé. Tout d'abord, le script demande à l'utilisateur si un mot de passe doit être généré ou s'il va le préciser. Dans le premier cas, nous demandons la saisie de la taille du mot de passe. Ensuite, un mot de passe de la longueur indiquée est généré par la fonction `New-RandomPassword`.

Si l'utilisateur a choisi d'indiquer le mot de passe, le script invoque l'applet de commande `Read-Host` avec l'option `AsSecureString` :

```
#--------------------
# Obtenir le mot de passe.
#--------------------
$Choices = Set-ChoiceMessage "Oui" " Non"
$Prompt = New-PromptYesNo "Question :" `
    "Dois-je générer un mot de passe aléatoire ?" $Choices

while(!$Password){

    trap{write-host "Veuillez saisir un nombre entier !" `
        -Foregroundcolor Red; Continue}

    if ($Prompt -eq 0){
        write-host
        [int]$Length = read-host "Veuillez indiquer la longueur du mot de passe"

        if ($Length -gt 0){
            &{
                $Temp = New-RandomPassword $Length

                write-host
                write-host "Voici le nouveau mot de passe aléatoire :" `
                    -Foregroundcolor White

               [System.Runtime.InteropServices.Marshal]::PtrToStringAuto( `
               [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR( `
                    $Temp))

                $Prompt = New-PromptYesNo "Question :" `
                    "Est-il correct ?" $Choices

                if ($Prompt -eq 0){
                    $Script:Password = $Temp
                    }
            }
            }
        else{
           write-host "La longueur du mot de passe doit être supérieure à 0 !" `
                -Foregroundcolor Red
```

```
            }
        }
    else{
        write-host
        $Password = read-host "Veuillez saisir le mot de passe" -AsSecureString
        }
    }
```

Le script dispose donc du nouveau mot de passe. Il doit à présent obtenir la liste des machines dont le mot de passe est à modifier. Le code suivant se charge de cette tâche. Grâce à la classe `DirectoryServices.DirectorySearcher`, il recherche les objets d'ordinateurs (les serveurs) qui se trouvent dans l'UO. Ensuite, il contacte chacun d'eux et ajoute une ligne à l'objet `DataTable $ServersTable`, constituée du `dNSHostName` du serveur et de son état :

```
#--------------------
# Obtenir les ordinateurs et leur état.
#--------------------
write-host
write-host "Obtention des informations sur les serveurs" -NoNewLine

&{
    trap{write-host `t "[ERREUR]" -Foregroundcolor Red;
        throw write-host $_ -Foregroundcolor Red;
        Break}

    $Root =
        new-object DirectoryServices.DirectoryEntry "LDAP://$OUDN"

    $Searcher = new-object DirectoryServices.DirectorySearcher
    $Searcher.SearchRoot = $Root
    $Searcher.PageSize = 1000

    $SearchItem = "CN"
    $SearchValue = "*"
    $SearchClass = "Computer"
    $SearchCat = "*"
    $Searcher.Filter =
        "(&($($SearchItem)=$($SearchValue))(objectClass=$( `
        $SearchClass))(objectCategory=$($SearchCat)))"
    $Script:Computers = $Searcher.FindAll()
```

```
}
write-host `t "[OK]" -Foregroundcolor Green

write-host "Obtention des informations d'état" -NoNewLine

$Computers | foreach-object -Begin {$i=0;} `
    -Process {$Ping = new-object Net.NetworkInformation.Ping;
            &{$dNSHostName = $_.GetDirectoryEntry().dNSHostName.ToString();
            trap{"[ERREUR] de ping : " + $dNSHostName + " $_" | out-file `
                $ErrorLogName -Append; Continue};
            $Result = $Ping.Send($dNSHostName);
            if ($Result.Status -eq "Success"){  `
                [Void]$ServersTable.Rows.Add($dNSHostName, "En ligne")} `
            else{[Void]$ServersTable.Rows.Add($dNSHostName, "Hors ligne")};
            $i = $i+1;
           write-progress -Activity "Contact des serveurs - $($dNSHostName)" `
                -Status "Progression :" `
                -PercentComplete ($i / $Computers.Count * 100)}}

write-host `t "[OK]" -Foregroundcolor Green

# Écrire les informations d'état dans le journal des erreurs.
$ServersTable | out-file $ErrorLogName –Append
```

L'étape suivante modifie les mots de passe sur tous les serveurs en ligne. Tout d'abord, le script convertit en chaîne normale la chaîne sécurisée donnée par la variable `$Password`. Ensuite, à l'aide de la méthode `Select()` de `DataTable`, il place dans la variable `$OnlineServers` tous les objets d'ordinateurs qui correspondent à des serveurs en ligne. Puis, il utilise WMI pour se connecter à chacun d'eux, déterminer le compte Administrateur et lui affecter le mot de passe contenu dans la variable `$Password` :

```
write-host "Modification des mots de passe"  -NoNewLine

$Password = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto( `
    [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR( `
    $Password))

$OnlineServers = $ServersTable.Select("Status = 'En ligne'")

foreach ($Server in $OnlineServers) {
    &{
```

```
        write-progress -Activity "Obtention des utilisateurs - $($Server.Name)" `
             -Status "Veuillez patienter..."

        $Users = get-wmiobject -ErrorVariable Err -ErrorAction `
                    SilentlyContinue Win32_UserAccount -Computer $Server.Name

        write-progress -Activity "Obtention des utilisateurs - $($Server.Name)" `
             -Status "Terminé" -completed $True

        if ($Err.Count -ne 0){
            "[ERREUR] d'obtention des utilisateurs : " + $Server.Name + " " + `
                $Err | out-file `
                $ErrorLogName -Append
            }
        else{
            foreach ($User in $Users){
                if ($User.SID.EndsWith("-500") -eq $True){
                    write-progress -Activity `
                        "Modification du mot de passe - `$($User.Name)" `
                        -Status "Veuillez patienter..."

                    trap{"[ERREUR] de modification du mot de passe : " + `
                        $Server.Name + " " + $_ | out-file `
                        $ErrorLogName -Append; Continue}

                    $WinNTUser =
                       new-object System.DirectoryServices.DirectoryEntry( `
                        "WinNT://" +  $Server.Name + "/" + $User.Name)

                    $WinNTUser.SetPassword($Password)
                    $Null = $WinNTUser.SetInfo

                    write-progress -Activity `
                        "Modification du mot de passe - $($User.Name)" `
                        -Status "Terminé" -completed $True
                    }
                }
            }
    }
    }

write-host `t "[OK]" -Foregroundcolor Green
write-host
write-host "Le script est terminé !" -Foregroundcolor Green
write-host "Consultez le fichier $ErrorLogName en cas d'erreurs." `
-Foregroundcolor Yellow
```

En résumé

Au cours de ce chapitre, nous avons étudié deux scripts PowerShell qui ont été développés pour répondre à des besoins d'automation très forts. Dans le premier, nous avons découvert que PowerShell était capable de sortir de son rôle d'outil d'automation et de répondre à un besoin sécuritaire critique en remplaçant le shell de Windows. Dans le second, nous avons vu qu'il pouvait être un outil d'automation performant. Les deux scripts n'ont cependant fait qu'effleurer la surface des procédures d'automation possibles avec PowerShell.

Comme nous l'avons déjà répété maintes fois dans cet ouvrage, les possibilités de PowerShell sont sans limites. Ce chapitre constitue seulement un pas dans l'exploration de PowerShell et de ses capacités.

11

Administrer Exchange avec PowerShell

Dans ce chapitre

- Introduction
- Exchange Management Shell (EMS)
- Le script GetDatabaseSizeReport.ps1
- Le script GetEvent1221Info.ps1
- Le script ProvisionExchangeUsers.ps1

Introduction

Ce chapitre explique comment utiliser PowerShell pour administrer un environnement Exchange Server 2007. Ce logiciel se sert de PowerShell pour réaliser des tâches d'automation et de gestion au travers d'EMS (*Exchange Management Shell*). Par ailleurs, nous présentons le concept de composant logiciel enfichable PowerShell, ce qu'est en réalité EMS. Enfin, nous étudions trois scripts PowerShell qui prennent en charge la gestion d'un environnement Exchange Server 2007 et nous précisons comment les utiliser pour répondre aux besoins d'automation.

Exchange Management Shell (EMS)

Depuis plusieurs années, les administrateurs d'Exchange n'ont eu que deux possibilités pour réaliser des tâches répétitives : les effectuer à la main au travers de l'interface graphique ou écrire des scripts dans des langages de programmation complexes. Bien que ces langages de programmation permettent d'effectuer de nombreuses tâches routinières dans un environnement Exchange, ils n'ont pas été conçus explicitement pour cela. Par conséquent, même la tâche la plus simple peut demander des centaines de lignes de code.

Avec le temps, l'impossibilité d'automatiser des tâches s'est largement révélée comme l'aspect le plus frustrant de l'administration d'un environnement Exchange. En réalité, comme le souligne cet ouvrage, l'automation Windows en général n'est pas suffisante car Microsoft s'est principalement appuyé sur des interfaces graphiques et très peu sur des interfaces en ligne de commande. Cette frustration est devenue l'une des motivations de l'équipe PowerShell, conduite par Jeffrey Snover, pour développer une interface CLI du shell qui permette aux administrateurs de tout réaliser depuis la ligne de commande.

À peu près au même moment, l'équipe chargée du développement d'Exchange était en pleine écriture des spécifications de la version suivante (E12, qui est devenu Exchange Server 2007). Initialement, elle semblait opter pour une autre interface graphique MMC (*Microsoft Management Console*) limitée pour la gestion d'Exchange. Mais elle a décidé de prendre une voie différente en s'orientant vers une administration fondée sur PowerShell.

En conséquence, dans Exchange Server 2007, la configuration et l'administration se font avec deux nouveaux outils : EMS et EMC (*Exchange Management Console*). Pour accéder aux informations et aux paramètres de configuration d'un environnement Exchange Server 2007 et pour les modifier, ils s'appuient tous deux sur PowerShell.

INFO

Exchange Server 2007 est le premier logiciel Microsoft qui utilise exclusivement PowerShell pour mettre en œuvre ses interfaces de gestion.

EMS est une interface de gestion en ligne de commande conçue pour l'administration et la configuration d'un serveur. Puisqu'elle repose sur une plate-forme PowerShell, elle peut se connecter à l'environnement d'exécution .NET (également appelé CLR, pour *Common Language Runtime*). Les tâches qui devaient autrefois être réalisées manuellement dans

l'application d'administration peuvent désormais faire l'objet de scripts, ce qui apporte aux administrateurs une plus grande souplesse dans les activités répétitives. D'autre part, les administrateurs peuvent gérer tous les aspects d'Exchange Server 2007, y compris la création et la maintenance des comptes de courrier électronique, la configuration des connecteurs et des agents de transport SMTP (*Simple Mail Transport Protocol*), ainsi que la définition des propriétés pour les enregistrements dans des bases de données. Toutes les tâches d'administration d'un environnement Exchange peuvent à présent être accomplies depuis la ligne de commande. De plus, EMS peut être utilisé pour contrôler des paramètres, créer des rapports, fournir des informations sur la santé des serveurs Exchange et surtout automatiser des tâches fréquentes.

EMC est un outil graphique MMC 3.0 pour la visualisation et la modification de la configuration d'un déploiement Exchange Server 2007. Alors que l'interface d'EMC était analogue à celle d'ESM (*Exchange System Manager*) dans les versions précédentes d'Exchange, elle a été revue. Elle est désormais mieux organisée et plus facile à appréhender. Puisque EMC est limité aux modifications que les administrateurs peuvent effectuer, certains paramètres de configuration ne sont disponibles qu'au travers d'EMS.

EMS et EMC s'appuient sur PowerShell pour réaliser les tâches d'administration. EMC n'est qu'une interface graphique qui invoque EMS pour effectuer les tâches. EMS n'est qu'un composant logiciel enfichable pour PowerShell. Par conséquent, peu importe l'outil employé par les administrateurs pour créer un rapport ou modifier un paramètre, ils utilisent en réalité PowerShell.

Un simple composant logiciel enfichable

Un composant logiciel enfichable (*snap-in*) n'est rien d'autre qu'une collection d'une ou de plusieurs applets de commande compilées dans une DLL et utilisées pour étendre les fonctionnalités de PowerShell. En général, cette extension des fonctionnalités a pour objectif la gestion d'une application au travers de PowerShell et peut se faire aisément en utilisant des composants logiciels enfichables, tout comme ils permettent d'étendre les possibilités de la MMC.

Un composant logiciel enfichable PowerShell doit être chargé dans la session PowerShell en cours avant de pouvoir être utilisé. Par exemple, supposons que nous venions de terminer un composant PowerShell en C#. Il a été compilé dans la bibliothèque `MonPremierSnapin.dll` et

nous voulons l'utiliser dans PowerShell. Avant toute chose, nous devons l'enregistrer dans l'installation de PowerShell à l'aide de l'outil .NET Framework Installer (`installutil.exe`) :

```
PS C:\Dev> set-alias IntUtil $Env:windir\Microsoft.NET\Framework\v2.0.50727\
installutil.exe
PS C:\Dev> IntUtil MonPremierSnapin.dll
Microsoft (R) .NET Framework Installation utility Version 2.0.50727.832
Copyright (c) Microsoft Corporation. Tous droits réservés.

Exécution d'une installation traitée avec transaction.
...
L'installation traitée avec transaction est terminée.
PS C:\Dev>
```

Pour une version 64 bits de Windows, voici le chemin de l'outil .NET Framework Installer :

```
PS C:\Dev> set-alias IntUtil $Env:windir\Microsoft.NET\Framework64\
v2.0.50727\installutil.exe
```

Une fois le composant logiciel enregistré, nous pouvons vérifier son existence à l'aide de l'applet de commande `Get-PSSnapin` et de son paramètre `registered` :

```
PS C:\Dev> get-pssnapin -registered

Name        : MonPremierSnapin
PSVersion   : 1.0
Description : Pour devenir le maître du monde.


PS C:\Dev>
```

INFO

La liste retournée par `Get-PSSnapin` est constituée uniquement des composants logiciels enfichables enregistrés dans une installation PowerShell. Elle ne contient aucun des composants fournis par l'installation de base de PowerShell.

Après avoir vérifié l'enregistrement du composant logiciel enfichable, nous le chargeons dans la session PowerShell en utilisant l'applet de commande `Add-PSSnapin` :

```
PS C:\Dev> add-pssnapin MyFirstSnapin
PS C:\Dev>
```

L'applet de commande `Get-PSSnapin` nous permet ensuite de confirmer sa disponibilité dans la session PowerShell en cours :

```
PS C:\Dev> get-pssnapin

Name        : Microsoft.PowerShell.Core
PSVersion   : 1.0
Description : Ce composant logiciel enfichable Windows PowerShell contient des
              applets de commande de gestion qui permettent de gérer les
              composants de Windows PowerShell.

Name        : Microsoft.PowerShell.Host
PSVersion   : 1.0
Description : Ce composant logiciel enfichable Windows PowerShell contient des
              applets de commande utilisées par l'hôte Windows PowerShell.

Name        : Microsoft.PowerShell.Management
PSVersion   : 1.0
Description : Ce composant logiciel enfichable Windows PowerShell contient des
              applets de commande de gestion qui permettent de gérer les
              composants Windows.
Name        : Microsoft.PowerShell.Security
PSVersion   : 1.0
Description : Ce composant logiciel enfichable Windows PowerShell contient des
              applets de commande qui permettent de gérer la sécurité de Windows
              PowerShell.

Name        : Microsoft.PowerShell.Utility
PSVersion   : 1.0
Description : Ce composant logiciel enfichable Windows PowerShell contient des
              applets de commande utilitaires qui permettent de manipuler des
              données.

Name        : MonPremierSnapin
PSVersion   : 1.0
Description : Pour devenir le maître du monde.


PS C:\Dev>
```

Nous pouvons désormais utiliser notre composant logiciel enfichable `MonPremierSnapin` dans la session PowerShell. Cependant, si nous fermons cette session et en ouvrons une nouvelle, il faut à nouveau charger le composant. Comme les alias, les fonctions et les variables, un composant logiciel enfichable est, par défaut, valide uniquement dans la session PowerShell courante. Pour qu'il persiste entre les sessions, il doit être chargé à chaque démarrage d'une session PowerShell.

Comme nous l'avons expliqué au Chapitre 2, "Les fondamentaux de PowerShell", l'utilisation d'un profil PowerShell permet de rendre persistants les alias, les fonctions et les variables. Nous pouvons également employer un profil pour charger un composant logiciel enfichable dans nos sessions PowerShell. Une autre méthode consiste à utiliser un fichier console PowerShell ; ce fichier de configuration a l'extension `.psc1`. Il est constitué d'informations XML qui donnent la liste des composants logiciels enfichables chargés au démarrage de la session PowerShell. Pour créer un fichier console, nous utilisons l'applet de commande `Export-Console`, comme le montre l'exemple suivant qui crée `MaConsole.psc1` :

```
PS C:\Dev> export-console MaConsole
PS C:\Dev>
```

L'extrait de code suivant est un exemple de fichier console PowerShell :

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>1.0</PSVersion>
    <PSSnapIns>
    <PSSnapIn Name="MonPremierSnapin" />
  </PSSnapIns>
</PSConsoleFile>
```

PowerShell peut ensuite utiliser ce document XML pour charger les composants logiciels enfichables conformément à une configuration précédente de sa console. Pour cela, il suffit de lancer PowerShell en définissant le paramètre `PSConsoleFile` :

```
C:\>powershell.exe –PSConsoleFile C:\Dev\MaConsole.psc1
```

Naturellement, nous ne souhaitons pas saisir cette commande à chaque lancement de PowerShell. Nous pouvons donc créer un raccourci qui démarre notre configuration personnelle de PowerShell. Cette méthode est analogue à l'ouverture d'EMS depuis le menu Démarrer de Windows.

Puisque EMS n'est qu'un composant logiciel enfichable pour PowerShell, il suffit de le charger dans la session PowerShell pour pouvoir accéder à ses applets de commande :

```
PS C:\> add-pssnapin Microsoft.Exchange.Management.PowerShell.Admin
PS C:\>
```

Cependant, le chargement du composant EMS et le démarrage d'EMS à partir du menu Démarrer de Windows ne sont pas totalement équivalents. Si nous chargeons simplement le composant logiciel enfichable, nous ne disposons pas de la console d'administration personnalisée d'Exchange. La session PowerShell ne ressemble pas et ne fonctionne pas comme EMS car le composant charge uniquement les applets de commande pour l'administration de l'environnement Exchange. Pour que la session PowerShell ressemble à EMS, nous devons exécuter le même script de configuration que celui visé par le raccourci du menu Démarrer. Ce script, `Exchange.ps1`, se trouve dans le répertoire par défaut des binaires d'Exchange Server 2007 : `C:\Program Files\Microsoft\Exchange Server\Bin`.

Le script GetDatabaseSizeReport.ps1

`GetDatabaseSizeReport.ps1` est le premier script étudié dans ce chapitre. Il génère un rapport sur la taille des bases de données des boîtes aux lettres dans un déploiement Exchange. Ce rapport contient les informations suivantes :

- le nom du serveur d'une boîte aux lettres ;
- le nom complet de la base de données, y compris celui du groupe de stockage ;
- le lecteur sur lequel se trouve la base de données ;
- l'espace disponible sur le lecteur, en gigaoctets ;
- la taille de la base de données, en gigaoctets.

Voici un exemple de rapport créé par `GetDatabaseSizeReport.ps1` :

```
Server,Database,Drive, FreeSpace,Size
SFEX01,SG1\DB1,C:,34.67,40.453
SFEX02,SG1\DB1,F:,40.56,20.232
SFEX02,SG1\DB2,F:,40.56,30.2144
SFEX02,SG2\DB1,F:,40.56,45.333
```

Toute information concernant notre environnement réseau est utile. Cependant, lorsque nous utilisons Exchange, connaître la taille des bases de données des boîtes aux lettres, leur croissance, l'espace disponible sur le lecteur hôte et le comportement général des bases de données dans l'environnement réseau peut permettre d'éviter certains problèmes.

Ce script a été développé pour companyabc.com, une petite société dont le réseau est composé de plusieurs centaines d'utilisateurs et de deux serveurs Exchange. Les contraintes budgétaires n'autorisent qu'une seule personne dans le service informatique. Elles n'ont également pas permis l'achat et l'installation d'un logiciel de surveillance et de compte-rendu sur les systèmes informatiques. Par conséquent, le seul employé du service informatique ne disposait que de méthodes manuelles pour s'assurer du fonctionnement des systèmes et n'avait pas souvent le temps d'effectuer une surveillance proactive.

Les bases de données des boîtes aux lettres d'Exchange ont donc grossi jusqu'à empêcher toute maintenance hors ligne et les disques hébergeant ces bases de données manquaient d'espace. Après avoir évité de justesse plusieurs désastres, la direction de companyabc.com a demandé au service informatique de trouver une solution pour améliorer la surveillance des bases de données Exchange. Ayant besoin d'un moyen rapide, souple et bon marché, l'unique développeur s'est tourné vers les scripts et a demandé le développement de `GetDatabaseSizeReport.ps1`.

Vous en trouverez une copie dans le dossier `Scripts\Chapitre 11\GetDatabaseSizeReport` et en téléchargement depuis le site **www.pearsoneducation.fr**. L'exécution de ce script ne nécessite aucun paramètre. Cependant, un paramètre facultatif, `ExportFile`, peut préciser le nom du fichier CSV dans lequel sont enregistrées les données du rapport. Voici la commande qui permet d'exécuter le script `GetDatabaseSizeReport.ps1` :

```
PS C:\Scripts> .\GetDatabaseSizeReport.ps1
```

Les Figures 11.1 et 11.2 illustrent son exécution.

Voici la suite des opérations effectuées par le script `GetDatabaseSizeReport.ps1` :

1. Le script crée deux objets `DataTable` : `$ServersTable`, qui enregistre les informations d'état des serveurs de boîtes aux lettres Exchange, et `$ReportTable`, qui contient le rapport sur la taille des bases de données.
2. Il crée un journal des erreurs nommé `GetDatabaseSizeReport_Errors.log` à l'aide de l'applet de commande `Out-File`. Ce journal permet aux utilisateurs d'obtenir des informations détaillées sur les erreurs.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\GetDatabaseSizeReport.ps1 report.csv

 Obtention des informations sur le lecteur - E2007
    Veuillez patienter...

# Utilisateur :  tyson
# Date :         Tue Oct 30 13:54:05     2007
#########################################

Obtention des serveurs de boîtes aux lettres     [OK]
Obtention des informations d'état        [OK]
Obtention des informations pour le rapport_
```

Figure 11.1

Exécution en cours du script GetDatabaseSizeReport.ps1.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\GetDatabaseSizeReport.ps1 report.csv

#########################################
# Script         GetDatabaseSizeReport.ps1
# Usage :        Ce script génère un rapport sur la taille des bases de données
 Exchange.
# Utilisateur :  tyson
# Date :         Tue Oct 30 13:54:05     2007
#########################################

Obtention des serveurs de boîtes aux lettres     [OK]
Obtention des informations d'état        [OK]
Obtention des informations pour le rapport       [OK]


   Server: E2007

Database                                Drive FreeSpace Size
--------                                ----- --------- ----
First Storage Group\Mailbox Database C:    2,714     0,0644683837890625


   Server: e2007CCR

Database                                  Drive FreeSpace Size
--------                                  ----- --------- ----
Storage Group 2\Mailbox Database 2        F:    0,995     0,0058746337890625
First storage group\Mailbox Database 2 F:    0,995     0,0058746337890625


PS C:\Scripts>
```

Figure 11.2

Exécution du script GetDatabaseSizeReport.ps1 terminée.

3. Le script invoque l'applet de commande `Get-MailboxServer` pour obtenir une liste de tous les serveurs de boîtes aux lettres Exchange, qui est ensuite placée dans la variable `$MailboxServers`.
4. Le script utilise la classe `System.Net.NetworkInformation.Ping` pour contacter chaque serveur de la collection `$MailboxServers`. Si un serveur répond, une nouvelle ligne est alors ajoutée dans l'objet `$ServersTable`. Elle est constituée du nom du serveur et de l'état "En ligne". Si un serveur ne répond pas, une nouvelle ligne est également ajoutée dans cet objet `$ServersTable`, mais son état est fixé à "Hors ligne".
5. La liste des serveurs et de leur état est enregistrée dans le journal des erreurs du script afin de pouvoir y faire ensuite référence à l'aide de l'applet de commande `Out-File`.
6. Pour chaque serveur "En ligne" présent dans `$ServersTable`, le script procède aux opérations suivantes :
   – L'applet de commande `Get-MailboxDatabase` est appelée pour obtenir une liste de toutes les bases de données des boîtes aux lettres sur le serveur. Les informations `Name`, `StorageGroupName` et `EdbFilePath` de chaque base sont placées dans la variable `$Databases`.
   – Pour chaque base de boîtes aux lettres présente dans la collection `$Databases`, le script invoque l'applet de commande `Get-WmiObject` afin de réunir des informations sur la taille de la base et l'espace disponible sur le lecteur. Il ajoute ensuite à `$ReportTable` une ligne qui contient le nom du serveur de boîtes aux lettres (`$Server.Name`), le nom de la base de données (`$DBName`), la lettre du lecteur hébergeant la base (`$DBDriveName`), l'espace disponible (`$DBDriveFreeSpace`) et la taille de la base (`$DBSize`).
7. Le script exporte toutes les données de `$ReportTable` à l'aide de la fonction `Export-DataTable`.

INFO

Ce script et les autres scripts de ce chapitre ne pourront être exécutés qu'en utilisant une session PowerShell dans laquelle le composant logiciel enfichable `Microsoft.Exchange.Management.PowerShell.Admin` a été chargé.

Le premier extrait de code contient l'en-tête du script `GetDatabaseSizeReport.ps1`. Il fournit des informations sur le rôle du script, sa date de dernière mise à jour et son auteur.

Juste après l'en-tête, nous trouvons le seul paramètre `ExportFile` du script :

```
##################################################
# GetDatabaseSizeReport.ps1
# Ce script génère un rapport sur la taille des bases de données Exchange.
#
# Créé le : 26/10/2006
# Auteur : Tyson Kopczynski
##################################################
param([string] $ExportFile)
```

Dans le script `GetDatabaseSizeReport.ps1`, une seule fonction (`Export-DataTable`) est définie :

```
##################################################
# Fonctions.
##################################################
#-------------------------------------------------
# Export-DataTable
#-------------------------------------------------
# Usage :        Exporter un DataSet dans un fichier CSV.
# $Data :        L'objet DataSet.
# $FileName :    Nom du fichier CSV d'exportation.

function Export-DataTable{
    param ($Data, $FileName)

    $Null = `
        [System.Reflection.Assembly]::LoadWithPartialName( `
            "System.Windows.Forms")

    trap{write-host "[ERREUR] $_" -Foregroundcolor Red; Continue}

    if ($FileName -eq ""){
        $exFileName = new-object System.Windows.Forms.saveFileDialog
        $exFileName.DefaultExt = "csv"
        $exFileName.Filter = "CSV (Comma delimited)(*.csv)|*.csv"
        $exFileName.ShowDialog()
        $FileName = $exFileName.FileName
        }

    if ($FileName -ne ""){
        $LogFile = new-object System.IO.StreamWriter($FileName, $False)
```

```
        for ($i=0; $i -le $Data.Columns.Count-1; $i++){
            $LogFile.Write($Data.Columns[$i].ColumnName)

            if ($i -lt $Data.Columns.Count-1){
                $LogFile.Write(",")
                }
            }

        $LogFile.WriteLine()

        foreach ($Row in $Data.Rows){
            for ($i=0; $i -le $Data.Columns.Count-1; $i++){
                $LogFile.Write($Row[$i].ToString())

                if ($i -lt $Data.Columns.Count-1){
                    $LogFile.Write(",")
                    }
                }

            $LogFile.WriteLine()
            }

        $LogFile.Close()
        }
    }
```

Pour exporter les données, la fonction `Export-DataTable` s'appuie sur la classe .NET `System.IO.StreamWriter` et crée `$LogFile`, un objet de la classe .NET `TextWriter`. Celui-ci permet d'écrire un objet dans une chaîne ou des chaînes dans un fichier ou de sérialiser un document XML. Dans ce script, `$LogFile` est utilisé pour envoyer le contenu du `DataTable` dans le fichier CSV (qui est créé en même temps que `$LogFile`). Pour cela, la fonction `Export-DataTable` écrit dans le fichier CSV les noms des colonnes du `DataTable`, séparés par une virgule (`,`). Ensuite, elle parcourt chaque ligne du `DataTable` et écrit ses valeurs dans le fichier CSV, en les séparant par une virgule (`,`).

Si `Export-DataTable` est invoquée sans préciser le nom du fichier CSV, elle utilise la classe .NET `System.Windows.Forms.saveFileDialog` pour construire une boîte de dialogue Enregistrer sous, qui permet d'indiquer le nom et l'emplacement de ce fichier (voir Figure 11.3).

Cet exemple illustre l'une des nombreuses possibilités offertes à PowerShell par Windows Forms pour collecter ou afficher des données.

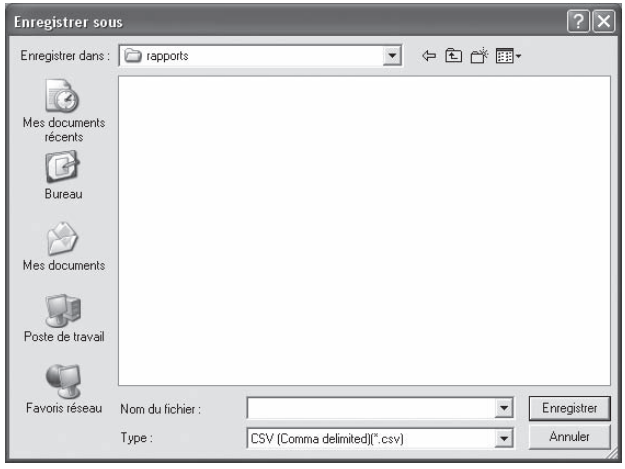


Figure 11.3

Boîte de dialogue Enregistrer sous fournie par Windows Forms.

Dans l'extrait de code suivant, nous définissons les variables qui seront utilisées par la suite. Par ailleurs, la bibliothèque `LibraryGen.ps1` est chargée de manière à fournir les fonctions d'utilisation du script :

```
##################################################
# Code principal.
##################################################
#-------------------
# Charger des bibliothèques.
#-------------------
. .\LibraryGen.ps1

#-------------------
# Définir les variables de configuration.
#-------------------
$ScriptName = "GetDatabaseSizeReport.ps1"
$ScriptUsage = "Ce script génère un rapport sur la taille des bases" `
   +"de données Exchange."
$ScriptCommand = "$ScriptName -ExportFile valeur"
$ScriptParams = "ExportFile = Fichier CSV d'exportation."
$ScriptExamples = "$ScriptName ""rapport.csv"""
$ErrorLogName = "GetDatabaseSizeReport.log"
$Date = Date
```

Ensuite, le script vérifie si l'utilisateur a besoin d'une aide :

```
#--------------------
# Vérifier les paramètres obligatoires.
#--------------------
if ($args[0] -match '-(\?|(h|(help)))'){
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }
```

Le code suivant crée les deux objets `DataTable`. Le premier se trouve dans `$ServersTable` et contient les informations sur le serveur. Le second est placé dans `$ReportTable` et contient les informations du rapport :

```
#--------------------
# Définir les DataTable.
#--------------------
$ServersTable = new-object System.Data.DataTable
$ServersTable.TableName = "Servers"
[Void]$ServersTable.Columns.Add("Name")
[Void]$ServersTable.Columns.Add("Status")

$ReportTable = new-object System.Data.DataTable
$ReportTable.TableName = "Servers"
[Void]$ReportTable.Columns.Add("Server")
[Void]$ReportTable.Columns.Add("Database")
[Void]$ReportTable.Columns.Add("Drive")
[Void]$ReportTable.Columns.Add("FreeSpace")
[Void]$ReportTable.Columns.Add("Size")
```

Puis, l'applet de commande `Out-File` crée un journal des erreurs et y écrit des informations d'en-tête. Ensuite, la fonction `Get-ScriptHeader` signale à l'opérateur du script que la partie automation a démarré :

```
#--------------------
# Début du script.
#--------------------
# Commencer le journal des erreurs.
$ScriptName + " Exécuté le : " + $Date | out-file $ErrorLogName
write-host
Get-ScriptHeader $ScriptName $ScriptUsage
write-host
```

Après l'affichage de l'en-tête à l'utilisateur, la tâche suivante du script consiste à obtenir une liste des serveurs de boîtes aux lettres à l'aide de l'applet de commande `Get-MailboxServer`. Ensuite, pour chaque objet présent dans la variable `$MailboxServers`, il contacte le serveur correspondant afin de déterminer son état. Au cours de cette procédure, l'état obtenu et le nom du serveur sont écrits dans une nouvelle ligne de l'objet `$ServersTable` :

```
#--------------------
# Obtenir les serveurs et leur état.
#--------------------
write-host "Obtention des serveurs de boîtes aux lettres" -NoNewLine
$MailboxServers = get-mailboxserver
write-host `t "[OK]" -Foregroundcolor Green

write-host "Obtention des informations d'état" -NoNewLine

$MailboxServers | foreach-object -Begin {$i=0;} `
    -Process {&{$Ping = new-object Net.NetworkInformation.Ping;
            $MBServerName = $_.Name;
            trap{"[ERREUR] de ping : " + $MBServerName + " $_" | out-file `
                $ErrorLogName -Append; Continue};
            $Result = $Ping.Send($MBServerName);
            if ($Result.Status -eq "Success"){ `
                [Void]$ServersTable.Rows.Add($MBServerName, "En ligne")} `
            else{[Void]$ServersTable.Rows.Add($MBServerName, "Hors ligne")};
            $i = $i+1;
            write-progress -Activity "Contact des serveurs - $($MBServerName)" `
                 -Status "Progression :" `
                 -PercentComplete ($i / $MailboxServers.Count * 100)}}

write-host `t "[OK]" -Foregroundcolor Green

# Écrire les informations d'état dans le journal des erreurs.
$ServersTable | out-file $ErrorLogName –Append
```

La phase suivante consiste à générer le rapport final. Le script se sert de l'applet de commande `Get-MailboxDatabase` pour obtenir le `EdbFilePath` de chaque serveur Exchange en ligne. Ensuite, pour chaque base de données des boîtes aux lettres, le script utilise WMI pour connaître la taille de la base de données et l'espace disponible sur le disque qui l'héberge.

Toutes ces informations sont ensuite ajoutées dans une nouvelle ligne de l'objet `$ReportTable` (un `DataTable`) :

```
#-------------------
# Obtenir les informations pour le rapport.
#-------------------
write-host "Obtention des informations pour le rapport" -NoNewLine

$OnlineServers = $ServersTable.Select("Status = 'En ligne'")

foreach ($Server in $OnlineServers) {
    &{
        trap{"[ERREUR] de création du rapport : " + $Server.Name + " $_" | `
            out-file $ErrorLogName -Append; Continue}

       write-progress `
            -Activity "Obtention des informations des bases de données
            - $($Server.Name)" `
            -Status "Veuillez patienter..."

        $Databases = get-mailboxdatabase -Server $Server.Name | `
            select Name, StorageGroupName, EdbFilePath

        foreach ($Database in $Databases){
            &{
                write-progress `
                    -Activity "Obtention des informations sur le lecteur " + `
                    - $($Server.Name)" `
                    -Status "Veuillez patienter..."

                $DBDriveName = $Database.EdbFilePath.DriveName
                $DBDrive = `
                    get-wmiobject Win32_PerfRawData_PerfDisk_LogicalDisk `
                    -Computer $Server.Name -Filter "Name = '$DBDriveName'"

                write-progress -Activity `
                    "Obtention des informations sur la taille du lecteur " + `
                    - $($Server.Name)" `
                    -Status "Veuillez patienter..."
                # \ doit être remplacé par \\.
                $DBPath = $Database.EdbFilePath.PathName.Replace("\","\\")
                $DBFile = get-wmiobject CIM_DataFile -Computer $Server.Name `
                    -Filter "Name = '$DBPath'"

                $DBName = $Database.StorageGroupName + "\" + $Database.Name
```

```
                # Les mégaoctets doivent être convertis en gigaoctets.
                $DBDriveFreeSpace = $DBDrive.FreeMegabytes / 1000

                # Les octets doivent être convertis en gigaoctets.
                $DBSize = $DBFile.FileSize / 1073741824

                 [Void]$ReportTable.Rows.Add($Server.Name, $DBName, `
                    $DBDriveName, $DBDriveFreeSpace, $DBSize)
            }
            }

        write-progress -Activity `
            "Obtention des informations des bases de données - $($Server.Name)" `
                -Status "Terminé" -completed $True
    }
    }
write-host `t "[OK]" -Foregroundcolor Green
```

Pour finir, le script affiche le rapport sur la console PowerShell à l'aide de l'applet de commande `Format-Table` et exporte les données dans le fichier CSV en invoquant la fonction `Export-DataTable`.

```
$ReportTable | format-table -groupBy Server Database, Drive, `
    FreeSpace, Size -autosize

$Null = Export-DataTable $ReportTable $ExportFile
```

Le script GetEvent1221Info.ps1

Les administrateurs peuvent utiliser le script `GetEvent1221Info.ps1` pour effectuer des recherches dans les journaux d'événements d'application des serveurs Exchange Server 2007 et générer un rapport qui contient les messages dont l'identifiant d'événement est 1221. À partir de ces messages, les administrateurs Exchange peuvent déterminer la quantité d'espace vide présent dans une base de données pendant la durée indiquée (nombre de jours avant aujourd'hui). Voici les points contenus dans le rapport :

- le nom du serveur des boîtes aux lettres ;
- la date et l'heure auxquelles l'événement a été écrit dans le journal Application ;

- le nom complet de la base de données, y compris le nom du groupe de stockage ;
- la quantité d'espace vide, en mégaoctets.

Voici un exemple de rapport généré par le script `GetEvent1221Info.ps1` :

```
Server,TimeWritten,Database,MB
SFEX02,10/27/2006 1:00:02 AM,SG1\DB1,500
SFEX02,10/27/2006 1:00:06 AM,SG2\PF1,700
SFEX02,10/27/2006 2:00:00 AM,SG1\DB1,500
SFEX02,10/27/2006 2:00:01 AM,SG2\PF1,700
SFEX02,10/27/2006 3:00:00 AM,SG1\DB1,500
SFEX02,10/27/2006 3:00:32 AM,SG2\PF1,700
SFEX02,10/27/2006 4:00:00 AM,SG1\DB1,500
SFEX02,10/27/2006 4:00:00 AM,SG2\PF1,700
SFEX01,10/27/2006 1:00:04 AM,SG1\DB2,200
SFEX01,10/27/2006 1:00:04 AM,SG1\DB1,100
SFEX01,10/27/2006 2:00:00 AM,SG1\DB1,200
SFEX01,10/27/2006 2:00:00 AM,SG1\DB2,100
SFEX01,10/27/2006 3:15:00 AM,SG1\DB1,100
SFEX01,10/27/2006 3:15:00 AM,SG1\DB2,200
SFEX01,10/27/2006 4:00:00 AM,SG1\DB1,200
SFEX01,10/27/2006 4:00:00 AM,SG1\DB2,100
```

Ce script a été développé pour companyabc.com, une société de cinquante utilisateurs dont les boîtes aux lettres Exchange sont très volumineuses (4 Go et plus). Cette entreprise produit des paquetages marketing constitués d'images numériques et dont la taille moyenne est supérieure à 20 Mo. Les employés de companyabc.com travaillent à domicile et dans différentes agences. Ils s'échangent généralement les paquetages marketing par courrier électronique au lieu de les poster dans un endroit partagé.

Puisque les employés utilisent leur boîte aux lettres comme des systèmes de fichiers en ligne, la taille de ces boîtes a augmenté très rapidement. Comprenant que des boîtes d'une telle taille sont coûteuses et difficiles à maintenir, l'administrateur Exchange de companyabc.com a demandé que le contenu marketing soit enregistré localement sur les disques durs des utilisateurs et retiré de leur boîte aux lettres. Cette pratique a évité aux bases de données Exchange de grandir trop rapidement. Cependant, la fréquence de suppression des messages électroniques volumineux a engendré un autre problème : les bases de données Exchange sont remplies de vastes zones vides.

La quantité d'espace vide est importante car lorsqu'une base de données Exchange a grandi, sa taille ne peut être réduite tant que l'administrateur ne procède pas à une défrag-

mentation hors connexion. Par exemple, une base de données a atteint 12 Go, mais les utilisateurs ont supprimé 3 Go de messages. Après une défragmentation en ligne, les identifiants d'événements 1221 indiquent un espace disponible de 3 Go. Les nouveaux messages écrits dans la base utilisent cet espace et sa taille n'augmente pas tant que tout cet espace n'est pas utilisé.

La base de données occupe toujours 12 Go sur le disque dur, même si elle ne contient que 9 Go de données. Si elle est plus volumineuse que nécessaire, elle risque d'augmenter le temps nécessaire aux sauvegardes et aux restaurations. En examinant les événements d'identifiant 1221, les administrateurs peuvent déterminer si une défragmentation hors connexion est nécessaire pour réduire la taille de la base de données et améliorer les performances globales. Par ailleurs, en surveillant périodiquement ces événements dans les journaux, les administrateurs suivent la quantité d'espace vide moyen d'une base de données et déterminent plus facilement le schéma de croissance des données réelles dans la base. Grâce à cette information, ils sont en mesure de décider du moment où un espace supplémentaire doit lui être alloué.

N'ayant pas le budget nécessaire pour acheter des outils Exchange, companyabc.com a demandé le développement d'un script qui surveille l'espace disponible dans les bases de données Exchange. Le script résultant se nomme `GetEvent1221Info.ps1`.

Vous en trouverez une copie dans le dossier `Scripts\Chapitre 11\GetEvent1221Info` et en téléchargement depuis le site **www.pearsoneducation.fr**. L'exécution de ce script nécessite la définition d'un paramètre. L'argument du paramètre `Days` doit fixer la période (en nombre de jours) concernée par la recherche des événements d'identifiant 1221 dans les serveurs des boîtes aux lettres. L'argument du paramètre facultatif `ExportFile` précise le nom du fichier CSV dans lequel seront exportées les données du rapport. Voici la commande qui permet d'exécuter le script `GetEvent1221Info.ps1` :

```
PS C:\Scripts> .\GetEvent1221Info.ps1 5
```

Les Figures 11.4 et 11.5 illustrent son exécution.

Voici la suite des opérations réalisées par le script `GetEvent1221Info.ps1` :

1. Le script crée deux objets `DataTable` : `$ServersTable`, qui enregistre les informations d'état des serveurs de boîtes aux lettres Exchange, et `$EventsTable`, qui contient le rapport des événements 1221.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\GetEvent1221Info.ps1 5

 Collecte des événements depuis - E2007
    Veuillez patienter...

# Utilisateur :  tyson
# Date :         Tue Oct 30 14:13:13     2007
#######################################

Obtention des serveurs de boîtes aux lettres     [OK]
Obtention des informations d'état        [OK]
Obtention des informations événements
```

Figure 11.4

Exécution au cours du script GetEvent1221Info.ps1.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\GetEvent1221Info.ps1 5

#######################################
# Script         GetEvent1221Info.ps1
# Usage :        Ce script exploite les événements 1221 des serveurs des boîtes
 aux lettres.
# Utilisateur :  tyson
# Date :         Tue Oct 30 14:13:13     2007
#######################################

Obtention des serveurs de boîtes aux lettres     [OK]
Obtention des informations d'état        [OK]
Obtention des informations événements    [OK]

Le script est terminé !
Consultez GetEvent1221Info.log pour les erreurs.
PS C:\Scripts>
```

Figure 11.5

Exécution du script GetEvent1221Info.ps1 terminée.

2. Il crée un journal des erreurs nommé `GetEvent1221Info _Errors.log` à l'aide de l'applet de commande `Out-File`. Ce journal permet aux utilisateurs d'obtenir des informations détaillées sur les erreurs.
3. Le script invoque l'applet de commande `Get-MailboxServer` pour obtenir une liste de tous les serveurs de boîtes aux lettres Exchange, qui est ensuite placée dans la variable `$MailboxServers`.
4. Le script utilise la classe `System.Net.NetworkInformation.Ping` pour contacter chaque serveur de la collection `$MailboxServers`. Si un serveur répond, une nouvelle ligne est alors ajoutée dans l'objet `$ServersTable`. Elle est constituée du nom du serveur et de l'état "En ligne". Si un serveur ne répond pas, une nouvelle ligne est également ajoutée dans cet objet `$ServersTable`, mais son état est fixé à "Hors ligne".
5. La liste des serveurs et de leur état est enregistrée dans le journal des erreurs du script afin de pouvoir y faire ensuite référence à l'aide de l'applet de commande `Out-File`.
6. Pour chaque serveur "En ligne" présent dans `$ServersTable`, le script procède aux opérations suivantes :
   - La fonction `Get-RemoteEventLog` est appelée de manière à créer un objet (`$Events`) lié au journal Application du serveur. Pour créer cet objet, elle utilise la classe .NET `System.Diagnostics.Eventlog`, qui permet à une application ou à un script d'interagir avec les journaux des événements d'une machine.
   - Ensuite, le script invoque l'applet de commande `Select-Object` pour sélectionner, à partir de la propriété Entries de l'objet `$Events`, tous les événements 1221 qui sont survenus pendant la période indiquée (`$Days`). La collection d'événements résultante est placée dans la variable `$1221Events`.
   - Pour chaque objet de la collection `$1221Events`, le script appelle sa méthode `get_timewritten()` afin d'enregistrer dans la variable `$TimeWritten` le moment de l'événement. Ensuite, une expression régulière extrait du message de l'événement l'espace libre (`$MB`) et le nom (`$Database`) de la base de données.
   - Une ligne qui contient le nom du serveur (`$Server.Name`), la date de l'événement (`$TimeWritten`), le nom de la base de données (`$Database`) et l'espace disponible en mégaoctets (`$MB`) est ajoutée à `$EventsTable`.
7. Le script exporte toutes les données de `$EventsTable` à l'aide de la fonction `Export-DataTable`.

Le premier extrait de code contient l'en-tête du script `GetEvent1221Info.ps1`. Il fournit des informations sur le rôle du script, sa date de dernière mise à jour et son auteur. Juste après l'en-tête, nous trouvons les paramètres du script :

```
##################################################
# GetEvent1221Info.ps1
# Ce script exploite les événements 1221 des serveurs des
# boîtes aux lettres.
#
# Créé le : 26/10/2006
# Auteur : Tyson Kopczynski
##################################################
param([int] $Days, [string] $ExportFile)
```

Ensuite, la fonction `Get-RemoteEventLog` est définie. Elle collecte les informations `EventLog` d'une machine distante en utilisant la classe `System.Diagnostics.Eventlog`. Puis, c'est au tour de la fonction `Export-DataTable`, que nous avons déjà rencontrée à la section précédente :

```
##################################################
# Fonctions.
##################################################
#-----------------------------------------------
# Get-RemoteEventLog
#-----------------------------------------------

# Usage :        Recueillir les informations EventLog depuis
#                une machine distante.
# $Machine :     Nom de la machine ("MonServeur").
# $Log :         Nom du EventLog ("Application").

function Get-RemoteEventLog{
    param ($Machine, $Log)

    trap{Continue}
    new-object System.Diagnostics.Eventlog $Log, $Machine
    }

#-----------------------------------------------
# Export-DataTable
#-----------------------------------------------
# Usage :        Exporter un DataSet dans un fichier CSV.
# $Data :        L'objet DataSet.
# $FileName :    Nom du fichier CSV d'exportation.
```

```
function Export-DataTable{
    param ($Data, $FileName)

    $Null = `
        [System.Reflection.Assembly]::LoadWithPartialName( `
            "System.Windows.Forms")

    trap{write-host "[ERREUR] $_" -Foregroundcolor Red; Continue}

    if ($FileName -eq ""){
        $exFileName = new-object System.Windows.Forms.saveFileDialog
        $exFileName.DefaultExt = "csv"
        $exFileName.Filter = "CSV (Comma delimited)(*.csv)|*.csv"
        $exFileName.ShowDialog()
        $FileName = $exFileName.FileName
        }

    if ($FileName -ne ""){
        $LogFile = new-object System.IO.StreamWriter($FileName, $False)

        for ($i=0; $i -le $Data.Columns.Count-1; $i++){
            $LogFile.Write($Data.Columns[$i].ColumnName)

            if ($i -lt $Data.Columns.Count-1){
                $LogFile.Write(",")
                }
            }

        $LogFile.WriteLine()

        foreach ($Row in $Data.Rows){
            for ($i=0; $i -le $Data.Columns.Count-1; $i++){
                $LogFile.Write($Row[$i].ToString())

                if ($i -lt $Data.Columns.Count-1){
                    $LogFile.Write(",")
                    }
                }

            $LogFile.WriteLine()
            }

        $LogFile.Close()
        }
    }
```

Dans l'extrait de code suivant, nous définissons les variables qui seront utilisées par le script. Par ailleurs, la bibliothèque `LibraryGen.ps1` est chargée de manière à fournir les fonctions d'utilisation du script :

```
##################################################
# Code principal.
##################################################
#--------------------
# Charger des bibliothèques.
#--------------------
. .\LibraryGen.ps1

#--------------------
# Définir les variables de configuration.
#--------------------
$ScriptName = "GetEvent1221Info.ps1"
$ScriptUsage = "Ce script exploite les événements 1221 des serveurs" `
+ "des boîtes aux lettres."
$ScriptCommand = "$ScriptName -Days valeur -ExportFile valeur"
$ScriptParams = "Days = Nombre de jours concernés par la recherche" `
+ "des événements.", "ExportFile = Fichier CSV d'exportation."
$ScriptExamples = "$ScriptName 5 ""rapport.csv"""
$ErrorLogName = "GetEvent1221Info.log"
$Date = Date
```

Ensuite, le script vérifie si l'utilisateur a besoin d'une aide. Si ce n'est pas le cas, il s'assure que le paramètre `Days` est défini. S'il ne l'est pas, il informe l'opérateur du script que ce paramètre est obligatoire et affiche les informations d'utilisation :

```
#-------------------
# Vérifier les paramètres obligatoires.
#-------------------
if ($args[0] -match '-(\?|(h|(help)))'){
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }

if (!$Days){
    write-host
    write-host "Veuillez préciser le nombre de jours !" -Foregroundcolor Red
```

```
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }
```

Le code suivant crée les deux objets `DataTable`. Le premier se trouve dans `$ServersTable` et contient les informations sur le serveur. Le second est placé dans `$EventsTable` et contient les informations du rapport :

```
#-------------------
# Définir les DataTable.
#-------------------
$ServersTable = new-object System.Data.DataTable
$ServersTable.TableName = "Servers"
[Void]$ServersTable.Columns.Add("Name")
[Void]$ServersTable.Columns.Add("Status")

$EventsTable = new-object System.Data.DataTable
$EventsTable.TableName = "Servers"
[Void]$EventsTable.Columns.Add("Server")
[Void]$EventsTable.Columns.Add("TimeWritten",[DateTime])
[Void]$EventsTable.Columns.Add("Database")
[Void]$EventsTable.Columns.Add("MB")
```

Puis, l'applet de commande `Out-File` crée un journal des erreurs et y écrit des informations d'en-tête. Ensuite, la fonction `Get-ScriptHeader` signale à l'opérateur du script que la partie automation a démarré :

```
#-------------------
# Début du script.
#-------------------
# Commencer le journal des erreurs.
$ScriptName + " Exécuté le : " + $Date | out-file $ErrorLogName

write-host
Get-ScriptHeader $ScriptName $ScriptUsage
write-host
```

La tâche suivante consiste à obtenir une liste des serveurs de boîtes aux lettres à l'aide de l'applet de commande `Get-MailboxServer`. Ensuite, pour chaque objet présent dans la variable `$MailboxServers`, nous contactons le serveur correspondant afin de déterminer son état. Au cours de cette procédure, l'état obtenu et le nom du serveur sont écrits dans une nouvelle ligne de l'objet `$ServersTable` :

```
#-------------------
# Obtenir les serveurs et leur état.
#-------------------
write-host "Obtention des serveurs de boîtes aux lettres" -NoNewLine
$MailboxServers = get-mailboxserver
write-host `t "[OK]" -Foregroundcolor Green

write-host "Obtention des informations d'état" -NoNewLine

$MailboxServers | foreach-object -Begin {$i=0;} `
    -Process {&{$Ping = new-object Net.NetworkInformation.Ping;
            $MBServerName = $_.Name;
            trap{"[ERREUR] de ping : " + $MBServerName + " $_" | out-file `
                $ErrorLogName -Append; Continue};
            $Result = $Ping.Send($MBServerName);
            if ($Result.Status -eq "Success"){ `
                [Void]$ServersTable.Rows.Add($MBServerName, "En ligne")} `
            else{[Void]$ServersTable.Rows.Add($MBServerName, "Hors ligne")};
            $i = $i+1;
            write-progress -Activity "Contact des serveurs - $($MBServerName)" `
                 -Status "Progression :" `
                 -PercentComplete ($i / $MailboxServers.Count * 100)}}

write-host `t "[OK]" -Foregroundcolor Green

# Écrire les informations d'état dans le journal des erreurs.
$ServersTable | out-file $ErrorLogName –Append
```

La phase suivante consiste à générer le rapport final. Le script invoque la méthode `Select()` de `DataTable` pour créer une collection des serveurs en ligne (`$OnlineServers`). Ensuite, pour chaque serveur de la collection `$OnlineServers`, il appelle la fonction `Get-RemoteEventLog` afin d'obtenir tous les messages des événements Application de ce serveur.

Pour chaque message d'événement 1221, une nouvelle ligne contenant ces informations mises en forme est ajoutée à l'objet `DataTable` dans `$EventsTable` :

```
#-------------------
# Obtenir les informations d'événement.
#-------------------
write-host "Obtention des informations événements" -NoNewLine

$OnlineServers = $ServersTable.Select("Status = 'En ligne'")

foreach ($Server in $OnlineServers){
    &{
        trap{"[ERREUR] pour les informations d'événement : " `
            + "$Server.Name + " $_" | `
            out-file $ErrorLogName -Append; Continue}

        $Events = Get-RemoteEventLog $Server.Name "Application"

        # L'instruction suivante peut demander beaucoup de temps
        # en fonction du nombre de serveurs.
        write-progress -Activity "Collecte des événements depuis" `
            + "- $($Server.Name)" `
            -Status "Veuillez patienter..."

        $1221Events = $Events.Entries | where {$_.EventID -eq "1221" -and `
        $_.TimeWritten -ge $Date.AddDays(-$Days)}

        foreach ($1221Event in $1221Events){
            &{
                $Message = $1221Event | select Message
                $TimeWritten = $1221Event.get_timewritten()

                # Cette expression régulière extrait du message le nom
                # de la base de données.
                $Database = [Regex]::Match($Message, '"[^"\r\n]*"')
                $Database = $Database.Value.Replace('"', "")

                # Cette expression régulière extrait la quantité
                # d'espace vide.
                $MB = [Regex]::Match($Message, '[0-9]+')
                [Void]$EventsTable.Rows.Add($Server.Name, $TimeWritten, `
                    $Database, $MB)
```

```
            }
            }

        write-progress -Activity "Collecte des événements depuis" `
            + " - $($Server.Name)" `
            -Status "Terminé" -completed $True
    }
    }
write-host `t "[OK]" -Foregroundcolor Green
```

Pour finir, le script exporte les données de `$EventsTable` dans le fichier CSV en invoquant la fonction `Export-DataTable` :

```
#--------------------
# Exporter les données dans un fichier CSV.
#--------------------
$Null = Export-DataTable $EventsTable $ExportFile

write-host
write-host "Le script a terminé !" -Foregroundcolor Green
write-host "Consultez $ErrorLogName pour les erreurs." `
-Foregroundcolor Yellow
```

Le script ProvisionExchangeUsers.ps1

Grâce au script `ProvisionExchangeUsers.ps1`, les administrateurs Exchange créent facilement et rapidement des comptes d'utilisateurs dans des environnements Exchange Server 2007, à partir d'informations définies dans un fichier CSV, dont voici la structure :

- prénom de l'utilisateur ;
- nom de l'utilisateur ;
- alias de messagerie de l'utilisateur ;
- nom complet de la base de données.

En voici un exemple :

```
Prenom,Nom,Alias,BaseDonnees
Stu,Gronko,sgronko,SFEX01\SG1\DB1
Caelie,Hallauer,challauer,SFEX02\SG2\DB2
Duane,Putnam,dputnam,SFEX02\SG2\DB2
Essie,Fea,efea,SFEX02\SG1\DB1
Rona,Trovato,rtrovato,SFEX01\SG1\DB2
Gottfried,Leibniz,gleibniz,SFEx01\SG1\DB1
```

Lorsque le code de `ProvisionExchangeUsers.ps1` est modifié, le format du fichier CSV et les informations qui définissent les comptes d'utilisateurs peuvent être ajustés à n'importe quel environnement. Cette souplesse est importante pour répondre aux besoins d'automation souvent évolutifs.

Ce script a été demandé par companyabc.com au cours d'un processus de plusieurs fusions qui ont conduit à la création de nombreux nouveaux comptes d'utilisateurs avec messagerie. Le nombre de comptes à créer et les variations dans les informations disponibles lors de la création des comptes pour une fusion font que la meilleure solution a consisté à employer une méthode automatisée qui pouvait être adaptée aux différents besoins. Pour répondre aux contraintes de souplesse, le service informatique de companyabc.com a développé le script `ProvisionExchangeUsers.ps1`.

Vous en trouverez une copie dans le dossier `Scripts\Chapitre 11\ProvisionExchangeUsers` et en téléchargement depuis le site **www.pearsoneducation.fr**. L'exécution de ce script nécessite la définition de trois paramètres. L'argument du paramètre `UPNSuffix` indique le suffixe de nom principal universel (UPN, *universal Principal Name*) pour les nouveaux comptes avec messagerie. L'argument d'`OUDN` précise le `distinguishedName` de l'UO dans laquelle les comptes doivent être enregistrés. L'argument d'`ImportFile` désigne le fichier CSV à importer qui contient la liste des utilisateurs. Voici la commande qui permet d'exécuter le script `ProvisionExchangeUsers.ps1` :

```
PS C:\Scripts> .\ProvisionExchangeUsers.ps1 "companyabc.com" "OU=Accounts,
DC=companyabc,DC=com" users.csv
```

Les Figures 11.6 et 11.7 illustrent son exécution.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\ProvisionExchangeUsers.ps1 "companyabc.com" "OU=Accounts,DC=com
panyabc,DC=com" users.csv

 Ajout des utilisateurs
    Progression :
    [oooooooooooo                                                          ]

# Utilisateur :  tyson
# Date :         Tue Oct 30 14:42:02     2007
#########################################

Connexion au domaine     companyabc.com
Vérification du nom de l'UO      [OK]
Vérification du fichier à importer       [OK]

Veuillez saisir le mot de passe: *********
```

Figure 11.6

Exécution au cours du script ProvisionExchangeUsers.ps1.

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe
PS C:\Scripts> .\ProvisionExchangeUsers.ps1 "companyabc.com" "OU=Accounts,DC=com
panyabc,DC=com" users.csv

#########################################
# Script         ProvisionExchangeUsers.ps1
# Usage :        Ce script crée des utilisateurs Exchange d'après le contenu
# du fichier CSV indiqué.
# Utilisateur :  tyson
# Date :         Tue Oct 30 14:42:02     2007
#########################################

Connexion au domaine     companyabc.com
Vérification du nom de l'UO      [OK]
Vérification du fichier à importer       [OK]

Veuillez saisir le mot de passe: *********

Le script est terminé !
Consultez le fichier ProvisionExchangeUsers.log en cas d'erreurs.
PS C:\Scripts>
```

Figure 11.7

Exécution du script ProvisionExchangeUsers.ps1 terminée.

Voici la suite des opérations effectuées par le script `ProvisionExchangeUsers.ps1` :

1. Le script crée un journal des erreurs nommé `ProvisionExchangeUsers_Errors.log` à l'aide de l'applet de commande `Out-File`. Ce journal permet aux utilisateurs d'obtenir des informations détaillées sur les erreurs.
2. Il se connecte au domaine d'ouverture de session actuel en invoquant la fonction `Get-CurrentDomain`. À partir de l'objet retourné par cette fonction, il affiche le nom du domaine sur la console PowerShell. Si la connexion échoue, le script se termine.
3. Ensuite, le script vérifie que l'UO indiquée existe dans le domaine courant en appelant la fonction `Get-ADObject`. Si l'UO n'est pas valide, il se termine.
4. Il utilise l'applet de commande `Test-Path` pour vérifier que le fichier CSV d'importation est valide. Si ce n'est pas le cas, le script se termine.
5. Le script emploie l'applet de commande `Read-Host` avec le paramètre `AsSecureString` afin d'obtenir le mot de passe des nouveaux comptes. La chaîne sécurisée obtenue est enregistrée dans la variable `$Password`.
6. Il invoque l'applet de commande `Import-Csv` pour placer le contenu du fichier CSV dans la variable `$Users`.
7. Pour chaque utilisateur dans la collection `$Users`, le script appelle l'applet de commande `New-Mailbox` pour créer un compte d'utilisateur avec messagerie en fonction des informations du fichier CSV et de celles fournies par l'utilisateur. Les erreurs générées pendant la création du compte sont enregistrées dans le journal des erreurs du script à l'aide de l'applet de commande `Out-File`.

Le premier extrait de code contient l'en-tête du script `ProvisionExchangeUsers.ps1`. Il fournit des informations sur le rôle du script, sa date de dernière mise à jour et son auteur. Juste après l'en-tête, nous trouvons les paramètres du script :

```
##################################################
# ProvisionExchangeUsers.ps1
# Ce script crée des utilisateurs Exchange d'après le contenu
# du fichier CSV indiqué.
#
# Créé le : 10/21/2006
# Auteur : Tyson Kopczynski
##################################################
param([string] $UPNSuffix, [string] $OUDN, [string] $ImportFile)
```

Dans l'extrait de code suivant, nous définissons les variables qui seront utilisées par la suite. Par ailleurs, la bibliothèque `LibraryGen.ps1` est chargée de manière à fournir les fonctions d'utilisation du script :

```
##################################################
# Code principal.
##################################################
#--------------------
# Charger des bibliothèques.
#-------------------
. .\LibraryGen.ps1

#-------------------
# Définir les variables de configuration.
#-------------------
$ScriptName = "ProvisionExchangeUsers.ps1"
$ScriptUsage = "Ce script crée des utilisateurs Exchange d'après" `
+ "le contenu du fichier CSV indiqué."
$ScriptCommand = "$ScriptName -UPNSuffix valeur -OUDN valeur -ImportFile valeur"
$ScriptParams = "UPNSuffix = Suffixe UPN des nouveaux utilisateurs.", `
    "OUDN = Nom distinctif de l'UO dans laquelle les utilisateurs sont créés.", `
    "ImportFile = Fichier CSV à importer."
$ScriptExamples = "$ScriptName ""companyabc.com""" `
    + " ""OU=Accounts,DC=companyabc,DC=com""" `
    + " ""users.csv"""
$ErrorLogName = "ProvisionExchangeUsers.log"
$Date = Date
```

Ensuite, le script vérifie si l'utilisateur a besoin d'une aide. Si ce n'est pas le cas, il contrôle si les paramètres `UPNSuffix`, `OUDN` et `ImportFile` sont définis. S'ils ne le sont pas, il informe l'opérateur du script que ces paramètres sont obligatoires et affiche les informations d'utilisation :

```
#-------------------
# Vérifier les paramètres obligatoires.
#-------------------
if ($args[0] -match '-(\?|(h|(help)))'){
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }
```

```
if (!$UPNSuffix){
    write-host
    write-host "Veuillez préciser le suffixe UPN !" -Foregroundcolor Red
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }

if (!$OUDN){
    write-host
    write-host "Veuillez préciser l'UO où créer les utilisateurs !" `
        -Foregroundcolor Red
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }

if (!$ImportFile){
    write-host
    write-host "Veuillez préciser le fichier CSV à importer !" `
        -Foregroundcolor Red
    write-host
    Get-ScriptHeader $ScriptName $ScriptUsage
    Show-ScriptUsage $ScriptCommand $ScriptParams $ScriptExamples
    Return
    }
```

Puis, l'applet de commande `Out-File` crée un journal des erreurs et y écrit des informations d'en-tête. Ensuite, la fonction `Get-ScriptHeader` signale à l'opérateur du script que la partie automation a démarré :

```
#-------------------
# Début du script.
#-------------------
# Commencer le journal des erreurs.
$ScriptName + " Exécuté le : " + $Date | out-file $ErrorLogName

write-host
Get-ScriptHeader $ScriptName $ScriptUsage
write-host
write-host "Connexion au domaine" –NoNewLine
```

Le script doit ensuite vérifier qu'il existe une connexion valide au domaine. Pour cela, il invoque `Get-CurrentDomain`. S'il n'existe aucune connexion valide, le script se termine et retourne son code d'état. Dans le cas contraire, il poursuit son exécution et affiche le nom de domaine sur la console :

```
.{
    trap{write-host `t "[ERREUR]" -Foregroundcolor Red;
        throw write-host $_ -Foregroundcolor Red;
        Break}

    write-host "Connexion au domaine" -NoNewLine

    # Tester la connexion au domaine.
    $Domain = Get-CurrentDomain

    # Afficher le nom du domaine.
    write-host `t $Domain.Name -Foregroundcolor Green
}
```

Nous vérifions ensuite le nom distinctif dans la variable `$OUDN`. Pour cela, le script se sert de la fonction `Get-ADObject`. Elle se connecte à Active Directory et recherche l'UO par son nom distinctif. Si la fonction retourne un objet, l'UO est valide. Sinon, l'UO est considérée comme invalide et le script se termine :

```
write-host "Vérification du nom de l'UO" -NoNewLine

if (!(Get-ADObject "distinguishedName" $OUDN "organizationalUnit")){
    write-host `t "Invalide !" -Foregroundcolor Red
    write-host
    Break
    }
else{
    write-host `t "[OK]" -Foregroundcolor Green
    }
```

Le script vérifie ensuite la validité du fichier à importer à l'aide de l'applet de commande `Test-Path` :

```
write-host "Vérification du fichier à importer" -NoNewLine

if (!(test-path $ImportFile -pathType Leaf)){
    throw write-host `t "Invalide !" -Foregroundcolor Red
    }
else{
    write-host `t "[OK]" -Foregroundcolor Green
    }
```

Ensuite, pour obtenir le mot de passe de l'utilisateur, il invoque l'applet de commande `Read-Host` avec l'option `AsSecureString` :

```
#--------------------
# Obtenir le mot de passe.
#--------------------
write-host
$Password = read-host "Veuillez saisir le mot de passe" –AsSecureString
```

Enfin, le script crée les nouveaux comptes d'utilisateurs en utilisant l'applet de commande `New-Mailbox`, les informations provenant du fichier CSV et celles fournies par l'utilisateur du script :

```
#-------------------
# Créer les boîtes aux lettres.
#-------------------
write-host
write-progress -Activity "Ajout des utilisateurs" `
    -Status "Veuillez patienter..."

$Users = import-csv $ImportFile

$Users | foreach-object -Begin {$i=0;} -Process {$FName = $_.FName;
    $LName = $_.LName;
```

```
    $Alias = $_.Alias;
    $Database = $_.Database;
    $UPN = $Alias + "@" + $UPNSuffix;
    $Name = $FName + " " + $LName;
    $Null = new-mailbox -Name $Name -Database $Database `
        -OrganizationalUnit $OUDN -UserPrincipalName $UPN `
        -Password $Password -ResetPasswordOnNextLogon $True `
        -Alias $Alias -DisplayName $Name -FirstName $FName `
        -LastName $LName -ErrorVariable Err -ErrorAction `
        SilentlyContinue;
    if ($Err.Count -ne 0){ `
        "[ERREUR] ajout de l'utilisateur : " + $Alias + " " + $Err | `
        out-file $ErrorLogName -Append};
    $i = $i+1;
    write-progress -Activity "Ajout des utilisateurs" `
        -Status "Progression :" `
        -PercentComplete ($i / $Users.Count * 100)}

write-host "Le script est terminé !" -Foregroundcolor Green
write-host "Consultez le fichier $ErrorLogName en cas d'erreurs." -
Foregroundcolor Yellow
```

En résumé

Nous venons de voir comment PowerShell pouvait servir à administrer Exchange Server 2007, non seulement au travers des interfaces graphiques avec EMC, mais également depuis la ligne de commande avec EMS. Exchange Server 2007 est la première des nombreuses applications qui utiliseront PowerShell ainsi. Pour cela, elle s'appuie sur les possibilités d'extension de PowerShell à l'aide des composants logiciels enfichables. Grâce à eux, de nouvelles applets de commande deviennent disponibles aux utilisateurs de PowerShell et augmentent leur capacité à administrer un déploiement Exchange.

Les scripts étudiés au fil de ce chapitre sont une bonne démonstration de ce que nous pouvons accomplir à l'aide du composant logiciel enfichable d'Exchange Server 2007. Grâce à ces exemples, vous savez à présent comment exploiter PowerShell pour obtenir des informations sur la taille d'une base de données Exchange, calculer l'espace vide dans une base de données et créer rapidement des comptes d'utilisateurs avec messagerie électronique. Mais les possibilités d'administration d'Exchange ne se bornent pas à cela. Comme nous l'avons répété souvent, les tâches réalisables avec PowerShell ne sont limitées que par vos propres talents de développement et d'imagination.

Accepter le fait que les scripts peuvent accomplir de nombreuses tâches constitue la première étape dans la compréhension des possibilités de PowerShell. Avant de vous attaquer à des besoins d'automation plus complexes, vous devez bien évidemment comprendre PowerShell. Cependant, en faisant ce premier pas, vous avez commencé un voyage d'exploration qui vous conduira à utiliser PowerShell comme l'a imaginé son équipe de développement.

Cet ouvrage vous a guidé vers deux aspects de ce voyage. Tout d'abord, vous avez compris ce qu'est PowerShell et comment l'utiliser. Cependant, les informations de fond et les explications des caractéristiques se sont limitées à quelques chapitres, focalisées sur les sujets les plus importants à la compréhension du fonctionnement de PowerShell. Ensuite, cet ouvrage a abordé l'utilisation de PowerShell sous un angle inhabituel. Au lieu d'expliquer toutes les nuances des caractéristiques et de la syntaxe du langage de PowerShell, il a montré comment exploiter PowerShell.

Pour cela, plusieurs chapitres ont comparé l'écriture de scripts Windows et de scripts PowerShell. Des exemples en ligne de commande et des scripts opérationnels ont été étudiés, tant dans leur version VBScript que PowerShell. En procédant ainsi, vous avez pu mettre en relation vos connaissances des scripts Windows et les nouveaux concepts de PowerShell. Les deux derniers chapitres ont montré comment utiliser PowerShell pour répondre à différents besoins d'automation et pour administrer d'Exchange Server 2007. Comme lors des chapitres précédents, l'idée centrale était l'application réelle de PowerShell.

Vous êtes à présent arrivé à la fin de cet ouvrage, mais votre voyage se poursuit. PowerShell fait partie des produits les plus intéressants que Microsoft a créés depuis un certain temps. Jeffrey Snover et les autres membres de l'équipe de PowerShell doivent être félicités pour avoir identifié un besoin et développé l'outil qui permet d'y répondre. Avec l'adoption croissante de PowerShell par les applications de Microsoft et d'autres fournisseurs, l'étendue des possibilités de PowerShell va être de plus en plus évidente et ne fera qu'augmenter.

Index

Symboles

A

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Référence

Windows® PowerShell

PowerShell est un nouveau shell en ligne de commande et un langage de scripts exceptionnel, qui a été développé par Microsoft pour donner aux informaticiens la possibilité d'automatiser et de personnaliser totalement les tâches d'administration de leurs systèmes. En exploitant l'incroyable puissance de .NET Framework, PowerShell, avec sa syntaxe facile à apprendre et ses outils élaborés, a été conçu dès le départ pour accélérer les développements et offrir toute la puissance et la flexibilité nécessaires à une meilleure productivité.

Avec cet ouvrage complet et riche en exemples, vous commencerez par acquérir les bases de l'interface, vous constaterez ses liens avec l'écriture classique des scripts Windows, puis vous exploiterez vos connaissances pour les appliquer au développement de scripts PowerShell.

Pour illustrer ses explications, l'auteur fournit de nombreux exemples commentés de scripts opérationnels. Il vous enseigne des méthodes PowerShell inédites qui vous seront d'une grande utilité pour administrer Windows Server, Active Directory et Exchange Server 2007.

Téléchargez les codes source des exemples sur le site de Pearson Education France **www.pearson.fr**.

TABLE DES MATIÈRES

A propos de l'auteur

Avec plus de neuf années d'expérience dans le domaine informatique, **Tyson Kopczynski** est devenu un spécialiste d'Active Directory, des stratégies de groupe, des scripts Windows, de Windows Rights Management Services, de PKI et de la sécurité des technologies de l'information. En tant que consultant pour Convergent Computing (CCO), il a travaillé avec la nouvelle génération de technologies Microsoft depuis leur début et a joué un rôle essentiel dans le développement des pratiques d'écriture de scripts.

Programmation

Niveau : Intermédiaire
Configuration : Windows XP et ultérieur

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4015-3