



Collection

EXPERT IT

Nouvelle édition

Windows PowerShell (versions 1 et 2)

Guide de référence
pour l'administration système

**Arnaud
PETITJEAN**



**Robin
LEMESLE**



Rechercher & télécharger

RECHERCHER

www.editions-eni.fr

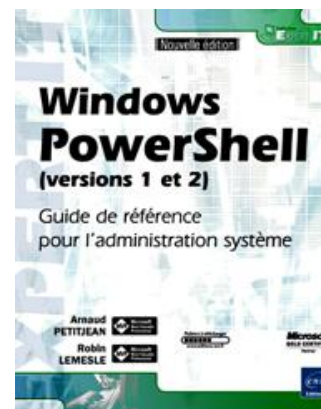
Microsoft
GOLD CERTIFIED
Partner



Windows PowerShell (v1 et 2)

Guide de référence pour l'administration système

Robin LEMESLE - Arnaud PETITJEAN



Résumé

Ce livre sur **Windows PowerShell**, écrit par les créateurs du site **PowerShell-Scripting.com**, s'adresse aussi bien aux IT Pros souhaitant optimiser leurs tâches d'administration système, qu'à un public intermédiaire de techniciens et administrateurs système. PowerShell est désormais profondément ancré dans les produits Microsoft tels que : Windows 7, Windows Server 2008 et 2008 R2, Exchange Server 2007 et 2010, SQL Server 2008, System Center, etc.

Le présent ouvrage se substitue à l'édition précédente car, outre les fonctionnalités de **PowerShell version 1**, il inclut les nouvelles fonctionnalités propres à la **version 2** et exploitables sur les dernières versions de produits Microsoft. Ces fonctionnalités sont clairement identifiées de façon à ce que le lecteur, selon ses besoins, puisse facilement faire la différence entre les deux versions.

De la **technologie .NET** aux objets **COM** en passant par **WMI** et **ADSI**, les nombreux cas concrets d'utilisation en entreprise vous aideront à devenir plus performant dans vos tâches quotidiennes.

A travers les 5 premiers chapitres, le lecteur découvrira **PowerShell sous toutes ses facettes** : de la simple utilisation de l'interpréteur de commandes, jusqu'aux techniques de scripting les plus avancées.

Le chapitre 6 sur la **technologie .NET** lui montrera que l'usage de PowerShell est pratiquement sans limites et lui ouvrira une fenêtre sur le monde de la création d'interfaces graphiques avec **Windows Forms** et **Windows Presentation Foundation (WPF)**.

Le chapitre 9 est quant à lui consacré aux **technologies dites "de remoting"** qui autorisent l'exécution de commandes ou de scripts **PowerShell à distance** grâce aux nouvelles fonctionnalités de la **version 2**.

Dans le chapitre 11 le lecteur apprendra à maîtriser le jeu de commandes PowerShell étendu apporté par le rôle **Active Directory 2008 R2**.

Enfin les chapitres suivants lui permettront de mettre en oeuvre PowerShell dans le **monde de l'administration système** au travers de nombreux cas concrets d'utilisation en situation réelle et de découvrir les outils et acteurs les plus importants de l'**écosystème Windows PowerShell**.

Parmi les **nombreux exemples** traités dans le livre, vous découvrirez comment : lister les comptes périmés d'un domaine - créer des utilisateurs par lots - surveiller les journaux d'événements - changer le mot de passe administrateur de toutes les machines d'un domaine - créer des comptes utilisateurs locaux ou du domaine - générer des rapports d'inventaires - gérer la configuration réseau d'ordinateurs à distance - générer des mots de passe - envoyer des mails - interagir avec des applications telles que Office ou encore Windows Live Messenger - et bien d'autres...

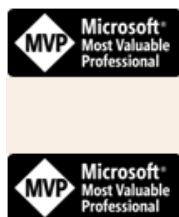
Des éléments complémentaires sont en **téléchargement** sur cette page et sur le site de la communauté PowerShell francophone : PowerShell-Scripting.com.

Arnaud Petitjean et Robin Lemesle sont reconnus Microsoft MVP (Most Valuable Professional) sur PowerShell.

Les chapitres du livre :

Avant-propos - Introduction - A la découverte de PowerShell - Fondamentaux - Maîtrise du Shell - Gestion des erreurs de débogage - La sécurité - .NET - Objets COM - Windows Management Instrumentation (WMI) - Exécution à distance - Manipulation d'objets annuaire avec ADSI - Module Active Directory de Windows Server 2008 - Études de cas - Ressources complémentaires - Conclusion - Annexes

L'auteur



Le livre est écrit par les créateurs de la communauté PowerShell francophone : **PowerShell-Scripting**.

Robin Lemesle est Ingénieur Système. Il intervient sur les technologies Microsoft, Citrix et VMware. Son goût du scripting via PowerShell, lui permet de transmettre son expérience de terrain de manière structurée afin d'offrir au lecteur un apprentissage rapide et efficace. Il est reconnu **Microsoft MVP (Most Valuable Professional)** sur **PowerShell**.

Arnaud Petitjean est Ingénieur Architecte Système, spécialiste en infrastructures systèmes et en virtualisation serveurs et postes clients (VDI). Sa passion sans limites pour PowerShell l'a naturellement conduit à devenir également formateur. Il est reconnu **Microsoft MVP (Most Valuable Professional)** sur **PowerShell**.

Ce livre numérique a été conçu et est diffusé dans le respect des droits d'auteur. Toutes les marques citées ont été déposées par leur éditeur respectif. La loi du 11 Mars 1957 n'autorisant aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les "copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective", et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, "toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayant cause, est illicite" (alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code Pénal. Copyright Editions ENI

À propos de PowerShell

Depuis Windows PowerShell 1.0 apparu en Septembre 2006 et aujourd'hui avec la version 2.0 intégrée à Windows Server 2008 R2 et Windows 7, Microsoft a introduit un virage important dans l'administration des systèmes Windows. Pour s'en convaincre, il suffit de lister le nombre de commandes de base et d'observer la diversité d'applications du marché - provenant de Microsoft et d'ailleurs - qui fournissent des commandes PowerShell.

PowerShell 2.0 est une version majeure qui permet enfin l'émancipation de la ligne de commande sur les environnements Windows. Cela va profondément modifier les habitudes de travail de nombreux IT Pro et en particulier des administrateurs système du monde entier.

À propos du livre

Dans cette deuxième édition du livre, les auteurs ont fait le choix de traiter les deux versions de PowerShell. En effet, il n'est pas rare d'administrer un environnement dans lequel les produits Microsoft cohabitent avec des versions différentes (Exchange Server 2007, SQL Server 2008, Windows 7, Windows Server 2008 et 2008 R2 par exemple). L'administrateur disposera ainsi du livre de référence pour les deux versions de PowerShell.

Les lecteurs qui connaissent déjà PowerShell 1.0 auront l'occasion de conforter leurs acquis tout en découvrant de façon claire et précise les nouvelles fonctionnalités (et elles sont nombreuses) apportées par PowerShell 2. D'un autre côté, les lecteurs qui découvrent PowerShell, suivront un enseignement progressif qui les conduira à une utilisation de l'outil quelle que soit sa version.

Le livre est organisé autour de 14 chapitres principaux qui permettent au lecteur de maîtriser PowerShell, d'en découvrir les usages courants mais aussi les techniques plus avancées (scripting, création d'interfaces graphiques, exécution à distance...). Les exemples fournis dans les derniers chapitres présentent de nombreux cas concrets d'utilisation en situation réelle et permettent de découvrir les outils et acteurs les plus importants de l'écosystème Windows PowerShell.

Au-delà de leur propre expérience professionnelle en phase avec les technologies PowerShell, les auteurs animent la communauté francophone PowerShell et bénéficient à ce titre de nombreux retours d'expérience complémentaires sur le sujet.

Pourquoi utiliser les scripts ?

Depuis toujours les administrateurs système utilisent des scripts pour automatiser la réalisation des tâches fastidieuses. En effet, quoi de plus inintéressant que la répétition de tâches d'administration, telle que la création de comptes utilisateurs, pour ne citer que la plus célèbre. En revanche, quoi de plus valorisant et enrichissant personnellement que de passer un peu de temps à écrire un script qui réalisera ces tâches à notre place ?

On pourrait s'imaginer que l'investissement du temps passé à développer un script n'est rentable que si l'on met moins de temps qu'à faire les tâches manuellement, mais ceci est une vision restreinte des choses. Et vous allez en comprendre la raison...

Pour reprendre l'exemple précédent, imaginons que l'on ait cent comptes utilisateurs à créer. S'il on passe en moyenne dix minutes par compte (en n'étant ni dérangé par le téléphone, ni par les utilisateurs, ni par une réunion ultra importante planifiée à la dernière minute), il nous faudra environ trois à quatre jours à plein temps pour réaliser le travail. Sans compter toutes les erreurs qui se seront fatalement immiscées (nom mal orthographié, oubli de créer l'espace home directory ou la boîte aux lettres, permissions mal positionnées, etc.). Imaginons maintenant que nous décidions d'écrire LE script « qui va bien ». Supposons que nous sommes débutants et qu'au pire des cas, nous passions quatre jours à réaliser celui-ci. Et bien durant ces quatre jours nous aurons appris énormément de choses, et nous aurons en plus la possibilité de réutiliser ce script lors d'une prochaine série d'utilisateurs à créer. Bref, dès lors que nous commençons à réutiliser des scripts, nous gagnons du temps ; et ce temps pourra être consacré à d'autres choses. Le plus important est surtout que nous aurons augmenté notre niveau en « scripting ». Nous pourrions donc compter sur nos nouvelles compétences pour la réalisation d'autres tâches d'administration système, peut-être encore plus complexes. De quoi rendre le métier d'administrateur système nettement plus intéressant, voire même ludique !

D'autre part, le scripting a une vertu souvent insoupçonnée des directions informatiques : celle de la qualité. Qualité qui devient le maître mot de la plupart des grandes sociétés où l'une de leur principale préoccupation est l'augmentation constante de la qualité de service au travers des bonnes pratiques apportées par ITIL (*Information Technology Infrastructure Library*). Même si le scripting ne prétend pas tout résoudre, il apporte néanmoins une brique de base importante. En effet, l'intérêt principal du scripting réside dans l'automatisation des tâches en supprimant les erreurs induites par un traitement manuel.

Historique des langages de script

Au commencement de l'histoire de l'informatique, il y avait Unix... Ce système d'exploitation a vu le jour en 1968, et avec lui sont apparus les premiers environnements d'exécution de scripts appelés « Shells » (ou « coquilles » en français). En voici quelques-uns, pour ne citer que les plus connus (dans l'ordre chronologique) : Bourne shell (sh), C shell (csh), Korn shell (ksh), bash shell (le shell bash est un projet GNU initié par la Free Software Foundation). On appelle « script shell » les scripts développés pour ces environnements. Ils sont aujourd'hui encore extrêmement employés pour l'administration des systèmes Unix.

Pendant ce temps, Microsoft développait les premières versions de DOS à Bellevue dans la banlieue de Seattle (la toute première version s'appelait Q-DOS pour *Quick and Dirty Operating System*, système d'exploitation réalisé à la va vite). C'est en 1981 que le MS-DOS fit son apparition dans sa version 1.0 avec les premiers scripts batch (fichiers à l'extension .bat). Ces derniers étaient (et le sont toujours) très limités en terme de fonctionnalités, mais ils sont encore largement utilisés par les administrateurs de systèmes Windows pour la réalisation de tâches simples.

1987 est l'année du lancement du langage PERL (*Practical Extraction and Report Language*), développé par Larry Wall, dans sa version 1.0, pour les plates-formes Unix. Ce langage - dont les fonctionnalités ont été empruntées aux scripts shell et au langage C - a pour objectif la manipulation des fichiers, des données et des processus. Il fallut patienter encore dix années afin de voir apparaître PERL dans le monde Windows, en 1997 dans le kit de ressource technique de Windows NT 4.0.

Il existe encore beaucoup d'autres langages de scripts orientés système, que nous ne détaillerons pas, car ce n'est pas l'objet de cet ouvrage, mais qui méritent néanmoins d'être cités, il s'agit de : Rexx, Python, S-Lang, Tcl/tk, Ruby, Rebol, etc.

Deux autres langages de script très répandus sur le Web sont Javascript et VBScript. Le premier fut développé par Netscape et rendu public en 1995 (dans Navigator 2.0) afin de rendre les sites Internet plus dynamiques et plus vivants, ainsi que pour apporter davantage d'interactivité aux pages HTML (*HyperText Markup Language*). La riposte de Microsoft ne se fit pas attendre lorsqu'il sortit le langage JScript (dans Internet Explorer 3.0), langage très ressemblant à celui de Netscape. Pour mettre tout le monde d'accord, l'ECMA normalisa ces deux langages en 1997 en prenant le meilleur de chacun pour former l'ECMAScript (ISO/IEC 16262) (l'ECMA est un organisme international de normalisation des systèmes d'information et de communication). C'est aussi dans Internet Explorer 3 que VBScript fit son apparition. VBScript reprend l'intégralité des fonctionnalités de Javascript mais avec une syntaxe proche du Basic, alors que Javascript ressemble plus à Java ou au C.

Pour revenir au développement des scripts système, avant l'arrivée du kit de ressource technique de NT 4.0, les administrateurs système Windows n'avaient pas beaucoup d'autre choix que d'utiliser les scripts batch MS-DOS. Ces derniers étaient simples lorsqu'il s'agissait de réaliser des tâches triviales (comme par exemple monter un lecteur réseau), mais pouvaient s'avérer très ardues dès que l'on souhaitait analyser un fichier, ou réaliser une simple boucle. Heureusement Microsoft eut la bonne idée de mettre dans le kit de ressource, en plus de Perl, un interpréteur de script KiXtart. KiXtart étend de façon significative les scripts batch en apportant de nombreuses fonctionnalités réseau, ainsi qu'un puissant langage de script réellement pensé pour l'administration système. En bref, KiXtart est une réelle alternative aux bons vieux scripts batch. Seul petit « bémol », bien que Kix soit développé par des membres de Microsoft, il n'est officiellement pas supporté par Microsoft.

C'est alors qu'apparut discrètement avec Windows 98 et avec l'Option Pack de NT4.0, un fabuleux outil d'administration nommé WSH (*Windows Script Host*). Ce dernier offre un puissant environnement d'exécution de scripts VBScript, mais est assez peu connu dans ses débuts. C'est ainsi qu'à l'insu des utilisateurs, de nombreux virus ont usé des nouvelles fonctionnalités (et failles de sécurité) offertes par WSH, mettant ainsi en avant ses capacités...

Néanmoins à l'heure actuelle WSH/VBScript est le couple le plus répandu pour la réalisation de tâches d'administration système dans le monde Windows, mais cela pourrait bien changer prochainement...

1. Et PowerShell dans tout ça ?

Microsoft, aux alentours de 2004-2005, prit conscience des limites de l'interface graphique et des difficultés éprouvées par les utilisateurs pour la réalisation de scripts. Ainsi, la firme de Redmond changea radicalement sa vision des choses et prit un virage à quatre-vingt-dix degrés lorsqu'elle annonça la naissance de Monad. Monad était le nom de code de PowerShell dans ses versions préliminaires ; la version finale de PowerShell dans sa version 1.0 a été livrée au public fin 2006. Quant à la version 2.0, la dernière en date, celle-ci s'est trouvée intégrée dans Windows Server 2008 R2 et Windows 7. La version téléchargeable fut disponible fin octobre 2009.

La nouvelle stratégie de Microsoft pour l'administration système de ses produits est maintenant clairement orientée ligne de commandes. Certes les interfaces graphiques des outils d'administration - qui ont contribué au succès de Windows - demeureront mais celles-ci seront dorénavant construites au-dessus de PowerShell. Par exemple, depuis Microsoft Exchange 2007 et maintenant Microsoft Exchange 2010, les actions effectuées graphiquement exécutent en réalité des commandes PowerShell, commandes qu'il est possible de récupérer pour les intégrer dans des scripts.

PowerShell est profondément ancré dans Windows Server 2008 R2 et chaque nouveau rôle installé apporte avec lui des nouvelles commandes. On trouve à présent des commandes pour gérer : Active Directory, les stratégies de groupes, le clustering, AppLocker, les transferts de fichiers intelligents (BITS), IIS, la gestion des rôles et fonctionnalités, etc.

Mais PowerShell est également au cœur de nombreux produits Microsoft et non Microsoft tels que : Exchange Server 2007/2010, Virtual Machine Manager depuis la version 2007, SQL Server 2008, Microsoft Deployment Toolkit 2010, SharePoint 2010 et même VMware vSphere PowerCLI, Citrix XenApp, etc. Même des éditeurs de logiciels autres que Microsoft se mettent à développer des commandes PowerShell ; cela prouve l'intérêt que suscite PowerShell...

PowerShell apporte en outre un interpréteur de lignes de commandes interactif qui faisait défaut à WSH et qui était plus que limité avec l'invite de commandes CMD.exe. On peut souligner dans PowerShell la présence d'un jeu de commandes cohérent. Enfin chose révolutionnaire, PowerShell s'appuie sur la technologie .NET pour apporter une dimension objets aux scripts et donner ainsi accès à l'immense bibliothèque de classes du Framework .NET. Dernier point pour terminer, la sécurité dans PowerShell a été la préoccupation constante des membres de l'équipe de développement. Ainsi vous découvrirez tout au long de cet ouvrage que PowerShell est vraiment une révolution à lui seul, et qu'il modifiera profondément la façon d'administrer les systèmes Windows de la prochaine décennie.

Intérêt des scripts par rapport aux langages de programmation ?

Un script est un simple fichier texte ASCII (*American Standard Code for Information Interchange*) dans lequel s'enchaînent toutes les instructions qui le composent, à l'image de n'importe quel code source. La différence entre un langage de script et un langage de programmation à proprement parler (comme C/C++, Visual Basic, ou Delphi) tient au fait qu'un script n'est pas compilé. C'est-à-dire qu'il n'est pas transformé en un binaire directement exécutable par la machine, mais qu'il faut obligatoirement un interpréteur de commandes appelé aussi « hôte de script » ou « shell » pour lancer le script.

Un des intérêts majeurs des scripts par rapport aux langages de programmation classiques est qu'il faut peu d'instructions pour arriver à faire la même chose. La syntaxe est généralement simplifiée et la programmation est moins contraignante (pas besoin de déclarer les variables, peu de types de données, etc.).

Pour résumer...

Les scripts permettent :

- **Un gain de temps** : ils sont capables de réaliser des tâches complexes et d'être lancés automatiquement par le système, sans intervention humaine. Du fait de leur simplicité, les tâches d'administration s'effectuent plus rapidement ; elles font donc gagner un temps considérable aux administrateurs.
- **De limiter les erreurs** : un script n'a besoin d'être écrit qu'une seule fois et peut être utilisé un grand nombre de fois. Par conséquent, les risques d'erreurs liés à la réalisation manuelle d'une tâche sont grandement diminués.
- **Plus de flexibilité** : les scripts peuvent s'adapter à toutes les situations en intégrant un minimum de logique.

Présentation de PowerShell

PowerShell est à la fois un interpréteur de commandes et un puissant langage de scripts. Il tire sa puissance en grande partie grâce à son héritage génétique du Framework .NET sur lequel il s'appuie. Bien connu des développeurs, le Framework .NET l'est beaucoup moins des administrateurs système et autres développeurs de scripts ; ce qui est normal. Pour vulgariser en quelques mots, le Framework .NET est une énorme bibliothèque de classes à partir desquelles nous ferons naître des objets ; objets qui nous permettront d'agir sur l'ensemble du système d'exploitation en un minimum d'effort. Tous ceux qui ont goûté à la puissance du Framework .NET ne tariront pas d'éloges à son égard. C'est donc grâce à ce dernier que PowerShell a pu se doter d'une dimension objet. Et c'est d'ailleurs cette faculté à manipuler les objets qui fait de PowerShell un « shell » d'exception !

Avec PowerShell vous ne manipulerez donc plus uniquement du texte, comme c'est le cas avec la plupart des autres shells, mais le plus souvent des objets ; et ce, sans vraiment vous en rendre compte. Par exemple, lorsque vous utiliserez le pipe « | » pour passer des informations à une commande, et bien vous ne transmettez pas du texte, mais un objet avec tout ce qui le caractérise (ses propriétés et ses méthodes). C'est grâce à cela que les scripts PowerShell sont généralement plus concis que les autres langages de scripts tels que le VBScript pour n'en citer qu'un seul.

D'autre part, PowerShell est fourni avec un jeu de commandes extrêmement riche. On en dénombre environ cent trente dans la version 1, soit près du double de celles que nous avons l'habitude d'utiliser avec CMD.exe et plus de deux cent trente dans la version 2. Ceci étant dit, les commandes CMD restent toujours utilisables avec PowerShell, si besoin est. Les commandes PowerShell possèdent l'immense avantage d'être toutes conçues sur le même modèle. Elles ont des noms faciles à retenir, et il est aisé de deviner des noms de commandes que l'on ne connaît pas. Chacune d'elles possède également un jeu de paramètres important, paramètres que l'on mémorise assez facilement du fait d'une forte cohérence de l'ensemble.

Enfin, pour terminer cette présentation, sachez qu'il existe une aide en ligne intégrée à la console que vous pouvez solliciter à tout moment et qui est très complète. L'aide en ligne donne accès à trois niveaux d'explications : standards, détaillées, ou maximum. À partir de l'aide détaillée, de nombreux exemples illustrent l'utilisation des commandes.

Tous les points que nous venons de vous exposer font de PowerShell un langage de script (mais pas seulement) très puissant mais surtout facile à apprendre, même pour ceux qui n'ont jamais goûté à la programmation.

Installation de PowerShell

Le pré-requis minimum nécessaire à l'installation de Windows PowerShell est le Framework .NET. Ce dernier n'est pas disponible pour la plate-forme Windows 2000, par conséquent Windows 2000 n'est pas pris en charge.

Dans le cas où vous ne puissiez pas installer PowerShell dans sa toute dernière version (la version 2.0), voici un tableau récapitulatif de ce qu'il vous faudra installer selon votre système d'exploitation :

PowerShell v1.0	Windows XP SP2 minimum	Windows Vista	Windows Server 2003 SP1 minimum Windows Server 2003 R2	Windows Server 2008	Windows Server 2008 R2 / Windows 7
Framework .NET 2.0	à installer	intégré	à installer	intégré	intégré
Binaires PowerShell	WindowsXP-KB926140- v5-x86- FRA.exe	Windows6.0-KB928439-x86.msu Windows6.0-KB928439-x64.msu	WindowsServer2003-KB926140-v5-x86-FRA.exe WindowsServer2003.WindowsXP- KB926139-v2 -x64-ENU.exe	intégré à installer en tant que composant additionnel	non supporté

Tableau de compatibilité pour PowerShell v1

PowerShell v2.0	Windows XP SP3 minimum	Windows Vista SP1 minimum	Windows Server 2003 SP2 minimum Windows Server 2003 R2	Windows Server 2008	Windows Server 2008 R2 / Windows 7
Framework .NET 2.0	installation nécessaire	intégré	installation nécessaire	intégré	intégré
Framework .NET 3.0	facultatif	intégré	facultatif	intégré	intégré
Framework .NET 3.5	facultatif	facultatif	facultatif	facultatif	intégré
Binaires PowerShell	WindowsXP-KB968930-x86-FRA.exe	Windows6.0-KB968930-x86.msu Windows6.0-KB968930-x64.msu	WindowsServer2003-KB968930-x86- FRA.exe WindowsServer2003-KB968930-x64- FRA.exe	Windows6.0-KB968930-x86.msu Windows6.0-KB968930-x64.msu	intégré

Tableau de compatibilité pour PowerShell v2

Qu'il s'agisse de la v1 ou de la v2, l'exécutable PowerShell.exe est toujours installé dans le répertoire C:\Windows\System32\WindowsPowerShell\v1.0. Et ce pour la simple et bonne raison que les deux versions utilisent le même moteur interne, à savoir celui de la v1. Il en est de même pour les extensions de script - dont nous parlerons plus tard dans cet ouvrage - qui se nomment « *.ps1 » quelle que soit la version.

Informations concernant l'installation de PowerShell et de ses pré-requis :

- Le Framework .NET 3.0 est nécessaire si vous souhaitez utiliser l'éditeur graphique PowerShell (voir plus loin) et la commande Out-GridView.
- La commande Get-Event ne fonctionne pas sur Windows XP et nécessite le Framework .NET 3.5.
- WinRM/WSMan est nécessaire pour l'exécution de scripts à distance et en arrière-plan.

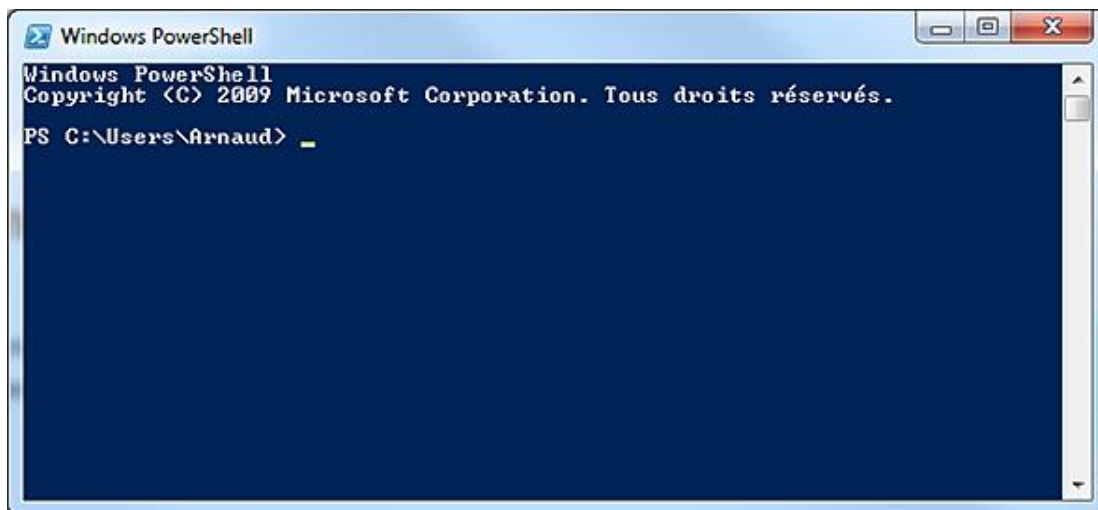
- Comme vous pouvez le remarquer, PowerShell version 2 est installé de base sur les plates-formes Windows Server 2008 R2 et Windows 7.

Prise en main

Avec la version 2 est apparu un appréciable éditeur de scripts PowerShell en mode graphique, mais dans un premier temps, intéressons-nous à la console « classique ».

1. Découverte de la console ligne de commandes

Au premier coup d'œil, rien ne permet de distinguer une fenêtre « console PowerShell » d'une fenêtre « invite de commande CMD.exe », si ce n'est la couleur de fond (bleue pour l'une et noire pour l'autre).



La console PowerShell au démarrage

Voici les touches et séquences de touches qui nous permettent de « naviguer » dans la console :

Touche	Description
[Flèche en haut]/ [Flèche en bas]	Permet de faire défiler l'historique des commandes déjà frappées.
[F7]	Affiche une boîte contenant l'historique des commandes. La sélection s'effectue à l'aide des [Flèche en haut] [Flèche à droite]/[Flèche à gauche].
[F8]	Fait défiler l'historique sur la ligne de commande.
[F9]	Permet de rappeler une commande de l'historique à partir de son numéro.
[Flèche à droite]/[Flèche à gauche]	Permet de déplacer le curseur sur la ligne de commande courante.
[Ctrl][Flèche à droite]	Déplace le curseur vers la droite en passant d'un mot à l'autre sur la ligne de commande.
[Ctrl][Flèche à gauche]	Déplace le curseur vers la gauche en passant d'un mot à l'autre sur la ligne de commande.
[Home]	Ramène le curseur au début de la ligne de commande.
[Fin]	Envoie le curseur à la fin de la ligne de commande.
[Ctrl] C	Met fin à l'exécution de l'instruction courante.
[Ctrl][Pause]	Met fin à l'exécution de la console.

Nous avons mis en gras, les touches qui nous semblent être les plus utiles. Soulignons l'intérêt de la touche [F7] qui permet en un coup d'œil de retrouver une commande dans l'historique.

```
acer le nom de la variable par sa valeur.
me"
est :
0: Trace-Command -name FormatViewBinding
1: write 'Powershell, it's good for heal
2: write 'Powershell, it's good for hea
3: write 'Powershell, it's good for hea
4: help about_qu
5: Write-Host 'Il est facile d'éc
6: dans PowerShell.'
7: help about_Escape_character
8: help about
9: help about_Quoting_Rules
illemet
ontre c
erra" c
est :
on dit.
```

Historique des commandes avec [F7]

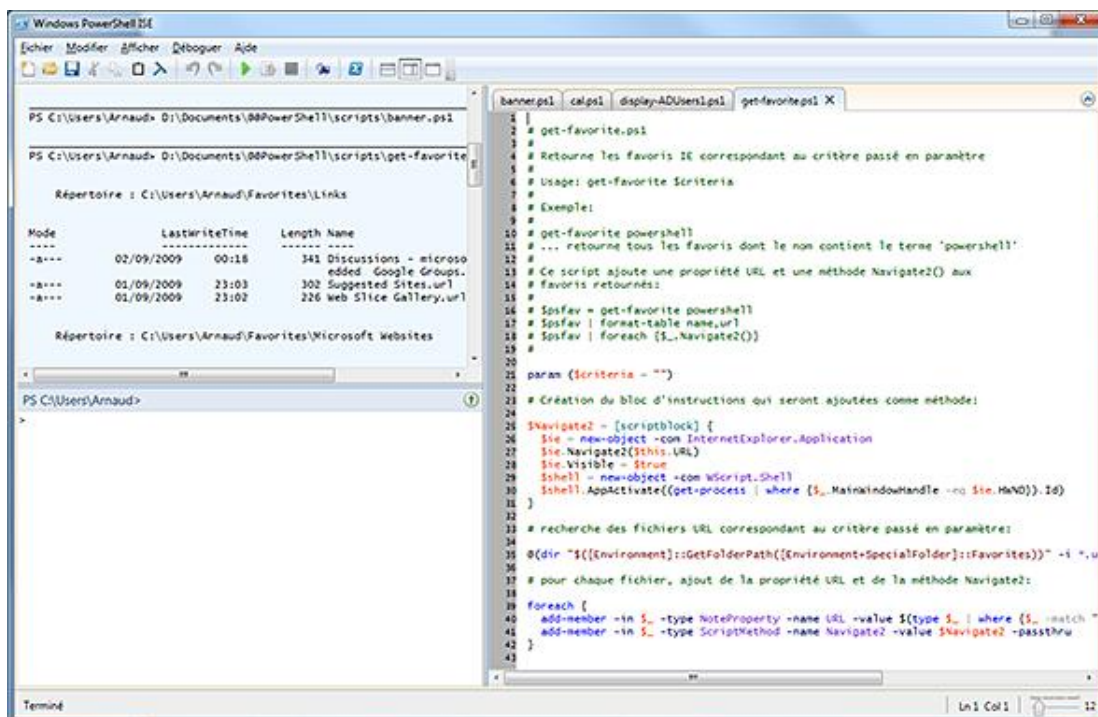
➤ Une fois la commande retrouvée dans l'historique, vous pouvez soit presser la touche [Entrée] pour la sélectionner et l'exécuter, soit presser la flèche droite (ou gauche) pour modifier la commande avant de l'exécuter.

2. L'environnement d'écriture de scripts intégré (ISE)

Un éditeur de scripts est maintenant de la partie (dans powershell v2), ce qui est vraiment très pratique ! Grâce à lui, exit le bon vieux Bloc Notes et vive la coloration syntaxique, l'affichage des numéros de lignes, le débogueur intégré, et l'aide en ligne en mode graphique. On a également la possibilité d'ouvrir une console PowerShell sur une machine distance directement dans l'éditeur. On pourrait juste éventuellement lui reprocher le manque de la fonctionnalité « IntelliSense » comme dans l'éditeur de Visual Studio. Ceci étant, si cette fonctionnalité vous est chère, sachez qu'elle est apportée gratuitement par l'outil **PowerGUI Script Editor** de la société Quest Software.

Cependant, nous nous réjouissons de pouvoir disposer de cet éditeur graphique sur toutes les machines sur lesquelles PowerShell (v2 + Framework .NET 3.0) est installé.

Voici l'interface dans toute sa splendeur :



La console PowerShell ISE

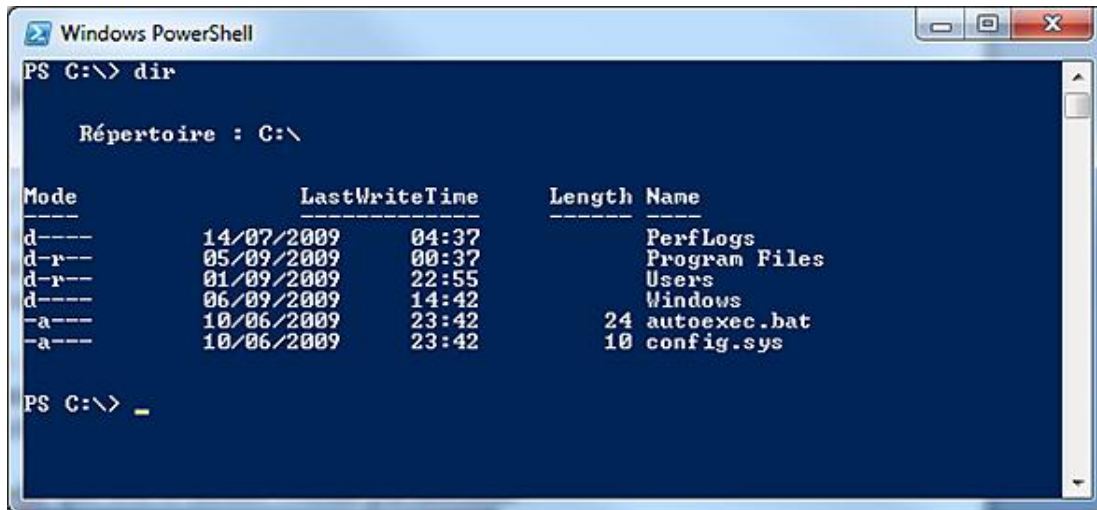
Comme vous l'aurez sans doute remarqué, la console est composée de plusieurs volets. Dans le volet principal se trouve l'éditeur de script dans lequel vont se loger des onglets. Cela permet de travailler sur plusieurs scripts à la fois. Un autre volet vous permettra de saisir directement des commandes interactives comme dans la console classique. Enfin dans le dernier volet vous trouverez la sortie d'exécution de vos scripts.

Une transition en douceur avec le passé

Si vous êtes déjà un utilisateur de PowerShell v1, alors soyez rassuré car vos scripts continueront en principe à fonctionner avec la version 2. En principe, car la version 2 apporte quelques nouveaux mots clés, commandes et variables et si par malchance vous les avez employés en tant que variable ou fonction dans vos scripts, vous pourriez rencontrer quelques dysfonctionnements. Mais ne vous affolez pas pour autant car dans la plupart des cas les scripts développés en version 1 fonctionnent parfaitement en version 2.

Quant aux inconditionnels du CMD, qu'ils se rassurent également : PowerShell ne fait pas table rase du passé. Pratiquement toutes les commandes qui étaient incluses dans CMD le sont aussi dans PowerShell ; certaines le sont sous forme d'alias, de fonctions, ou de fichiers externes. Ces derniers étant alors les commandes originales.

Prenons par exemple la commande `dir` que tout le monde connaît parfaitement :



```
Windows PowerShell
PS C:\> dir

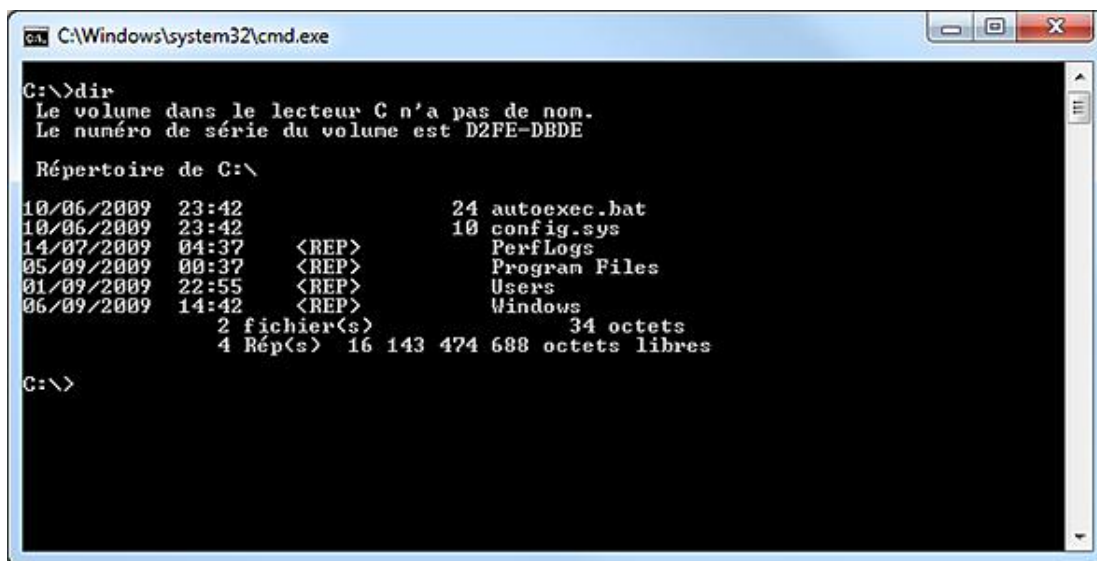
Répertoire : C:\

Mode                LastWriteTime         Length Name
----                -
d-----         14/07/2009         04:37      PerfLogs
d-r--         05/09/2009         00:37    Program Files
d-r--         01/09/2009         22:55        Users
d-----         06/09/2009         14:42      Windows
-a---         10/06/2009         23:42          24 autoexec.bat
-a---         10/06/2009         23:42          10 config.sys

PS C:\> _
```

Dir sous PowerShell

Et maintenant la même dans l'invite de commande CMD :



```
C:\Windows\system32\cmd.exe
C:\>dir
Le volume dans le lecteur C n'a pas de nom.
Le numéro de série du volume est D2FE-DBDE

Répertoire de C:\

10/06/2009  23:42                24 autoexec.bat
10/06/2009  23:42                10 config.sys
14/07/2009  04:37             <REP>      PerfLogs
05/09/2009  00:37             <REP>    Program Files
01/09/2009  22:55             <REP>        Users
06/09/2009  14:42             <REP>      Windows
                2 fichier(s)             34 octets
                4 Rép(s) 16 143 474 688 octets libres

C:\>
```

Dir sous CMD

Vous constaterez par vous-même qu'au premier abord la différence n'est pas flagrante, si ce n'est la couleur du fond qui change ainsi que la taille de la fenêtre, plus généreuse sous PowerShell. Pour les tâches courantes, telles que la navigation dans les répertoires et les fichiers, vous n'aurez donc pas besoin de connaître les vraies commandes PowerShell qui se cachent derrière les alias.

Voici une liste non exhaustive d'anciennes commandes que vous pouvez réutiliser dans PowerShell : **dir, md, cd, rd, move, ren, cls, copy**.

Les Unixiens ne seront pas perdus non plus car la plupart des commandes Unix de base fonctionnent grâce aux alias

PowerShell, tels que : **ls, mkdir, cp, mv, pwd, cat, mount, lp, ps**, etc.

Pour connaître la liste complète des alias disponibles, tapez la commande suivante : **Get-Alias**. Et pour les fonctions, tapez : **Get-Command -CommandType function**



Retrouvez la liste complète des alias et des fonctions en annexes Liste des alias et Liste des fonctions.

Nous vous invitons donc dès à présent à ne plus utiliser CMD.exe, y compris pour effectuer des tâches basiques. Ainsi vous vous familiariserez très rapidement avec PowerShell et apprendrez au fur et à mesure tout le nouveau jeu de commandes. Vous gagnerez comme cela très vite en compétence et en efficacité.

Les commandes de base

Avant toute chose, PowerShell n'est rien de plus qu'un environnement en lignes de commandes au service du système d'exploitation mais aussi et surtout au service des utilisateurs. En tant que tel, il est donc livré avec tout un jeu de commandes qu'il est bon de connaître. Ou tout du moins savoir comment les trouver ou les retrouver...

1. Constitution des commandes

Les commandes de PowerShell sont appelées « cmdlets » (pour command-applets). Pour notre part, comme il n'existe pas de traduction officielle, nous avons pris le parti de les nommer « **commandelettes** ». Elles sont pour la plupart d'entre elles constituées de la manière suivante : un verbe et un nom séparés par un tiret (-) : **verbe-nom**. Par exemple, **Get-Command**.

Le verbe (en anglais bien sûr) décrit l'action que l'on va appliquer sur le nom. Dans cet exemple on va récupérer (**Get**) les commandes (**Command**).

Avec PowerShell on trouve toute une série de verbes génériques : **Get**, **Set**, **Add**, **Remove**, etc. qui se combinent avec différents noms comme **Path**, **Variable**, **Item**, **Object**, etc.

Les noms constituant les commandes sont toujours au singulier ; et ceci est également vrai pour leurs paramètres.

On peut donc en croisant les verbes et les noms, se souvenir facilement de bon nombre de commandes. Notez que les commandes, ainsi que leurs paramètres associés, peuvent s'écrire indifféremment en majuscules ou en minuscules. L'analyseur de syntaxe PowerShell n'étant pas sensible à la casse.



Retrouvez la liste complète des commandelettes en annexe Liste des commandes.

2. Get-Command

Si vous ne deviez n'en retenir qu'une seule, alors retenez au moins celle-là : **Get-Command**.

```
PS > Get-Command -CommandType cmdlet
```

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...
Cmdlet	ConvertFrom-SecureString	ConvertFrom-SecureString [-S...
Cmdlet	Convert-Path	Convert-Path [-Path] <String...
Cmdlet	ConvertTo-Html	ConvertTo-Html [[-Property] ...
Cmdlet	ConvertTo-SecureString	ConvertTo-SecureString [-Str...
Cmdlet	Copy-Item	Copy-Item [-Path] <String[>...
Cmdlet	Copy-ItemProperty	Copy-ItemProperty[-Path] <S...
Cmdlet	Export-Alias	Export-Alias [-Path] <String...
...		
...		
...		
Cmdlet	Where-Object	Where-Object [-FilterScript]...
Cmdlet	Write-Debug	Write-Debug [-Message] Stri...
Cmdlet	Write-Error	Write-Error [-Message] <Stri...
Cmdlet	Write-Host	Write-Host [[-Object] <Objec...
Cmdlet	Write-Output	Write-Output [-InputObject] ...
Cmdlet	Write-Progress	Write-Progress [-Activity] <...
Cmdlet	Write-Verbose	Write-Verbose [-Message] <St...
Cmdlet	Write-Warning	Write-Warning [-Message] <St...

Get-Command vous permet de connaître toutes les commandes intégrées à PowerShell. Dans la première version de PowerShell, les commandes de base étaient au nombre de 129. À présent, dans la version 2, elles sont au nombre de 236. Pour le vérifier, vous pouvez taper :

```
PS > Get-Command -CommandType cmdlet | Measure-Object

Count      : 236
Average    :
Sum        :
Maximum    :
Minimum    :
Property   :
```

Si votre résultat diffère, c'est que vous avez probablement dû installer des commandelettes supplémentaires soit par le biais de *snap-ins*, de *fonctions avancées*, de *modules* (nous y reviendrons plus loin dans cet ouvrage) ou bien en ajoutant des rôles ou fonctionnalités si vous vous trouvez sur une plate-forme Windows Server.

Pour en savoir plus sur **Get-Command**, tapez la commande :

```
PS > Get-Help Get-Command -Detailed | more
```

ou

```
PS > Help Get-Command
```

Par exemple :

```
PS > Help Get-Command
```

NOM

Get-Command

RÉSUMÉ

Obtient des informations de base sur les applets de commande et d'autres éléments des commandes Windows PowerShell.

SYNTAXE

```
Get-Command [[-Name] <string[]>] [-CommandType {Alias | Function |
Filter | Cmdlet | ExternalScript | Application | Script | All}]
[[-ArgumentList] <Object[]>] [-Module <string[]>] [-Syntax] [-TotalCount
<int>] [<CommonParameters>]
Get-Command [-Noun <string[]>] [-Verb <string[]>] [[-ArgumentList]
<Object[]>] [-Module <string[]>] [-Syntax] [-TotalCount <int>]
[<CommonParameters>]
```

DESCRIPTION

L'applet de commande Get-Command obtient des informations de base sur les applets de commande et d'autres éléments des commandes Windows PowerShell de la session, tels qu'alias, fonctions, filtres, scripts et applications.

Get-Command obtient directement ses données du code d'une applet de commande, d'une fonction, d'un script ou d'un alias, contrairement à Get-Help, qui les obtient des fichiers de rubrique d'aide. Sans paramètres, « Get-Command » obtient toutes les applets de commande et fonctions de la session active. « Get-Command * » obtient tous les éléments Windows PowerShell et tous les fichiers autres que Windows PowerShell dans la variable d'environnement Path (\$env:path). Elle regroupe les fichiers dans le type de commande « Application ».

Vous pouvez utiliser le paramètre Module de Get-Command pour rechercher les commandes qui ont été ajoutées à la session en ajoutant un composant logiciel enfichable Windows PowerShell ou en important un module.

LIENS CONNEXES

Online version: <http://go.microsoft.com/fwlink/?LinkID=113309>
about_Command_Precedence
Get-Help
Get-PSDrive
Get-Member
Import-PSSession
Export-PSSession

REMARQUES

Pour consulter les exemples, tapez : "get-help Get-Command -examples".

Pour plus d'informations, tapez : "get-help Get-Command -detailed".

Pour obtenir des informations techniques, tapez :

"get-help Get-Command -full".

Cette ligne de commandes nous permet d'obtenir une aide détaillée sur l'utilisation de **Get-Command**. Nous voyons par exemple qu'il est possible d'utiliser le paramètre **-verb**. Voyons ce que pourrait donner ceci :

```
PS > Get-Command -Verb write
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Write-Debug	Write-Debug [-Message] ...
Cmdlet	Write-Error	Write-Error [-Message] ...
Cmdlet	Write-EventLog	Write-EventLog [-LogNam...
Cmdlet	Write-Host	Write-Host [[-Object] <...
Cmdlet	Write-Output	Write-Output [-InputObj...
Cmdlet	Write-Progress	Write-Progress [-Activi...
Cmdlet	Write-Verbose	Write-Verbose [-Message...
Cmdlet	Write-Warning	Write-Warning [-Message...

Nous venons d'obtenir la liste de toutes les commandes dont le verbe commence par **Write**. Nous verrons dans la prochaine partie comment sont structurées les commandes PowerShell.

De la même façon, nous pourrions obtenir la liste des commandes qui s'appliquent aux objets :

```
PS > Get-Command -Noun object
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Compare-Object	Compare-Object [-Refere...
Cmdlet	ForEach-Object	ForEach-Object [-Proces...
Cmdlet	Group-Object	Group-Object [[-Propert...
Cmdlet	Measure-Object	Measure-Object [[-Prope...
Cmdlet	New-Object	New-Object [-TypeName] ...
Cmdlet	Select-Object	Select-Object [[-Proper...
Cmdlet	Sort-Object	Sort-Object [[-Property...
Cmdlet	Tee-Object	Tee-Object [-FilePath] ...
Cmdlet	Where-Object	Where-Object [-FilterSc...

Nous pouvons également obtenir des commandes d'un certain type, dont les plus usitées sont : **Alias**, **Function**, **cmdlet**, **externalscript**, **application**.

Exemple :

```
PS > Get-Command -Commandtype alias
```

CommandType	Name	Definition
-----	----	-----
Alias	%	ForEach-Object
Alias	?	Where-Object
Alias	ac	Add-Content
Alias	asnp	Add-PSSnapin
Alias	cat	Get-Content
Alias	cd	Set-Location
Alias	chdir	Set-Location
Alias	clc	Clear-Content
Alias	clear	Clear-Host
Alias	cli	Clear-Item

Alias	clp	Clear-ItemProperty
Alias	cls	Clear-Host
Alias	clv	Clear-Variable
Alias	copy	Copy-Item
Alias	cp	Copy-Item
Alias	cpi	Copy-Item
Alias	cpp	Copy-ItemProperty
Alias	cvpa	Convert-Path
...		
...		
...		
Alias	spsv	Stop-Service
Alias	sv	Set-Variable
Alias	tee	Tee-Object
Alias	type	Get-Content
Alias	where	Where-Object
Alias	write	Write-Output

Si vous êtes à la recherche d'une commande dont vous ignorez le nom, mais si vous savez que la commande que vous recherchez doit vous fournir de l'information, il y a de fortes chances pour qu'elle commence par **Get**. Dans ces conditions, vous pouvez faire ceci : `Get-Command Get*` ou `Get-Command Get-*`.

```
PS > Get-Command Get-*
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Get-Acl	Get-Acl [-Path] <Strin...
Cmdlet	Get-Alias	Get-Alias [-Name] <Str...
Cmdlet	Get-AuthenticodeSignature	Get-AuthenticodeSignatu...
Cmdlet	Get-ChildItem	Get-ChildItem [-Path] ...
Cmdlet	Get-Command	Get-Command [-Argument...
Cmdlet	Get-ComputerRestorePoint	Get-ComputerRestorePoin...
Cmdlet	Get-Content	Get-Content [-Path] <St...
Cmdlet	Get-Counter	Get-Counter [-Counter]...
Cmdlet	Get-Credential	Get-Credential [-Creden...
Cmdlet	Get-Culture	Get-Culture [-Verbose] ...
Cmdlet	Get-Date	Get-Date [-Date] <Date...
Cmdlet	Get-Event	Get-Event [-SourceIden...
Cmdlet	Get-EventLog	Get-EventLog [-LogName]...
Cmdlet	Get-EventSubscriber	Get-EventSubscriber [-...
Cmdlet	Get-ExecutionPolicy	Get-ExecutionPolicy [-...

De la même façon, si vous savez que la commande que vous recherchez s'applique à des « items », vous pouvez essayer cela :

```
PS > Get-Command *-Item
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Clear-Item	Clear-Item [-Path] <Str...
Cmdlet	Copy-Item	Copy-Item [-Path] <Stri...
Cmdlet	Get-Item	Get-Item [-Path] <Strin...
Cmdlet	Invoke-Item	Invoke-Item [-Path] <St...
Cmdlet	Move-Item	Move-Item [-Path] <Stri...
Cmdlet	New-Item	New-Item [-Path] <Strin...
Cmdlet	Remove-Item	Remove-Item [-Path] <St...
Cmdlet	Rename-Item	Rename-Item [-Path] <St...
Cmdlet	Set-Item	Set-Item [-Path] <Strin...

Comme nous avons introduit la commande **Get-Help** dans l'un des exemples précédents, détaillons-la dès à présent.

3. Get-Help

Cette commande de base va nous permettre comme son nom l'indique d'obtenir de l'aide sur n'importe quelle commandelette, voire davantage !

Pour demander de l'aide sur une commande, vous pouvez le faire de différentes façons :

- `Get-HelpmaCommande`
- `HelpmaCommande`
- `maCommande -?`

`Get-HelpmaCommande` vous affiche l'aide standard.

Avec PowerShell, vous avez trois niveaux d'aide :

- l'aide standard,
- l'aide détaillée,
- l'aide complète.

Pour avoir accès à l'aide détaillée, ajoutez le paramètre **-Detailed**, soit `Get-HelpmaCommande-detailed`. Et pour l'aide complète, spécifiez le paramètre **-Full**, `Get-HelpmaCommande-full`.

Lorsque vous tapez `maCommande -?`, vous ne pouvez pas spécifier de niveau de détail, l'aide retournée est alors l'aide standard.



Nous vous recommandons de préférer l'utilisation de la commande `Help maCommande`, suivi du niveau de détail désiré (`-detailed` ou `-full`) car cela vous offre deux avantages intéressants : le premier, c'est que cela est plus court à taper, et le second, l'aide s'affichera page par page. `Help` est une fonction qui permet d'afficher le contenu de l'aide page par page. Pour l'essentiel, celle-ci se contente d'appeler **Get-Help** et de passer son contenu à **more**.



Si vous tapez simplement la commande `Help`, vous aurez alors accès à toutes les rubriques d'aide que PowerShell peut vous proposer. Essayez, vous serez surpris.

L'aide de PowerShell est particulièrement riche car elle donne également accès à de l'aide sur l'utilisation des tableaux, des opérateurs de comparaison, des boucles, du pipe, des fonctions, etc.

Pour découvrir toutes les rubriques possibles tapez : `help about_*`

Cette aide est très précieuse lorsque l'on développe un script et que l'on a oublié de prendre avec soi le merveilleux ouvrage que vous tenez entre les mains... ☺

```
PS > Help about_*
```

Name	Category	Synopsis
----	-----	-----
<code>about_Alias</code>	HelpFile	Utilisation d'autres n...
<code>about_Arithmetic_Operators</code>	HelpFile	Opérateurs pouvant êtr...
<code>about_Array</code>	HelpFile	Structure de données c...
<code>about_Assignment_Operators</code>	HelpFile	Opérateurs pouvant êtr...
<code>about_Associative_Array</code>	HelpFile	Structure de données c...
<code>about_Automatic_Variables</code>	HelpFile	Variables définies aut...
<code>about_Break</code>	HelpFile	Instruction permettant...
<code>about_Command_Search</code>	HelpFile	Explique comment Windo...
<code>about_Command_Syntax</code>	HelpFile	Format de commande dan...
<code>about_CommonParameters</code>	HelpFile	Paramètres que chaque ...
...		
...		
...		
<code>about_Special_Characters</code>	HelpFile	Caractères spéciaux co...
<code>about_Switch</code>	HelpFile	Utilisation de switch ...
<code>about_System_State</code>	HelpFile	Données gérées par Win...
<code>about_Types</code>	HelpFile	Extension du système d...
<code>about_Where</code>	HelpFile	Objets filtre basés su...
<code>about_While</code>	HelpFile	Instruction de langage...
<code>about_Wildcard</code>	HelpFile	Utilisation de aractè...

De base, il existe près d'une centaine de rubriques d'aide. Largement de quoi approfondir vos connaissances sur de nombreux sujets. Nous vous encourageons à la lire car elle est d'excellente qualité et en français de surcroît !

Prenons à présent un exemple afin d'observer comment l'aide se présente :

```
PS > Help Get-Item
NOM
    Get-Item

RÉSUMÉ
    Obtient l'élément à l'emplacement spécifié.

SYNTAXE
    Get-Item [-LiteralPath] <string[]> [-Credential <PSCredential>] [-Exclude <string[]>] [-Filter <string>] [-Force] [-Include <string[]>] [-UseTransaction] [<CommonParameters>]

    Get-Item [-Path] <string[]> [-Credential <PSCredential>] [-Exclude <string[]>] [-Filter <string>] [-Force] [-Include <string[]>] [-UseTransaction] [<CommonParameters>]

DESCRIPTION
    L'applet de commande Get-Item obtient l'élément à l'emplacement spécifié. Elle n'obtient pas le contenu de l'élément à cet emplacement, sauf si vous utilisez un caractère générique (*) pour inclure l'ensemble du contenu de l'élément.

    L'applet de commande Get-Item est utilisée par les fournisseurs Windows PowerShell pour vous permettre de parcourir les différents types de magasins de données.

LIENS CONNEXES
    Online version: http://go.microsoft.com/fwlink/?LinkID=113319
    about_Providers
    Clear-Item
    Copy-Item
    Invoke-Item
    Move-Item
    Set-Item
    New-Item
    Remove-Item
    Rename-Item

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

Une nouveauté introduite par PowerShell v2 est le lien vers la version « Online » de l'aide. Un copier/coller de l'URL située dans la rubrique liens connexes dans votre navigateur vous permettra de bénéficier de la toute dernière version de l'aide sur la commande recherchée. Ceci étant, l'aide en français n'est pas toujours disponible en ligne.

4. Get-Member

Celle-ci est probablement la commande la plus intéressante de toutes car elle permet de lister toutes les propriétés et méthodes d'un objet ainsi que son type. Notez qu'il n'est pas nécessaire lorsque l'on fait ses premiers pas avec PowerShell de savoir maîtriser cette commande. En effet celle-ci met en jeu le concept d'objets que nous aborderons un peu plus loin dans le livre. Vous pourrez revenir sur cette commande par la suite, une fois les bases acquises.

Grâce à **Get-Member** vous allez pouvoir épater vos collègues de travail car vous allez gagner un temps considérable dans l'écriture de vos scripts.

```
PS > $maVariable = 'Bonjour tout le monde !'
```

Nous venons de créer la variable **\$maVariable** et lui avons affecté une valeur de type chaîne (**String**). Vous remarquerez que nous n'avons pas eu besoin de la déclarer car PowerShell reconnaît automatiquement son type en fonction de son contenu. Une variable commence toujours par le caractère dollar. Nous discuterons en détail des variables dans le prochain chapitre.

Maintenant, imaginons que nous voulions faire des actions dessus, comme par exemple la convertir en majuscules ou bien compter le nombre de caractères qu'elle contient.

Pour faire cela habituellement dans tout langage de scripts ou de programmation nous devons nous référer à la documentation pour connaître les commandes qui permettent la manipulation des chaînes de caractères. Bien sûr en PowerShell nous pouvons faire de même, mais c'est maintenant que la commande **Get-Member** prend tout son sens et vous allez comprendre pourquoi...

■ Tapez maintenant :

```
PS > $maVariable | Get-Member

TypeName: System.String

Name      MemberType      Definition
----      -
Clone     Method          System.Object Clone()
CompareTo Method          int CompareTo(System.Object valu...
Contains  Method          bool Contains(string value)
CopyTo    Method          System.Void CopyTo(int sourceInd...
EndsWith  Method          bool EndsWith(string value), boo...
Equals    Method          bool Equals(System.Object obj), ...
GetEnumerator Method        System.CharEnumerator GetEnumera...
GetHashCode Method        int GetHashCode()
GetType   Method          type GetType()
GetTypeCode Method        System.TypeCode GetTypeCode()
IndexOf   Method          int IndexOf(char value), int Ind...
IndexOfAny Method        int IndexOfAny(char[] anyOf), in...
Insert    Method          string Insert(int startIndex, st...
IsNormalized Method        bool IsNormalized(), bool IsNorm...
LastIndexOf Method        int LastIndexOf(char value), int...
LastIndexOfAny Method        int LastIndexOfAny(char[] anyOf)...
Normalize Method          string Normalize(), string Norma...
PadLeft   Method          string PadLeft(int totalWidth), ...
PadRight  Method          string PadRight(int totalWidth),...
Remove    Method          string Remove(int startIndex, in...
Replace   Method          string Replace(char oldChar, cha...
Split     Method          string[] Split(Params char[] sep...
StartsWith Method        bool StartsWith(string value), b...
Substring Method          string Substring(int startIndex)...
ToArray   Method          char[] ToCharArray(), char[] ToC...
ToLower   Method          string ToLower(), string ToLower...
ToLowerInvariant Method      string ToLowerInvariant()
ToString  Method          string ToString(), string ToStri...
ToUpper   Method          string ToUpper(), string ToUpper...
ToUpperInvariant Method      string ToUpperInvariant()
Trim      Method          string Trim(Params char[] trimCh...
TrimEnd   Method          string TrimEnd(Params char[] tri...
TrimStart Method          string TrimStart(Params char[] t...
Chars     ParameterizedProperty char Chars(int index) {get;}
Length    Property        System.Int32 Length {get;}
```

Nous voyons apparaître plusieurs éléments particulièrement intéressants :

- Le champ **TypeName** nous indique le type de notre variable. Soit comme on le supposait un type **String**.
- Une liste de noms de méthodes, de propriétés, et leur définition associée.

Sans gros effort nous pouvons donc imaginer que la méthode **ToUpper** va nous permettre de passer la chaîne contenue dans **\$maVariable** en majuscules.

```
PS > $maVariable.ToUpper()
```

BONJOUR TOUT LE MONDE !

De la même façon, on peut se dire que la propriété **Length** va nous donner le nombre de caractères contenus dans notre chaîne.

```
PS > $maVariable.Length
```

```
23
```

En bref, cette commandelette est vraiment indispensable dès lors qu'on y a goûté...



Une erreur classique lorsque l'on débute avec PowerShell est d'oublier les parenthèses de fin lorsque l'on fait appel à une méthode. Par exemple, si vous tapez **\$maVariable.ToUpper**, vous n'obtiendrez pas le résultat escompté car PowerShell affichera la définition de la méthode. Soyez donc vigilants sur ce point.



PowerShell v2 apporte à la commande **Get-Member** le commutateur **-Force**. Celui-ci permet l'affichage de propriétés et méthodes avancées sur les objets. Il n'est pas nécessaire de vous en soucier pour l'instant ; nous vous en reparlerons dans le chapitre Maîtrise du Shell.

Navigation dans les répertoires et les fichiers

1. Les nouvelles commandes

Nous avons vu que nous pouvions utiliser les bonnes vieilles commandes DOS/CMD afin de nous déplacer dans une hiérarchie de dossiers. Bien que cela soit toujours possible et fasse gagner du temps à celui qui les emploie, cela ne lui élève pas son niveau de connaissance de PowerShell.

Lorsque vous tapez **DIR** en PowerShell, vous faites en réalité appel à un alias. L'alias vous fait exécuter la commande **Get-ChildItem**. Pour le vérifier, tapez la commande suivante :

PS > Get-Alias dir		
CommandType	Name	Definition
-----	----	-----
Alias	dir	Get-ChildItem

Voici un tableau récapitulatif des principales commandes CMD et de leurs équivalents en PowerShell.

DOS/CMD	Équivalent PowerShell	Commandelette PowerShell	Description
DIR	DIR	Get-ChildItem	Lister le contenu d'un répertoire.
CD	CD	Set-Location	Changer de répertoire courant.
MD	MD	New-Item	Créer un fichier/répertoire.
RD	RD	Remove-Item	Supprimer un fichier/répertoire.
MOVE	MOVE	Move-Item	Déplacer un fichier/répertoire.
REN	REN	Rename-Item	Renommer un fichier/répertoire.
COPY	COPY	Copy-Item	Copier un fichier/répertoire.

Comme l'objet de cet ouvrage est l'apprentissage de PowerShell, nous nous efforcerons à ne plus utiliser les anciennes commandes DOS ; et nous vous encourageons à en faire de même !

2. Get-ChildItem (Alias : gci, ls, dir)

Cette commandelette nous permet d'obtenir la liste des fichiers et dossiers présents dans le système de fichiers.

Par exemple, observons le résultat de la commande suivante :

PS > gci c:\			
Répertoire : C:\			
Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	14/07/2009 04:37		PerfLogs
d-r--	05/09/2009 00:37		Program Files
d-r--	01/09/2009 22:55		Users
d----	06/09/2009 14:42		Windows
-a---	10/06/2009 23:42	24	autoexec.bat
-a---	10/06/2009 23:42	10	config.sys

Au premier regard, ce qui attire l'œil, c'est le nom donné à chaque colonne ; mais nous pouvons aussi observer la colonne Mode.

Celle-ci indique la nature des objets à l'intérieur du système de fichiers, voici les valeurs possibles :

- d : pour un répertoire,
- a : pour archive,
- r : pour un objet en lecture seule,
- h : pour un objet caché,
- s : pour un objet système.



Pour afficher les fichiers cachés, ajoutez à la commande **Get-Childitem** le paramètre **-Force**.

Exemple :

```
PS > gci c:\ -Force

Répertoire : C:\

Mode                LastWriteTime         Length Name
----                -
d--hs              01/09/2009    22:55             $Recycle.Bin
d--hs              14/07/2009    06:53      Documents and Settings
d-rh-              01/09/2009    23:19             MS0Cache
d---              14/07/2009    04:37             PerfLogs
d-r--              05/09/2009    00:37          Program Files
d--h-              04/09/2009    00:24          ProgramData
d--hs              01/09/2009    22:55             Recovery
d--hs              06/09/2009    10:10      System Volume Information
d-r--              01/09/2009    22:55             Users
d---              06/09/2009    14:42             Windows
-a---              10/06/2009    23:42           24 autoexec.bat
-a---              10/06/2009    23:42           10 config.sys
-a-hs              06/09/2009    15:18 1602318336 hiberfil.sys
-a-hs              06/09/2009    15:18 2136428544 pagefile.sys
```

Pour revenir sur le nom des colonnes, ceux-ci indiquent en réalité le nom d'une propriété de fichier ou de répertoire. Nous vous avons expliqué que PowerShell était basé sur des objets (contrairement à l'invite de commande), et bien vous allez pouvoir en juger par vous-même !

Voici quelques exemples :

- Afficher (récursivement) tous les fichiers ayant l'extension .log contenus à l'intérieur d'une arborescence :

```
PS > Get-ChildItem c:\temp\* -Include *.log -Recurse
```

- Obtenir le nom des fichiers dont la taille est supérieure à 32 Ko :

```
PS > Get-ChildItem | Where-Object {$_.Length -gt 32KB}
```

- Obtenir les fichiers dont la date de dernier enregistrement est postérieure au 01/01/2009 :

```
PS > Get-ChildItem | Where-Object {$_.LastWriteTime -gt '01/01/2009'}
```

Attention : la date est toujours au format américain, soit MM/JJ/AAAA (cf chapitre Maîtrise du Shell - Les dates).

Quelques explications :

Le pipe « | » permet de passer un ou plusieurs objets à la commande qui suit. Dans nos exemples nous passons chaque objet à la commandelette **Where-Object** (appelée aussi clause, ou encore filtre). Cette dernière va analyser les propriétés **Length** ou **LastWriteTime** (selon l'exemple), les comparer et retourner les objets correspondants si le test est vrai. Le « \$_ » indique qu'on traite l'objet courant. Pour plus d'information sur le pipe, reportez-vous au chapitre Fondamentaux - Redirections et Pipeline).

➤ Nous avons fait appel dans notre premier exemple à un quantificateur d'octets (le 32 KB). PowerShell intègre nativement ces quantificateurs d'octets pour simplifier l'écriture des tailles mémoire. Ils sont les suivants : KB (Ko), MB (Mo), GB (Go), TB (To) et PB (Po). N'oubliez pas qu'un 1 Ko équivaut à 1024 octets et non à 1000 comme on le voit bien (trop) souvent !

3. Set-Location (Alias : sl, cd, chdir)

Il n'y a pas grand-chose à dire sur cette commande, si ce n'est qu'elle nous permet de nous déplacer dans une arborescence de dossiers. Par exemple :

```
PS > Set-Location D:\
```

Comme dans CMD, on peut utiliser des chemins relatifs ainsi que les raccourcis « .. » et « \ », pour désigner respectivement le répertoire parent et le répertoire racine du disque en cours. Cependant dans PowerShell v1, contrairement à CMD qui supporte de coller « cd » et le raccourci (par exemple « cd.. »), il faut obligatoirement insérer un espace entre **Set-Location** (ou son alias) et le chemin, raccourci ou non (exemple « cd .. » - notez l'espace entre « cd » et « .. »).

➤ Si vraiment vous ne pouvez vous empêcher d'utiliser instinctivement « cd.. » (en collant les « .. » au « cd ») et que ça vous dérange de ne pouvoir le faire, il existe la solution de contournement suivante :

```
PS > function cd.. {Set-Location ..}
```

Cela crée une fonction du nom de « cd.. » qui exécute la commandelette **Set-Location** avec le paramètre « .. ». Nous ne pouvons pas créer d'alias, car un alias fait une correspondance « un pour un » entre un nom et une commande (ou fonction).

➤ Cela n'est plus le cas avec PowerShell v2 car cette fonction existe maintenant nativement.

4. Get-Location (Alias : gl, pwd)

Cette commande retourne l'emplacement actuel à l'intérieur d'une arborescence.

Voici le résultat d'exécution de **Get-Location** :

```
PS > Get-Location

Path
----
C:\Users
```

Et voici comment faire pour récupérer dans une variable le chemin (path) de l'emplacement courant en une seule ligne de commande.

```
PS > $chemin = (Get-Location).Path
```

Nous venons de stocker la valeur du chemin courant, en l'occurrence « C:\users » dans la variable **\$chemin**. Maintenant pour l'afficher, rien de plus simple, tapez seulement :

```
PS > $chemin

C:\Users
```


5. New-Item (Alias : ni, md)

Cette commandelette va nous permettre de créer des répertoires, à l'instar de la commande « md » en CMD, mais aussi des fichiers.

Examinons de plus près quelques-uns de ses paramètres :

Paramètre	Description
Path	Chemin d'accès de l'élément à créer (ex : C:\Temp).
ItemType	Type d'élément à créer : file pour un fichier, directory pour un dossier.
Name	Nom du nouvel élément à créer.
Value	Contenu de l'élément à créer (ex : "bonjour !" dans le cas d'un fichier texte).

a. Création d'un répertoire

```
PS > New-Item -ItemType directory -Name temp

Répertoire : C:\

Mode                LastWriteTime         Length Name
----                -
d-----          06/09/2009         17:37         temp
```



Si notre dossier avait contenu un espace, nous aurions dû le mettre entre guillemets. Par exemple :

```
PS > New-Item -Name 'dossier Test' -ItemType directory
```

b. Création d'un fichier

Imaginons que nous voulions créer un fichier nommé « monFichier.txt » qui contiendrait la phrase suivante « Vive PowerShell ! ». La commande sera la suivante :

```
PS > New-Item -Name monFichier.txt -ItemType file -Value 'Vive PowerShell'

Répertoire : C:\

Mode                LastWriteTime         Length Name
----                -
-a---          06/09/2009         17:39         17 monFichier.txt
```



Vous découvrirez dans le chapitre Maîtrise du Shell qu'il existe d'autres façons, encore plus pratiques, pour créer des fichiers. Sachez que les opérateurs de redirection « > » et « >> » fonctionnent aussi très bien avec PowerShell.

6. Remove-Item (Alias : ri, rm, rmdir, rd, erase, del)

La commandelette **Remove-Item**, comme son nom l'indique, permet de supprimer des fichiers ou des dossiers.

Nous pouvons l'utiliser de plusieurs manières :

```
PS > Remove-Item c:\temp\*.log
```

Dans cet exemple, nous venons de supprimer tous les fichiers .log contenus dans le répertoire c:\temp.

```
PS > Get-ChildItem c:\temp\* -Include *.txt -Recurse | Remove-Item
```

Ici nous supprimons sélectivement tous les fichiers contenus dans une arborescence de dossiers dont l'extension est « .txt ». Cette syntaxe est très pratique car on peut la construire petit à petit ; on liste d'abord les fichiers à supprimer, puis on les passe via le pipe à la commande **Remove-Item**.

Pour supprimer un fichier système, masqué ou en lecture seule, il suffit tout simplement d'utiliser le paramètre **-force**, comme dans l'exemple ci-dessous :

```
PS > Remove-Item fichierASupprimer.txt -Force
```

 **Remove-Item** possède aussi le paramètre **-whatif** ; celui-ci permet de dire ce que va faire la commande mais sans réellement l'exécuter. C'est en quelque sorte un mode simulation. Un autre paramètre intéressant est **-confirm**. Grâce à lui PowerShell vous demandera une confirmation pour chaque fichier à supprimer ; ce qui n'est pas le cas par défaut.

7. Move-Item (Alias : mi, move, mv)

Cette commande permet de déplacer un fichier ou un répertoire d'un emplacement vers un autre emplacement. Dans le cas d'un répertoire, le contenu est également déplacé. **Move-Item** permet en outre d'effectuer un renommage de l'objet manipulé.

a. Déplacement de fichiers

Exemple :

*Déplacer des fichiers *.jpg du répertoire courant vers le dossier « mes photos ».*

```
PS > Move-Item -Path *.jpg -destination 'mes photos'
```

Ou

```
PS > Move-Item *.jpg 'mes photos'
```

Nous avons, dans la première ligne de commandes, spécifié explicitement tous les paramètres. Alors que dans la seconde, nous nous sommes contentés du minimum cependant le résultat est le même. Cela est possible car l'interpréteur de commandes PowerShell est relativement « intelligent ». Lorsqu'il analyse une commande alors que les noms des paramètres ne sont pas renseignés, il va aller regarder la définition de la commande et passer au premier paramètre la première valeur, puis la seconde valeur au second paramètre, et ainsi de suite jusqu'à ce qu'il n'y ait plus de valeurs à transmettre.

 Pour que l'exemple ci-dessus fonctionne, il faudrait qu'au préalable nous ayons créé le dossier « mes photos ». Ceci étant, il est possible de forcer la création du dossier de destination en utilisant le paramètre **-force**.

b. Déplacement d'un répertoire

Le déplacement d'un répertoire est similaire au déplacement de fichiers.

Prenons l'exemple suivant :

```
PS > Move-Item 'mes photos' 'mes nouvelles photos'
```

Dans ce cas, nous avons déplacé l'intégralité du répertoire « mes photos » dans le répertoire « mes nouvelles photos ». Comment ferions-nous maintenant pour renommer le dossier « mes photos » en « mes nouvelles photos ».

» ?

Réponse :

```
PS > Move-Item 'mes photos' 'mes nouvelles photos'
```

C'est très curieux nous direz-vous car il s'agit de la même ligne de commande. Vous avez tout à fait raison, mais il s'agit d'un fonctionnement normal. La seule différence vient du fait que selon le résultat souhaité il vous faudra créer ou non au préalable le répertoire de destination. Si vous omettez de le faire, vous effectuerez un renommage du dossier.

8. Rename-Item (Alias : ren, rni)

L'objectif de cette commande est de renommer un fichier ou dossier. Celle-ci n'est que moyennement utile dans la mesure où elle fait double emploi avec sa cousine **Move-Item**. Ceci étant, il y a peut-être certains cas, que nous ne connaissons pas encore, dans lesquels elle trouverait son utilité... Peut-être pour éviter de confondre renommage et déplacement comme dans l'exemple précédent ?

a. Renommer un fichier

Exemple :

Renommer le fichier monFichierDeLog.txt en ficlog.txt.

```
PS > Rename-Item -Path c:\temp\monFichierDeLog.txt -Newname ficlog.txt
```

Ou

```
PS > Rename-Item c:\temp\monFichierDeLog.txt ficlog.txt
```

b. Renommer un dossier

Exemple :

Renommer le répertoire monDossier1 en monDossier2.

```
PS > Rename-Item -Path c:\temp\monDossier1 -Newname monDossier2
```

Ou

```
PS > Rename-Item c:\temp\monDossier1 monDossier2
```

9. Copy-Item (Alias : cpi, cp, copy)

Grâce à cette commande, nous allons pouvoir copier des fichiers ou des répertoires, voire les deux à la fois.

Quelques exemples :

Copie un fichier d'un répertoire source vers un répertoire destination :

```
PS > Copy-Item -Path c:\temp\ficLog.txt -destination d:\logs
```

Ou

```
PS > Copy-Item c:\temp\ficLog.txt d:\logs
```

Copie d'une arborescence de répertoires (c'est-à-dire avec tous les sous-dossiers et fichiers) :

```
PS > Copy-Item -Path RepSource -Destination RepDest -Recurse
```



Copy-Item crée automatiquement le répertoire de destination s'il n'existe pas.

10. Ce qu'on ne vous a pas dit sur la navigation : les fournisseurs

Maintenant que vous êtes familier avec le jeu de commandes qui permet de naviguer et de gérer une arborescence de fichiers et de dossiers, nous pouvons vous avouer que celui-ci permet également bien d'autres choses...

Toutes les commandes que nous avons vues précédemment permettent la manipulation :

- de la base de registres (valeurs et clés),
- de variables,
- des variables d'environnement,
- des alias,
- de la base des certificats X509 de votre ordinateur,
- des fonctions,
- et enfin du système de fichiers (que nous venons de détailler).



Un certificat vous permet de signer et/ou de chiffrer des données.

C'est ce qui explique la généricité du nom des commandes *-Item, dans la mesure où un « item » peut représenter par exemple un fichier, un dossier ou une clé de registre.

Tous les points que nous venons d'énumérer ci-dessus, vont être accessibles par le biais de ce que l'on appelle dans le jargon PowerShell des « fournisseurs » (on rencontre également très couramment le terme Provider ou PSProvider). Comme vous le constatez, ils sont au nombre de huit. Afin d'en obtenir la liste et les détails associés, tapez la commande **Get-PSProvider**.

```
PS > Get-PSProvider
```

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

L'accès aux contenus des fournisseurs se fait au moyen d'un « lecteur ». Voici la liste des lecteurs intégrés : **Alias**, **Env**, **A**, **C**, **D**, **E**,..., **Z**, **Function**, **HKLM**, **HKCU**, **Variable**, **Cert**, **WSMAN** (le nombre de lecteurs exploitables de type FileSystem dépend de chaque ordinateur, mais par défaut tous sont créés).

La navigation à l'intérieur de ces lecteurs se fait exactement de la même manière que pour explorer un système de fichiers sur un disque dur. Pour les utiliser, rien de plus simple, il suffit d'utiliser la syntaxe suivante : **Get-ChildItem Lecteur_du_fournisseur** :

Par exemple :

- Get-ChildItem Alias:
- Get-ChildItem Env:
- Get-ChildItem C:
- Get-ChildItem Function:
- Get-ChildItem HKLM:
- Get-ChildItem Variable:
- Get-ChildItem Cert:
- Get-ChildItem WSMAN:

Ces exemples nous permettent de lister le contenu de chacun des fournisseurs. Ceci étant, comme toute lettre de lecteur, nous pouvons entrer dedans et en explorer le contenu. Pour ce faire, essayons les commandes suivantes :

```
PS > Get-ChildItem Env:

Name                           Value
----                           -
ALLUSERSPROFILE                C:\ProgramData
APPDATA                        C:\Users\Administrator\AppData\Roaming
CLIENTNAME                     WIN7_BUREAU
CommonProgramFiles             C:\Program Files\Common Files
CommonProgramFiles(x86)        C:\Program Files (x86)\Common Files
CommonProgramW6432             C:\Program Files\Common Files
COMPUTERNAME                   W2K8R2SRV
ComSpec                        C:\Windows\system32\cmd.exe
FP_NO_HOST_CHECK               NO
HOMEDRIVE                      C:
HOMEPATH                      \Users\Administrator
LOCALAPPDATA                   C:\Users\Administrator\AppData\Local
LOGONSERVER                    \\W2K8R2SRV
NUMBER_OF_PROCESSORS           4
OS                             Windows_NT
Path                           %SystemRoot%\system32\WindowsPowerShell\v1...
PATHEXT                        .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WS...
PROCESSOR_ARCHITECTURE         AMD64
PROCESSOR_IDENTIFIER           Intel64 Family 6 Model 15 Stepping 11, Gen...
PROCESSOR_LEVEL                6
PROCESSOR_REVISION             0f0b
ProgramData                   C:\ProgramData
ProgramFiles                   C:\Program Files
ProgramFiles(x86)              C:\Program Files (x86)
ProgramW6432                   C:\Program Files
PSModulePath                   C:\Users\Administrator\Documents\WindowsPo...
PUBLIC                         C:\Users\Public
SESSIONNAME                    RDP-Tcp#0
SystemDrive                    C:
SystemRoot                     C:\Windows
TEMP                           C:\Users\ADMINI~1\AppData\Local\Temp\2
TMP                             C:\Users\ADMINI~1\AppData\Local\Temp\2
USERDOMAIN                     W2K8R2SRV
USERNAME                      Administrator
USERPROFILE                    C:\Users\Administrator
windir                         C:\Windows
```

Une fois à l'intérieur d'un fournisseur, nous pouvons utiliser la plupart des commandes vues précédemment, telles que : **New-Item**, **Remove-Item**, **Copy-Item**, **Rename-Item**, etc.

Dans l'exemple précédent, si nous nous étions positionnés à l'intérieur du fournisseur « Environment » (avec la commande « **cd Env** : »), l'utilisation de **New-Item** nous permettrait de créer une nouvelle variable d'environnement, et à l'inverse, **Remove-Item** nous permettrait d'en supprimer une.

Par exemple, pour créer la variable **varTest** :

```
PS > Set-Location Env:
PS > New-Item -Path . -Name varTest -Value 'Variable de test'
```

Name	Value
----	-----
varTest	Variable de test

Et pour la supprimer :

```
PS > Remove-Item Env:varTest
```

Pour obtenir simplement le contenu d'une variable, faites comme ceci si vous vous trouvez dans le fournisseur des variables d'environnement : `Get-Content maVariable`.

Sinon faites comme cela : `Get-Content Env:maVariable`

Exemple :

```
PS > Get-Content Env:windir
C:\Windows
```

Voilà nous venons de terminer l'introduction sur les fournisseurs ; vous les retrouverez tout au long de cet ouvrage car leur utilisation est fréquente. Ils vont nous simplifier considérablement la vie dans l'écriture de scripts.



Notez qu'il est également possible de créer vos propres fournisseurs ou d'installer des fournisseurs tiers développés par d'autres personnes.



Pour obtenir de l'aide très détaillée sur le fonctionnement de chaque fournisseur, utilisez la commande **help fournisseur**.

Exemple :

```
PS > help env
```

ou

```
PS > help wsman
```

Formatage de l'affichage

Faire une partie sur le thème du formatage de l'affichage du résultat des commandes peut certainement vous surprendre mais sachez qu'étant donné le caractère objet de PowerShell cela est indispensable et vous allez comprendre pourquoi.

Si vous vous demandez également pourquoi la fenêtre de la console PowerShell est plus généreuse en dimensions que celle de CMD, alors cette partie devrait répondre à vos attentes.

Étant donné que PowerShell possède la faculté intrinsèque de manipuler des objets et qu'il ne s'en prive pas, tout ce qui s'affiche à l'écran lors de l'exécution d'une commande n'est en réalité qu'une sélection de quelques propriétés. Le choix de ces propriétés, que nous appellerons « propriétés par défaut » a été réalisé de façon arbitraire par les créateurs de PowerShell. Nous pouvons les saluer au passage car leur choix est finalement assez bon. Quoi qu'il en soit, le nombre de propriétés à afficher dépend de la taille de la fenêtre PowerShell. Ce nombre dépend aussi de ce qu'est prêt à voir l'utilisateur final, car si pour l'équivalent d'un simple « dir » vous avez en retour quinze propriétés pour chaque fichier, cela serait très vite pénible à interpréter.

Restons donc sur l'exemple de « **dir** » ou plutôt de **Get-ChildItem**.

```
PS > Get-ChildItem c:\

Répertoire : C:\

Mode                LastWriteTime         Length Name
----                -
d-----         14/07/2009         04:37         PerfLogs
d-r--         05/09/2009         00:37         Program Files
d-r--         01/09/2009         22:55         Users
d-----         06/09/2009         14:42         Windows
-a---         10/06/2009         23:42         24 autoexec.bat
-a---         10/06/2009         23:42         10 config.sys
```

Nous pouvons observer que cette commande nous renvoie les propriétés suivantes : **Mode**, **LastWriteTime**, **Length**, et **Name**.

Cet affichage est l'affichage par défaut que l'on obtient sans ajouter de paramètres particuliers à notre commande **Get-ChildItem** ; il s'agit ici d'un affichage tabulaire.

Sachez qu'avec PowerShell vous disposez maintenant de commandes spécifiques pour le formatage de l'affichage. Elles sont au nombre de quatre et nous en détaillerons trois d'entre elles :

Nom	Alias	Description
Format-List	fl	Affiche les propriétés sous forme de liste.
Format-Table	ft	Affiche les propriétés sous forme tabulaire.
Format-Wide	fw	Affiche une seule propriété au format large table.
Format-Custom	fc	Affichage personnalisé des propriétés.

➤ Nous ne parlerons pas de **Format-Custom** car l'usage de cette commande est complexe et très particulier. De plus, elle n'apporte rien d'intéressant dans un cadre normal d'utilisation de PowerShell.

1. Format-List

Cette commande de formatage va nous permettre d'afficher les propriétés des objets sous forme de liste. C'est-à-dire que chaque propriété de chaque objet sera affichée sur une ligne distincte.

- Continuons sur l'exemple précédent, en essayant la commande suivante : `Get-ChildItem | Format-List`

```
PS > Get-ChildItem c:\ | Format-List
```

```
Name           : PerfLogs
CreationTime    : 14/07/2009 04:37:05
LastWriteTime   : 14/07/2009 04:37:05
LastAccessTime  : 14/07/2009 04:37:05
```

```
Name           : Program Files
CreationTime    : 14/07/2009 04:37:05
LastWriteTime   : 05/09/2009 00:37:14
LastAccessTime  : 05/09/2009 00:37:14
```

```
Name           : Users
CreationTime    : 14/07/2009 04:37:05
LastWriteTime   : 01/09/2009 22:55:24
LastAccessTime  : 01/09/2009 22:55:24
```

```
Name           : Windows
CreationTime    : 14/07/2009 04:37:05
LastWriteTime   : 06/09/2009 14:42:56
LastAccessTime  : 06/09/2009 14:42:56
```

```
Name           : autoexec.bat
Length          : 24
CreationTime    : 14/07/2009 04:04:04
LastWriteTime   : 10/06/2009 23:42:20
LastAccessTime  : 14/07/2009 04:04:04
VersionInfo     :
```

```
Name           : config.sys
Length          : 10
CreationTime    : 14/07/2009 04:04:04
LastWriteTime   : 10/06/2009 23:42:20
LastAccessTime  : 14/07/2009 04:04:04
VersionInfo     :
```

En observant attentivement le résultat de cette commande, nous pouvons nous rendre compte que nous listons des propriétés différentes que lors de l'exécution de `Get-ChildItem` sans paramètres. En effet nous avons « perdu » la propriété `mode`, et nous avons obtenu en plus les propriétés `CreationTime`, `LastAccessTime` et `VersionInfo`.

De plus nous pouvons remarquer que les propriétés s'affichent les unes en dessous des autres, et que chaque objet est séparé de l'objet qui le précède par une ligne vide.

a. Affichage sélectif des propriétés d'un objet

Le paramètre le plus fréquemment utilisé avec `Format-List` est le paramètre `-Property`. Celui-ci permet de n'afficher que certaines propriétés, et ce par ordre d'apparition derrière ce paramètre.

Par exemple, pour afficher les propriétés « `Name` » et « `Length` » des dossiers et fichiers contenus dans le répertoire `c:\`, nous pourrions écrire ceci :

```
PS > Get-ChildItem c:\ | Format-List -Property Name, Length
```

```
Name : PerfLogs
```

```
Name : Program Files
```

```
Name : Users
```

```
Name : Windows
```

```
Name   : autoexec.bat
Length : 24
```

```
Name   : config.sys
Length : 10
```


Nous pouvons remarquer dans notre exemple que la propriété longueur (Length) n'est disponible que pour les objets de type fichier.

Autre exemple, pour afficher sélectivement certaines propriétés des services Windows :

```
PS > Get-Service | Format-List -Property Name, Displayname, Status

Name           : AeLookupSvc
DisplayName     : Expérience d'application
Status         : Running

Name           : ALG
DisplayName     : Service de la passerelle de la couche Application
Status         : Stopped

Name           : Appinfo
DisplayName     : Informations d'application
Status         : Stopped
...
```

b. Affichage de toutes les propriétés disponibles d'un objet

Nous allons maintenant afficher toutes les propriétés d'un fichier (ou plutôt devrait-on dire d'un objet de type fichier) grâce à la commande suivante : `Get-ChildItemmonFichier| Format-List *`

Grâce à l'utilisation du caractère générique « * » nous listerons toutes les propriétés d'un objet. Nous ne sommes donc plus limités à l'affichage des propriétés par défaut.

```
PS > Get-ChildItem config.sys | Format-List *

PSPath           : Microsoft.PowerShell.Core\FileSystem::C:\config.sys
PSParentPath     : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName      : config.sys
PSDrive          : C
PSProvider       : Microsoft.PowerShell.Core\FileSystem
PSIsContainer    : False
VersionInfo      : File:           C:\config.sys
                  InternalName:
                  OriginalFilename:
                  FileVersion:
                  FileDescription:
                  Product:
                  ProductVersion:
                  Debug:           False
                  Patched:        False
                  PreRelease:     False
                  PrivateBuild:   False
                  SpecialBuild:   False
                  Language:
BaseName         : config
Mode             : -a---
Name             : config.sys
Length           : 10
DirectoryName    : C:\
Directory        : C:\
IsReadOnly       : False
Exists           : True
FullName         : C:\config.sys
Extension        : .sys
CreationTime     : 14/07/2009 04:04:04
CreationTimeUtc  : 14/07/2009 02:04:04
LastAccessTime   : 14/07/2009 04:04:04
LastAccessTimeUtc : 14/07/2009 02:04:04
LastWriteTime    : 10/06/2009 23:42:20
LastWriteTimeUtc : 10/06/2009 21:42:20
Attributes       : Archive
```

c. Obtenir une seule propriété d'un objet

À présent, nous voudrions connaître uniquement la date de création du fichier **config.sys**. Pour ce faire, utilisons la propriété **CreationTime**.

```
PS > (Get-ChildItem config.sys).CreationTime  
mardi 14 juillet 2009 04:04:04
```

Maintenant si nous voulions affecter cette propriété à une variable, nous pourrions utiliser la ligne de commandes suivante :

```
PS > $maVariable = (Get-ChildItem config.sys).CreationTime  
PS > $maVariable  
mardi 14 juillet 2009 04:04:04
```



L'avantage principal d'utiliser la commande **Format-List** par rapport à un affichage de type tableau (**Format-Table**), c'est que les valeurs des propriétés disposent de davantage de place à l'écran pour s'afficher, et donc ne sont pas tronquées. L'autre intérêt, et non des moindres, est de pouvoir lister toutes les propriétés d'un objet grâce au caractère générique « * ». Il est également possible d'utiliser le joker sur une partie du nom des propriétés : **gci | format-list name, *time** permet en plus du nom d'afficher toutes les propriétés dont le nom se termine par « time ».

Exemple :

```
PS > Get-ChildItem config.sys | Format-List name,*time  
  
Name           : config.sys  
CreationTime    : 14/07/2009 04:04:04  
LastAccessTime  : 14/07/2009 04:04:04  
LastWriteTime   : 10/06/2009 23:42:20
```



Une fois les propriétés d'un objet connues, vous aurez peut-être envie de les modifier. Pour ce faire, le plus simple est d'utiliser les méthodes associées à cet objet. Pour les découvrir il faut utiliser la commande **Get-Member**. Si nous reprenons notre exemple précédent, nous pourrions utiliser la commande suivante pour lister les méthodes associées à un objet fichier :

```
PS > Get-ChildItem config.sys | Get-Member -MemberType method
```

2. Format-Table

La commande **Format-Table** permet d'afficher les propriétés d'objets sous forme de tableau. Ce format est très pratique car il offre une vue synthétique ; d'ailleurs ce n'est certainement pas un hasard si la plupart des commandelettes retournent leur résultat sous ce format.

Tout comme **Format-List**, l'exécution de cette commande sans spécifier de paramètres, renvoie une liste de propriétés par défaut.



La liste des propriétés par défaut diffère en fonction du type d'objet à afficher. Nous verrons par la suite, dans le chapitre Maîtrise du Shell, comment modifier l'affichage par défaut.

Continuons sur l'exemple précédent, en essayant la commande suivante :

```
PS > Get-ChildItem c:\ | Format-Table  
  
Répertoire : C:\
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	14/07/2009 04:37		PerfLogs
d-r--	05/09/2009 00:37		Program Files
d-r--	01/09/2009 22:55		Users
d----	06/09/2009 14:42		Windows
-a---	10/06/2009 23:42	24	autoexec.bat
-a---	10/06/2009 23:42	10	config.sys

Oh surprise ! Nous remarquons que **Format-Table** n'a pas d'effet sur notre commande **Get-ChildItem** ; le résultat est identique sans **Format-Table**.

Ceci est normal car, par défaut, le résultat de **Get-ChildItem** se fait toujours dans ce format.

Vous venez de découvrir qu'avec PowerShell, chaque type d'objet possède une liste de propriétés affichées par défaut.

Retenez donc bien cela : « ce n'est pas parce que, par défaut, certaines propriétés ne s'affichent pas dans la console que l'objet ne les possède pas ».

Voici les paramètres les plus couramment utilisés avec **Format-Table** :

Paramètre	Description
Property	Propriété ou liste de propriétés à afficher.
Autosize	Ajuste la taille des colonnes au nombre de caractères à afficher.
HideTableHeaders	Masque les en-têtes de colonnes.
GroupBy	Regroupe l'affichage selon une propriété ou une valeur commune.

Voici quelques exemples pour illustrer ces paramètres :

Exemple :

Lister les propriétés personnalisées dans un tableau.

```
PS > Get-ChildItem c:\ | Format-Table -Property mode,name,length,
isreadonly,creationTime,lastAccesstime,attributes
```

Mode	Name	length	isreadonly	CreationTi me	LastAcces sTime	Attribute s
----	----	-----	-----	-----	-----	-----
d----	PerfLogs			14/07/2...	14/07/...	Directory
d-r--	Program...			14/07/2...	05/09/...	...ectory
d-r--	Users			14/07/2...	01/09/...	...ectory
d----	Windows			14/07/2...	06/09/...	Directory
-a---	autoexe...	24	False	14/07/2...	14/07/...	Archive
-a---	config.sys	10	False	14/07/2...	14/07/...	Archive

Dans cet exemple, vous remarquez qu'il y a des points de suspension un peu partout « ... ». Cela signifie que PowerShell a tronqué des valeurs car il n'avait pas assez de place pour les afficher. Par défaut, la console adapte l'affichage à la taille de la fenêtre, et pour ce faire elle occupe tout l'espace (à l'horizontal) qui lui est alloué et calcule la taille des colonnes en fonction de leur nombre. Dans ce cas précis, toutes les colonnes ont la même taille ; c'est la raison pour laquelle on peut voir un grand nombre d'espace entre certaines colonnes alors que d'autres n'ont pas assez de place pour afficher leurs données (si le calcul ne tombe pas juste, les premières colonnes (à gauche) peuvent avoir un ou deux caractères de plus que les autres).

Pour tenter de régler ce « problème », le paramètre **-Autosize** a été créé.

a. Taille automatique d'un tableau

- Essayez maintenant la même ligne de commandes que précédemment mais en ajoutant « **-autosize** » à la fin :

Exemple :

Lister les propriétés personnalisées dans un tableau de taille automatique.

```
PS > Get-ChildItem c:\ | Format-Table -Property mode,name,length,
isreadonly,creationTime,lastAccesstime,attributes -AutoSize
```

AVERTISSEMENT : la colonne « Attributes » ne tient pas à l'écran et a été supprimée.

Mode	Name	length	isreadonly	CreationTime	LastAccessTime
d----	PerfLogs			14/07/2009 04:37:05	14/07/2009 04...
d-r--	Program Files			14/07/2009 04:37:05	05/09/2009 00...
d-r--	Users			14/07/2009 04:37:05	01/09/2009 22...
d----	Windows			14/07/2009 04:37:05	06/09/2009 14...
-a---	autoexec.bat	24	False	14/07/2009 04:04:04	14/07/2009 04...
-a---	config.sys	10	False	14/07/2009 04:04:04	14/07/2009 04...

Victoire ! Nos informations se sont bien affichées et aucune donnée n'a été tronquée ou presque. Le rendu paraît à présent plus équilibré mais notez que pour en arriver là, la colonne *Attributes* a dû être supprimée. PowerShell a adapté la taille de chaque colonne à la taille maximale de son contenu.

Lorsque le paramètre *autosize* est spécifié, PowerShell donne la priorité à l'affichage des colonnes de gauche. Il considère que l'importance des colonnes est donnée par l'ordre dans lequel les propriétés sont spécifiées sur la ligne de commandes.



Powershell nous indique par un message d'avertissement quand il ne peut pas, par manque de place, afficher une colonne.

b. Regroupement de propriétés

Le paramètre **-GroupBy** permet de regrouper les informations à afficher par une propriété ou une valeur commune.

Exemple :

Regroupement d'informations autour d'une propriété commune.

```
PS > Get-ChildItem | Format-Table -Property mode,name,length,
isreadonly,creationTime,lastAccesstime -AutoSize -GroupBy isReadOnly
```

Mode	Name	length	isreadonly	CreationTime	LastAccessTime
d----	PerfLogs			14/07/2009 04:37:05	14/07/2009 04...
d-r--	Program Files			14/07/2009 04:37:05	05/09/2009 00...
d-r--	Users			14/07/2009 04:37:05	01/09/2009 22...
d----	Windows			14/07/2009 04:37:05	06/09/2009 14...

IsReadOnly: False

Mode	Name	length	isreadonly	CreationTime	LastAccessTime
-a---	autoexec.bat	24	False	14/07/2009 04:04:04	14/07/2009 04:...
-a---	config.sys	10	False	14/07/2009 04:04:04	14/07/2009 04:...

3. Format-Wide

Cette commande permet d'afficher la propriété par défaut d'un type de donnée sur une ou plusieurs colonnes. Nous insistons volontairement sur la propriété car **Format-Wide** ne peut en afficher qu'une seule à la fois.

Exemple :

Lister les fichiers sur deux colonnes avec **Format-Wide**.

```
PS > Get-ChildItem C:\Windows | Format-Wide

Répertoire : C:\Windows

[addins]                [AppCompat]
[AppPatch]              [assembly]
[Boot]                  [Branding]
[CSC]                   [Cursors]
[debug]                 [diagnostics]
[DigitalLocker]         [Downloaded Program Files]
...
[Temp]                  [tracing]
[twain_32]              [Vss]
[Web]                   [winsxs]
ativpsrm.bin            bfsvc.exe
bootstat.dat            DtcInstall.log
explorer.exe            fveupdate.exe
HelpPane.exe            hh.exe
mib.bin                 msdfmap.ini
notepad.exe             PFR0.log
regedit.exe             setupact.log
setuperr.log            Starter.xml
system.ini              TSSysprep.log
twain.dll               twain_32.dll
twunk_16.exe            twunk_32.exe
Ultimate.xml            win.ini
WindowsUpdate.log       winhelp.exe
winhlp32.exe            WMSysPr9.prx
write.exe               _default.pif
```

Étant donné que la propriété par défaut d'un fichier ou d'un dossier est le nom, celui-ci s'affiche ici sur deux colonnes. Comme pour **Format-Table**, PowerShell dimensionne automatiquement les colonnes. L'affichage sur deux colonnes est l'affichage par défaut de **Format-Wide**, mais celui-ci peut être changé.

Voici les paramètres les plus couramment utilisés avec **Format-Wide** :

Paramètre	Description
Property	Propriété à afficher. Une seule valeur est autorisée.
Autosize	Ajuste la taille des colonnes au nombre de caractères à afficher.
column	Force le résultat à s'afficher sur un nombre de colonnes passé en paramètre.

Exemple :

Choix d'une colonne autre que celle par défaut.

```
PS > Get-ChildItem C:\ | Format-Wide -Property fullname

C:\PerfLogs                C:\Program Files
C:\Users                   C:\Windows
C:\autoexec.bat            C:\config.sys
```

Cet exemple n'a que peu d'intérêt pour la commande **Get-ChildItem**. En revanche il en pourrait en avoir davantage pour **Get-Service** en affichant par exemple le nom détaillé de chaque service au lieu du nom court.

Exemple :

```
PS > Get-Service | Format-Wide -property displayName
```

Expérience d'application	Service de la passerelle de la co...
Identité de l'application	Informations d'application
Apple Mobile Device	Gestion d'applications
Générateur de points de terminaiso...	Audio Windows
Programme d'installation ActiveX (...)	Service de chiffrement de lecteur...
Moteur de filtrage de base	Service de transfert intelligent ...
Service Bonjour	Explorateur d'ordinateurs
Service de prise en charge Bluetooth	Symantec Event Manager...

Exemple :

Liste de fichiers au format large avec le paramètre **-Autosize**.

```
PS > Get-ChildItem C:\Windows | Format-Wide -Autosize
```

Répertoire : C:\Windows

[addins]	[AppCompat]
[AppPatch]	[assembly]
[Boot]	[Branding]
[CSC]	[Cursors]
[debug]	[diagnostics]
[DigitalLocker]	[Downloaded Program Files]
[ehome]	[en-US]
[Fonts]	[fr-FR]
...	
[Temp]	[tracing]
[twain_32]	[Vss]
[Web]	[winsxs]
ativpsrm.bin	bfsvc.exe
bootstat.dat	DtcInstall.log
explorer.exe	fveupdate.exe
HelpPane.exe	hh.exe
mib.bin	msdfmap.ini
notepad.exe	PFR0.log
regedit.exe	setupact.log
setuperr.log	Starter.xml
system.ini	TSSysprep.log
twain.dll	twain_32.dll
twunk_16.exe	twunk_32.exe
Ultimate.xml	win.ini
WindowsUpdate.log	winhelp.exe
winhlp32.exe	WMSysPr9.prx
write.exe	_default.pif

Une fois encore PowerShell se charge de la mise en page, et il faut dire qu'avec le paramètre **-Autosize** celle-ci est la plupart du temps bien réussie. On se demanderait presque pourquoi ce paramètre n'est pas activé par défaut tellement il est pratique !

Ceci étant la raison est la suivante : avec **-Autosize** il faut que la commandelette de formatage attende d'avoir tous les éléments avant de pouvoir les afficher avec des tailles de colonnes adéquates, alors que dans le cas où **-Autosize** n'est pas précisé, elle affiche les objets au fur et à mesure où elle les reçoit. Cela peut vous sembler être un détail, mais par exemple si l'on prend le cas d'un script qui dure deux heures et qui affiche des informations au fil de l'eau ; et bien si l'on spécifie le paramètre **-Autosize** pour le formatage du résultat, nous n'obtiendrons aucune information avant la fin d'exécution du script.

Exemple :

Affichage du résultat sur quatre colonnes.

```
PS > Get-ChildItem C:\Windows | Format-Wide -Column 4
```

Répertoire : C:\Windows

[addins]	[AppCompat]	[AppPatch]	[assembly]
[Boot]	[Branding]	[CSC]	[Cursors]
[debug]	[diagnostics]	[DigitalLocker]	[Downloaded Pr...
[ehome]	[en-US]	[Fonts]	[fr-FR]
[Globalization]	[Help]	[IME]	[inf]
[L2Schemas]	[LiveKernelRepo...]	[Logs]	[Media]
[Microsoft.NET]	[ModemLogs]	[Offline Web Pa...]	[Panther]
[PCHEALTH]	[Performance]	[PLA]	[PolicyDefinit...
[Prefetch]	[Registration]	[RemotePackages]	[rescache]
[Resources]	[SchCache]	[schemas]	[security]
[ServiceProfiles]	[servicing]	[Setup]	[ShellNew]
[SoftwareDistri...]	[Speech]	[system]	[System32]
[TAPI]	[Tasks]	[Temp]	[tracing]
[twain_32]	[Vss]	[Web]	[winsxs]
ativpsrm.bin	bfsvc.exe	bootstat.dat	DtcInstall.log
explorer.exe	fveupdate.exe	HelpPane.exe	hh.exe
mib.bin	msdfmap.ini	notepad.exe	PFR0.log
regedit.exe	setupact.log	setuperr.log	Starter.xml
system.ini	TSSysprep.log	twain.dll	twain_32.dll
twunk_16.exe	twunk_32.exe	Ultimate.xml	win.ini
WindowsUpdate.log	winhelp.exe	winhlp32.exe	WMSysPr9.prx
write.exe	_default.pif		

Comme vous pouvez le constater, **-Column** permet de forcer l'affichage sur un nombre de colonnes voulu. Dans l'exemple précédent, **-Autosize** nous avait affiché le résultat sur deux colonnes car au-delà, certaines informations auraient été tronquées (apparition de points de suspension dans le nom).

Règles à connaître

1. Utilisation des guillemets dans les chaînes de caractères

Généralement, dans tous les langages informatiques quels qu'ils soient, on utilise les guillemets doubles " " pour délimiter des chaînes de caractères. Bien que PowerShell ne déroge pas à cette règle, il y a quelques petites subtilités qu'il est bon de connaître.

Il existe dans PowerShell deux façons de créer une chaîne :

- En l'encadrant avec des guillemets doubles " "
- En l'encadrant avec des guillemets simples ' '

À première vue, il n'y a pas de différence notable entre ces deux écritures, par exemple :

```
PS > Write-Host "Bonjour !"
Bonjour !
```

```
PS > Write-Host 'Bonjour !'
Bonjour !
```

La nuance se fait sentir dès lors que l'on travaille avec des variables ; en effet les doubles guillemets ont pour effet de remplacer une variable par son contenu. Ce phénomène est ce que l'on appelle « la substitution des variables ». Les guillemets simples quant à eux ignorent les variables et conservent fidèlement la chaîne qu'ils contiennent.

Par exemple :

```
PS > $a = 'Bonjour'
PS > $b = 'monde !'

PS > Write-Host "$a $b"
Bonjour monde !

PS > Write-Host '$a $b'
$a $b
```



Bien que nous puissions aussi utiliser la commande **Write-Host** sans guillemets, nous vous conseillons de choisir systématiquement l'une des deux formes de guillemets ; et ce pour davantage de lisibilité.

Comment faire à présent pour créer une chaîne qui contienne un caractère dollar ainsi que le contenu d'une ou plusieurs variables ? Soit par exemple, la chaîne « \$c = Bonjour monde ! »

Essayons cela :

```
PS > Write-Host '$c = $a $b'
$c = $a $b

PS > Write-Host "$c = $a $b"
= Bonjour monde !
```

Aucune des deux écritures ne parvient à afficher correctement le résultat souhaité. Cependant grâce aux caractères d'échappement, nous allons pouvoir arriver à nos fins.



Bien qu'il soit tentant de continuer à utiliser les bonnes vieilles habitudes avec l'utilisation systématique des doubles guillemets ce n'est pas une bonne pratique !



Nous vous encourageons à utiliser les simples guillemets sauf lorsque vous savez qu'il y a une substitution de variables à effectuer ; il vous sera donc facile à ce moment là de changer de styles de guillemets. Il est préférable de travailler dans ce sens là plutôt que l'inverse, car la substitution de variables peut parfois provoquer des effets inattendus difficiles à déboguer. Cela est particulièrement vrai avec les expressions régulières et

notamment avec les opérateurs **-match** et **-replace**.

2. Caractères d'échappement

PowerShell met à notre disposition un caractère assez particulier : le backtick « ` » ou guillemet inverse en français. Celui-ci correspond au caractère obtenu en pressant la séquence de touches [AltGr]+7. Le backtick va nous permettre de transformer un caractère spécial en un caractère normal, par exemple placé devant un caractère « \$ » le backtick empêchera la substitution d'une variable.

Sachant cela, nous pourrions résoudre ainsi le problème de tout à l'heure :

```
PS > Write-Host "`$c = $a $b"
$c = Bonjour monde !
```

Ceux d'entre vous qui ont déjà pratiqué le langage C auront remarqué que le backtick est l'équivalent du caractère d'échappement backslash « \ » ou anti-slash en français.

Si nous devons donner une définition d'un caractère d'échappement, nous dirions simplement qu'un caractère d'échappement est un caractère qui a une signification particulière pour un interpréteur de commandes.

Voici la liste des caractères d'échappement de PowerShell et leurs effets :

Caractère d'échappement	Transformation résultante
`n	Saut de ligne
`f	Saut de page
`r	Retour chariot
`a	Bip sonore
`b	Retour arrière
`t	Tabulation horizontale
`v	Tabulation verticale
`0	Null
`'	Guillemet simple
`"	Guillemet double
` `	Backtick simple

Exemples :

```
PS > Write-Host "Phrase trop longue `nà couper en deux"
Phrase trop longue
à couper en deux
```

```
PS > Write-Host "Powershell c'est super !"
Powershell c'est super !
```

```
PS > Write-Host "J'é mets des `bips`" sonores `a`a"
J'é mets des "bips" sonores <bip><bip>
```

Le backtick lorsqu'il est utilisé en fin d'une ligne de commandes indique à PowerShell que celle-ci continue sur la ligne suivante. Cela est pratique pour la présentation d'une grande suite de commandes. Vous remarquerez tout au long

de cet ouvrage, que nous nous servons abondamment de cette technique afin de donner davantage de clarté à nos exemples.



L'équipe de développement de PowerShell n'a pas pu reprendre l'antislash comme caractère d'échappement car celui-ci est largement utilisé dans le monde Windows pour délimiter des chemins de répertoires.

3. Here-String

Une Here-String est une chaîne qui commence avec le séparateur arobase suivi du guillemet simple « @' » et qui se termine par un guillemet simple suivi de l'arobase « '@ » (le dernier séparateur doit absolument être précédé d'un retour chariot). Tous les caractères entre les délimiteurs '@' et '@' sont considérés comme du texte pur.

Les Here-Strings sont très utiles pour stocker des chaînes de caractères de plusieurs lignes. Elles évitent la pénible tâche de concaténer des variables. Pour bien comprendre leur fonctionnement, un petit exemple s'impose !

Exemple 1 :

```
PS > $chaine1 = @'
>> Lundi : début de semaine "difficile"
>> Mercredi : jour des enfants
>> Vendredi : vive le début du week-end !
>> '@

PS > $chaine1
Lundi : début de semaine "difficile"
Mercredi : jour des enfants
Vendredi : vive le début du week-end !
```

Comme pour une chaîne de caractères entre guillemets simples, le contenu d'une Here-String « simple quote » n'est pas interprété contrairement aux Here-String « doubles quotes ».

Exemple 2 :

```
PS > $s1 = 'Lundi'
PS > $s2 = 'Mercredi'
PS > $s3 = 'enfants'

PS > $chaine2 = @"
>> $s1 : début de semaine "difficile"
>> $s2 : jour des $s3
>> Vendredi : vive le début du week-end !
>> "@

PS > $chaine2
Lundi : début de semaine "difficile"
Mercredi : jour des enfants
Vendredi : vive le début du week-end !
```

Nous le verrons par la suite, mais sachez que les Here-Strings sont fabuleuses pour la manipulation des documents HTML ou XML.

4. Commentaires et blocs de commentaires

Lors de l'écriture de scripts il peut être utile de pouvoir insérer des commentaires tels que la description du script, la date du jour, ou autres explications techniques.

Pour ce faire, PowerShell utilise le caractère **dièse** « # » pour marquer le début d'un commentaire.

Exemple 1 :

```
# +-----+
```

```
# Entête du script
# +-----+
```

On peut aussi mettre des commentaires après des commandes ou des traitements.

Exemple 2 :

```
if ($i -eq 1)    # $i contient le choix de l'utilisateur
{
}
}
```

La version 2 de PowerShell offre la possibilité d'insérer des blocs de commentaires. On ouvre un bloc de commentaires avec « <# » et on ferme ce dernier avec « #> », tel que dans l'exemple ci-dessous :

```
<#
Début du bloc de commentaires
    Bla bla bla...
    Bla bla bla...
    Bla bla bla...
Fin du bloc de commentaires
#>
```

Les blocs de commentaires facilitent la mise en commentaire d'une partie d'un script, plutôt que de préfixer chaque ligne à commenter par le caractère dièse.

5. Substitution des variables

Ceci est un point très important qu'il vous est indispensable de connaître.

Lorsque vous désirez afficher la valeur d'une propriété d'un objet il faut toujours utiliser la syntaxe suivante : **\$(Sujet.propriété)**

En effet, si vous le ne faites pas, voici que ce vous pourriez obtenir :

```
PS > $a = Get-ChildItem c:\config.sys
PS > Write-Host "Taille du fichier : $a.Length octets"
Taille du fichier : C:\config.sys.Length octets
```

Vous l'aurez remarqué, PowerShell substitue la variable \$a par son contenu et traite « .Length » comme une chaîne de caractères. La syntaxe correcte est donc la suivante :

```
PS > Write-Host "Taille du fichier : $($a.Length) octets"
Taille du fichier : 10 octets
```

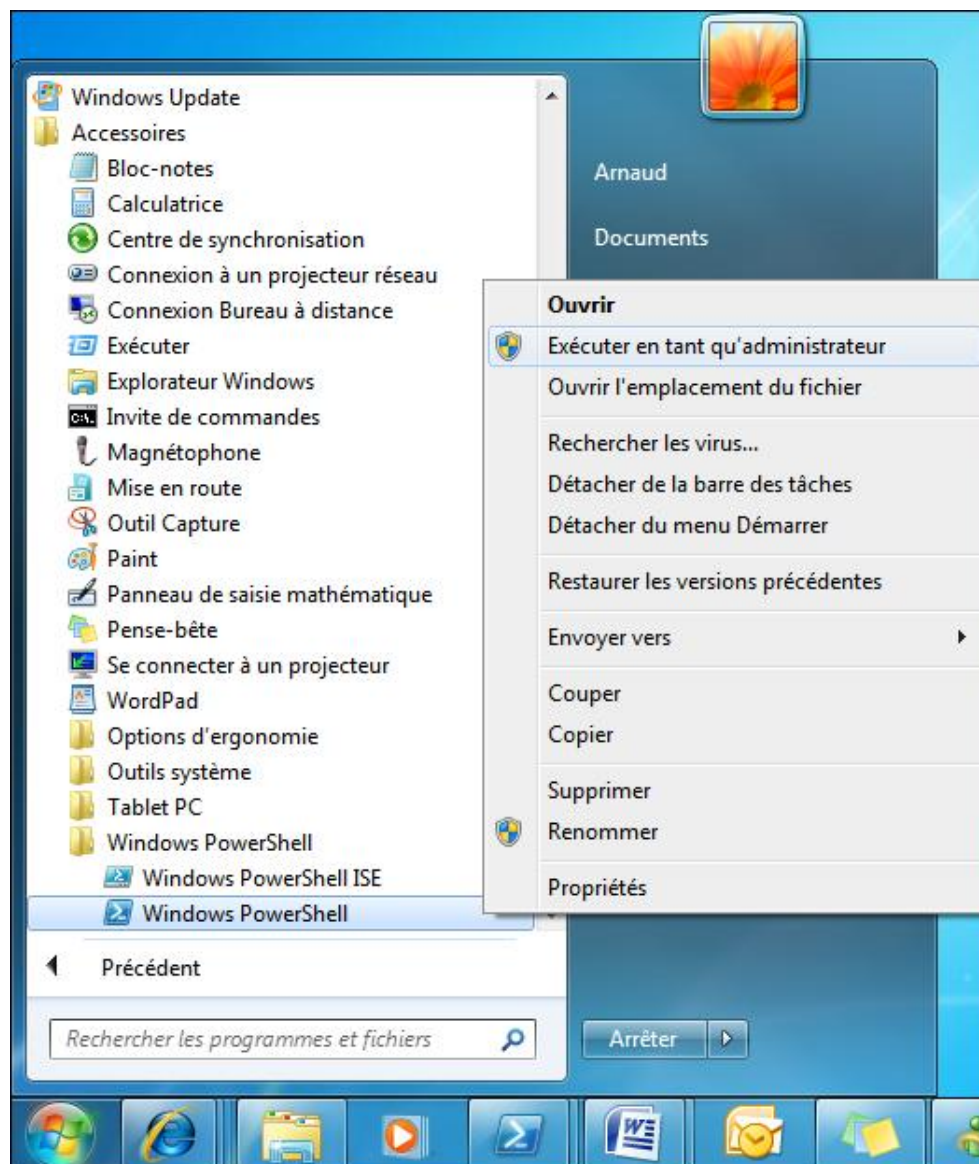


Faites également attention aux guillemets que vous utilisez lorsque vous construisez des chaînes de caractères, car rappelez-vous : les guillemets simples ne font pas de substitution de variables.

6. Démarrage de la console

Lorsque vous démarrez la console PowerShell par le biais du menu **Démarrer** ou par le biais d'un raccourci que vous auriez pu créer au bureau, il faut savoir que cette dernière s'exécute avec des droits de simple utilisateur et donc limités ; et ce, même si vous avez ouvert votre session avec un compte administrateur. Ne soyez donc pas surpris si vous vous voyez l'accès refusé à certains répertoires ou à certaines clés de registres.

Pour ouvrir une console classique ou graphique (ISE) avec le privilège Administrateur, vous devez systématiquement cliquer droit sur l'icône PowerShell (ou PowerShell ISE) et choisir **Exécuter en tant qu'administrateur** comme ci-après.



Menu Démarrer, Exécuter PowerShell en tant qu'administrateur

Vous ferez la différence entre les consoles PowerShell ouvertes en tant qu'administrateur et celles qui ne le sont pas en observant l'intitulé des fenêtres en haut à gauche de ces dernières, comme ci-dessous :



Intitulé des fenêtres

Les variables et constantes

1. Création et affectation

La création d'une variable en PowerShell est vraiment chose facile. À l'instar des langages objets, PowerShell ne dispose pas d'un langage typé, c'est-à-dire que les variables n'ont pas besoin d'être définies avant d'être utilisées. Ainsi, il suffit d'affecter via l'opérateur " = ", une valeur à votre variable et PowerShell se charge du reste, à savoir la créer et déterminer son type. La syntaxe utilisée est la suivante :

```
$variable = valeur d'un type quelconque
```

À l'inverse pour lire une variable, il suffit de taper tout simplement le nom de la variable dans la console.

- En tapant `$var_1 = 12` dans la console PowerShell nous créons une variable du nom de « var_1 », de type « int » (entier) et nous lui affectons la valeur 12.
- En tapant `$var_2 = 'A'` nous réalisons la même opération à l'exception que cette fois-ci votre variable est du type « string » (chaîne) même si elle ne contient qu'un caractère et que la valeur associée est la lettre A.



Vous pouvez retrouver le type de votre variable en lui appliquant la méthode `GetType`.

Exemple :

```
PS > $var_1 = 12
PS > $var_1.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     Int32                                     System.ValueType
```



Si vous utilisez des noms de variable avec des caractères spéciaux (& @ % - £ \$. , etc.) il est indispensable d'utiliser les caractères « { » et « } ».

Exemple :

```
${www.powershell-scripting.com} = 1
```

Ceci affectera la valeur 1 à la variable entre accolades.

a. Conversion de variables

Cependant, il peut être intéressant pour diverses raisons de rester maître du typage des variables. Alors que les inconditionnels se rassurent il existe une alternative au typage automatique. Pour ce faire, il nous faut définir le type souhaité entre crochets avant la création de la variable, comme par exemple :

```
PS > [int]$var=12
```

En écrivant la ligne précédente vous êtes sûr que la variable `$var` est du type entier. Mais il n'y a aucune différence entre `$var = 12` et `[int]$var = 12` nous direz vous ! Certes, car pour l'instant l'intérêt est minime, mais lorsque vous serez de grands « powershelleurs » et que vos scripts commenceront à prendre de l'ampleur, le fait de déclarer vos variables avec un type associé rendra votre script beaucoup plus compréhensible pour les autres mais permettra surtout d'éviter qu'une valeur d'un type différent ne lui soit affecté.

Par exemple :

```
PS > [int]$nombre = read-host 'Entrez un nombre '
Entrez un nombre: cent
```

```
Impossible de convertir la valeur « cent » en type « System.Int32 ».  
Erreur : « Le format de la chaîne d'entrée est incorrect. »
```

Dans l'exemple ci-dessus, le texte saisi (« cent ») n'a pas été reconnu comme un nombre entier et est donc rejeté par l'interpréteur de commandes. Si nous n'avions pas précisé le type [int] devant la variable \$nombre, le texte aurait été accepté et son traitement ultérieur aurait pu poser problème.

Si maintenant nous essayons d'attribuer une valeur entière dans un type « char » :

```
PS > [char]$var=65
```

Que va-t-il se passer ? PowerShell va tout simplement convertir la valeur entière en un caractère, et pas n'importe lequel, mais le caractère dont le code ASCII correspond à la valeur entière. Dans notre exemple \$var contiendra « A » car le caractère « A » correspond à 65 en code ASCII.

Et enfin, essayons de réaliser l'opération inverse, c'est-à-dire passer du type « string » au type « int ». Ceci n'est malheureusement pas possible directement :

```
PS > [int]$var = 'A'  
Impossible de convertir la valeur « A » en type « System.Int32 ».  
Erreur : « Le format de la chaîne d'entrée est incorrect. »  
Au niveau de ligne : 1 Caractère : 8+ [int]$a <= 'A'
```

Cependant il est possible de convertir une variable de type « char » en type « int » :

```
PS > [int][char]$var = 'A'  
PS > $var  
65
```



Le fait de pouvoir convertir uniquement des variables du type « char » vient du fait que l'on ne peut faire correspondre qu'un caractère à un code ASCII, et non toute une chaîne.

Regardons à présent ce qu'il se passe si nous affectons une valeur décimale de type « double » à une variable de type « int » :

```
PS> $var1=10.5  
PS> $var1  
10,5  
PS> $var2=[int]$var1  
PS> $var2  
10
```

En toute logique, la variable \$var2 est arrondie à la partie entière la plus proche, puisqu'une variable de type entière n'accepte que les entiers dans une plage comprise entre -2 147 483 648 et 2 147 483 647 inclus.

Mais si nous tentons de convertir une valeur beaucoup plus grande que la plage couverte par les entiers, voici ce qu'il va se passer :

```
PS> $var1=1e27  
PS >1E+27  
PS > $var2=[int]$var1  
Impossible de convertir la valeur « 1E+27 » en type « System.Int32 ».  
Erreur : « La valeur était trop grande ou trop petite pour un Int32.»
```

PowerShell va spécifier une erreur pour nous dire qu'il n'a pu réussir à convertir une valeur aussi longue dans une variable de type entière.

Bien entendu l'affectation des variables ne se limite pas au système décimal, nous pouvons également convertir des valeurs décimales en hexadécimales et les stocker dans une variable. Pour réaliser ce type d'opération, PowerShell s'appuie sur les formats d'affichage des chaînes de caractères (opérateur -f) du Framework .NET. Comme nous n'avons ni abordé les chaînes de caractères, ni les méthodes du Framework .NET, voici simplement les commandes permettant la conversion.

Exemple :

Conversion d'un nombre décimal en hexadécimal :

```
PS > $dec = 1234
```

```
PS > $hex = "{0:X}" -f $dec
PS > $hex
4D2
```

Attention car l'utilisation d'un format d'affichage de chaînes change le type de la variable **\$hex** en type « chaîne de caractères » (string). Pour le vérifier : tapez `$hex.GetType()`

Toujours avec le même principe, nous pouvons convertir tout nombre décimal de notre choix en nombre dans la base souhaitée. Pour cela il suffit d'utiliser la commande suivante : `[System.Convert]::ToString(<valeur_1>,<valeur_2>)` où `valeur_1` correspond au nombre (en base 10) à convertir et `valeur_2` la nouvelle base du nombre.

Exemple :

Conversion d'un nombre décimal en base 8.

```
PS > $Nb = [System.Convert]::ToString(1234,8)
PS > $Nb
2322
```

Exemple :

Conversion d'un nombre décimal en base 2.

```
PS > $Nb = [System.Convert]::ToString(1234,2)
PS > $Nb
10011010010
```

2. Les variables prédéfinies

PowerShell dispose d'un certain nombre de variables automatiques qu'il est bon de connaître. En voici la liste non exhaustive :



Les variables sur lesquelles figure une étoile ne sont disponibles que dans la version 2 de PowerShell.

Variable	Description
\$\$	Variable contenant le dernier jeton de la dernière ligne reçue par l'environnement (c'est-à-dire le dernier mot de la dernière commande tapée dans la console).
\$?	Variable contenant <i>true</i> si la dernière opération a réussi ou <i>false</i> dans le cas contraire.
\$^	Variable contenant le premier jeton de la dernière ligne reçue par l'environnement (c'est-à-dire le premier mot de la dernière commande tapée dans la console).
\$_	Variable contenant l'objet courant transmis par le pipe « », le pipe sera abordé plus tard dans ce chapitre.
\$Args	Variable contenant un tableau des arguments passés à une fonction ou à un script.
\$ConfirmPreference	Variable permettant de déterminer quelles commandelettes demandent automatiquement la confirmation de l'utilisateur avant exécution. Lorsque la valeur de <code>\$ConfirmPreference</code> (High , Medium , Low , None [Élevée, Moyenne, Faible, Aucune]) est supérieure ou égale au risque de l'action d'applet de commande (High , Medium , Low , None), Windows PowerShell demande automatiquement la confirmation de l'utilisateur avant d'exécuter l'action.

\$ConsoleFileName	Variable qui contient le chemin d'accès du fichier console (.psc1) qui a été utilisé en dernier dans la session.
\$DebugPreference	Variable contenant une valeur spécifique correspondant à une action préférentielle à établir. Utilisée avec la commande Write-Debug (cf. chapitre Gestion des erreurs et débogage - Le débogage - Affichage de messages en mode debug).
\$Error	Variable sous forme de tableau contenant l'enregistrement des erreurs affichées lors de la session (cf. chapitre Gestion des erreurs et débogage - Les erreurs non-critiques - Le type ErrorRecord).
\$ErrorActionPreference	Variable contenant une valeur spécifique correspondant à une action préférentielle à établir en cas d'erreur. utilisée avec la commande Write-Error (cf. chapitre Gestion des erreurs et débogage).
\$ErrorView	Variable déterminant le format d'affichage des messages d'erreur dans Windows PowerShell. (cf. chapitre Gestion des erreurs et débogage).
\$ExecutionContext	Variable contenant un objet EngineIntrinsics représentant le contexte d'exécution de l'hôte Windows PowerShell.
\$False	Variable contenant la valeur <i>false</i> . Cette variable est une constante, et par conséquent ne peut être modifiée.
\$Foreach	Variable qui fait référence à l'énumérateur d'une boucle Foreach.
\$FormatEnumerationLimit	Variable qui détermine le nombre d'éléments énumérés inclus dans un affichage.
\$Home	Variable contenant le chemin (<i>path</i>) du répertoire de base de l'utilisateur.
\$Host	Variable contenant des informations sur l'hôte.
\$Input	Variable énumérant les objets transmis par le pipeline.
\$LastExitCode	Variable contenant le code de sortie de la dernière exécution d'un fichier exécutable Win32.
\$MaximumAliasCount	Variable contenant le nombre maximal d'alias possibles dans la session.
\$MaximumDriveCount	Variable contenant le nombre maximal de lecteurs possibles dans la session (ceux fournis par le système ne sont pas pris en compte).
\$MaximumErrorCount	Variable contenant le nombre maximal d'erreurs enregistrées dans l'historique d'erreur pour la session.
\$MaximumFunctionCount	Variable contenant le nombre maximal de fonctions possibles dans la session.
\$MaximumHistoryCount	Variable contenant le nombre maximal de commandes qui peuvent être enregistrées dans l'historique.
\$MaximumVariableCount	Variable contenant le nombre maximal de variables possibles dans la session.
\$MyInvocation	Variable qui contient un objet relatif aux informations sur la commande en cours.
\$NestedPromptLevel	Variable qui indique le niveau d'invite actuel. La valeur 0 indique le niveau d'invite d'origine. La valeur est incrémentée lorsque vous accédez à un niveau imbriqué et décrétementée lorsque vous le quittez.

\$Null	Variable vide.
\$OFS	Variable contenant le séparateur de champ lors de la conversion d'un tableau en chaîne.
\$OutputEncoding	Variable contenant la méthode d'encodage de caractères utilisée par Windows PowerShell lorsqu'il envoie du texte à d'autres applications.
\$PID	Variable contenant le numéro ID du processus PowerShell.
\$Profile	Variable contenant le chemin (<i>path</i>) du profil Windows PowerShell.
*\$ProgressReference	Variable qui détermine la façon dont Windows PowerShell répond aux mises à jour de progression générées par un script, une commandelette ou un fournisseur.
*\$PSBoundParameters	Variable contenant un dictionnaire des paramètres et des valeurs actuelles en cours.
*\$PSCulture	Variable qui contient le nom de la culture actuellement utilisée dans le système d'exploitation (fr-FR pour la langue française).
*\$PSEmailServer	Variable contenant le serveur de messagerie à utiliser par défaut avec la commandelette Send-MailMessage.
\$PsHome	Variable contenant le chemin (<i>path</i>) où PowerShell est installé.
*\$PSSessionApplicationName	Variable contenant le nom de l'application utilisée pour l'utilisation des commandes à distance. L'application système par défaut est WSMAN.
*\$PSSessionConfigurationName	Variable contenant l'URL de la configuration de session utilisée par défaut.
*\$PSSessionOption	Variable contenant les valeurs par défaut lors d'une session à distance.
*\$PSUICulture	Variable qui contient le nom de la culture d'interface utilisateur (IU) qui est actuellement employée.
*\$PSVersionTable	Variable qui contient un tableau en lecture seule qui affiche les détails relatifs à la version de Windows PowerShell.
\$PWD	Variable indiquant le chemin complet du répertoire actif.
\$ReportErrorShowExceptionClass	Variable qui affiche les noms de classes des exceptions affichées.
\$ReportErrorShowInnerException	Variable qui affiche (lorsque sa valeur est <i>true</i>) la chaîne des exceptions internes.
\$ReportErrorShowSource	Variable qui affiche (lorsque sa valeur est <i>true</i>) les assembly names (cf. chapitre .NET - Utiliser des objets .NET avec PowerShell - Les Assemblies) des exceptions affichées.
\$ReportErrorShowStackTrace	Variable qui émet (lorsque sa valeur est <i>true</i>) les arborescences des appels de procédure d'exceptions.
\$ShellID	Variable indiquant l'identificateur du Shell.
\$ShouldProcessPreference	Spécifie l'action à entreprendre lorsque ShouldProcess est utilisé dans une commandelette.
\$ShouldProcessReturnPreference	Variable contenant la valeur retournée par ShouldPolicy.
\$StackTrace	Variable contenant les informations d'arborescence des appels de

	procédure détaillées relatives à la dernière erreur.
\$True	Variable contenant la valeur <i>true</i> .
\$VerbosePreference	Variable contenant une valeur spécifique correspondant à une action préférentielle à établir. Utilisée avec la commande Write-Verbose (cf. chapitre Gestion des erreurs et débogage - Le débogage - Affichage de messages en mode verbose).
\$WarningPreference	Variable contenant une valeur spécifique correspondant à une action préférentielle à établir. Utilisée avec la commande Write-Warning (cf. chapitre Gestion des erreurs et débogage - Le débogage - Affichage de messages en mode warning).
\$WhatIfPreference	Variable qui détermine si le paramètre WhatIf est activé automatiquement pour chaque commande qui le prend en charge.

3. Les différents opérateurs

Il existe plusieurs types d'opérateurs, qu'ils soient de type arithmétiques, binaires, logiques ou autres, ils vous permettront d'agir sur les variables. Gardez bien à l'esprit que connaître et maîtriser les différentes opérations est essentiel pour l'élaboration d'un bon script.

a. Les opérateurs arithmétiques

En ce qui concerne les opérations arithmétiques, il n'y a rien de compliqué. PowerShell traite les expressions de gauche à droite en respectant les règles des propriétés mathématiques ainsi que les parenthèses.

Exemple :

```
PS > 2+4*3
14
PS > (2+4)*3
18
```

La liste des opérateurs arithmétiques disponibles vous est donnée ci-dessous :

Signe	Signification
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo

Les quatre premiers opérateurs doivent logiquement vous sembler familiers, quand au dernier, l'opérateur modulo, il permet de renvoyer le reste d'une division entière de a par b.

Par exemple : en tapant 5%3 dans la console, nous obtiendrons 2. Tout simplement parce qu'il y a 1 fois 3 dans 5 et que le reste de la division vaut 2.

Vous constaterez que les opérations arithmétiques s'appliquent également aux variables. Ainsi, \$var_1 + \$var_2 vous donnera la somme des deux variables si elles contiennent des valeurs numériques.

Exemple de l'opérateur "+" sur deux entiers :

```
PS > $int1 = 10
```

```
PS > $int2 = 13
PS > $int2 + $int1
23
```

L'opérateur addition s'emploie également avec des chaînes (variables de type string). Dans ce cas là, l'opérateur sert à concaténer les deux chaînes :

```
PS > $chaine1 = 'A'
PS > $chaine2 = 'B'
PS > $chaine1 + $chaine2
AB
```

Toujours avec les chaînes de caractères, sachez qu'il est possible de répéter le contenu d'une chaîne grâce à l'opérateur multiplication :

```
PS > $chaine1 = 10 * 'A'
PS > $chaine1
AAAAAAAAAA
```



Retrouvez d'autres opérateurs mathématiques comme le calcul d'un sinus, cosinus, racine carrée, etc. via la classe System.Math disponibles dans le Framework .NET (cf. chapitre .NET sur l'utilisation du Framework et des types .NET).

b. Les opérateurs de comparaison

Avec un nom aussi évocateur, inutile de préciser que les opérateurs de comparaison vont nous permettre de faire des comparaisons de variables. En effet, lors de l'utilisation des structures conditionnelles que nous aborderons plus tard dans ce chapitre, nous utilisons ces fameux opérateurs pour obtenir un résultat de type booléen, c'est à dire Vrai (True) ou Faux (False), sur une comparaison donnée. Pour connaître les différentes comparaisons possibles, jetons un coup d'œil sur l'ensemble des opérations de comparaison.

Opérateur	Signification
-eq	Egal
-ne	Non égal (différent)
-gt	Strictement supérieur
-ge	Supérieur ou égal
-lt	Strictement inférieur
-le	Inférieur ou égal

À noter que les opérateurs de comparaison ne respectent pas la casse, c'est-à-dire les minuscules et les majuscules, lors d'une comparaison de chaîne. Pour remédier à cela faites simplement précéder le nom de l'opérateur de la lettre « c », comme par exemple -cle.



Pour que l'opérateur ne respecte pas la casse faites précéder le nom de l'opérateur de la lettre « i », comme par exemple -ile. Mais cela ne vous sera pas nécessaire car les opérateurs de comparaison ne respectent pas la casse par défaut.

c. Les opérateurs de comparaison générique

Une expression générique est une expression qui contient un caractère dit « générique ». Par exemple « * » pour signifier n'importe quelle suite de caractères, ou un « ? » pour un unique caractère. Il existe deux opérateurs de comparaison qui vous permettent de comparer une chaîne avec une expression générique.

Opérateur	Signification
-----------	---------------

-like	Comparaison d'égalité d'expression générique.
-notlike	Comparaison d'inégalité d'expression générique.

Pour mieux comprendre l'utilisation de ces opérateurs, voici quelques exemples d'applications :

```
PS > 'Powershell' -like '*shell'
True

PS > 'powershell' -like 'power*'
True

PS > 'powershell' -like '*wer*'
True

PS > 'powershell' -like '*war*'
False

PS > 'powershell' -like 'po?er*'
True

PS > 'power' -like 'po?er*'
True

PS > 'potter' -like 'po?er*'
False
```



L'opérateur de comparaison générique peut (comme les opérateurs de comparaison) ou non respecter la casse. Si vous souhaitez que l'opérateur respecte la casse, faites précéder le nom de l'opérateur de la lettre « c ». Pour faire le contraire, faites précéder le nom de la lettre « i ».

d. Les opérateurs de comparaison des expressions régulières

Une expression régulière appelée également « RegEx » est une expression composée de ce que l'on appelle des « métacaractères », qui vont correspondre à des valeurs particulières de caractères.

Si vous n'avez jamais entendu parler d'expressions régulières, nous vous conseillons grandement de jeter un œil sur les nombreux ouvrages traitant de ce sujet ou bien encore de consulter l'aide en ligne (Help about_Regular_Expression) qui est bien fournie.

PowerShell dispose de deux opérateurs de comparaison d'expressions régulières, qui vont nous retourner un booléen selon le résultat obtenu lors de la comparaison.

Opérateur	Signification
-match	Comparaison d'égalité entre une expression et une expression régulière.
-notmatch	Comparaison d'inégalité entre une expression et une expression régulière.

Pour mieux comprendre l'utilisation de ces opérateurs, voici quelques exemples d'applications :

```
PS > 'Powershell' -match 'power[so]hell'
True

PS > 'powershell' -match 'powershel[a-k]'
False

PS > 'powershell' -match 'powershel[a-z]'
True
```



L'opérateur de comparaison d'expression régulière peut, comme les opérateurs de comparaison, respecter ou non la casse. Pour que l'opérateur respecte la casse faites précéder le nom de l'opérateur de la lettre « c », pour faire le contraire, faites précéder le nom de la lettre « i ».

e. Les opérateurs de plage

L'opérateur de plage se note : « .. » (prononcez, point point). Il permet comme son nom l'indique de couvrir une plage de valeurs sans pour autant avoir à les saisir. Admettons que nous souhaitions couvrir une plage de valeurs allant de 1 à 10 (pour réaliser une boucle par exemple), et bien il suffit de taper la ligne qui suit :

```
PS > 1..10
```

On peut, de la même manière définir une plage dynamiquement en utilisant des variables. Rien ne vous empêche de définir une plage allant de \$var1 à \$var2 si ces valeurs sont des entiers.

Exemple :

```
PS > $var1 = 5
PS > $var2 = 10
PS > $var1 .. $var2
5
6
7
8
9
10
```

f. L'opérateur de remplacement

L'opérateur de remplacement permet de remplacer toute ou partie d'une valeur par une autre. Admettons que notre variable soit du type chaîne, que son contenu soit « PowerShell », et que nous souhaitions remplacer « Shell » par « Guy ».

Il faut donc utiliser l'opérateur de remplacement (-replace) suivi de la partie à remplacer et de la nouvelle valeur.

Exemple :

```
PS > 'PowerShell' -replace 'Shell', 'Guy'
PowerGuy
```



L'opérateur de remplacement peut (comme les opérateurs de comparaison) ou non respecter la casse. Pour que l'opérateur respecte la casse, faites précéder le nom de l'opérateur de la lettre « c », pour faire le contraire, faites précéder le nom de la lettre « i ».



Les chaînes de caractères (string) possèdent une méthode appelée Replace qui effectue la même chose.

Exemple :

```
PS > $MaChaine = 'PowerShell'
PS > $MaChaine.Replace('Shell', 'Guy')
PowerGuy
```

g. Les opérateurs de type

Jusqu'à présent, nous vous avons montré comment typer votre valeur et même comment récupérer le type avec la méthode GetType. Mais ce que nous allons désormais découvrir, est comment tester le type d'une variable. Par exemple, nous pourrions très bien être intéressés de savoir si une variable est de type « int » de façon à pouvoir lui attribuer une valeur entière. Et bien tout ceci s'effectue avec les deux opérateurs de type que voilà :

Opérateur	Signification

-is	Test si l'objet est du même type.
-isnot	Test si l'objet n'est pas du même type.

Pour mieux comprendre l'utilisation de ces opérateurs, voici quelques exemples d'applications :

```
PS > 'Bonjour' -is [string]
True

PS > 20 -is [int]
True

PS > 'B' -is [int]
False
```

h. Les opérateurs logiques

Les opérateurs logiques permettent de vérifier jusqu'à plusieurs comparaisons dans une même expression. Par exemple : `($var1 -eq $var2) -and ($var3 -eq $var4)`, vous renverra le booléen *true* si `$var1` est égale à `$var2` et que `$var3` est égale à `$var4`, dans le cas contraire la valeur *false* sera renvoyée. Voici la liste des opérateurs logiques disponibles :

Opérateur	Signification
-and	Et logique
-or	Ou logique
-not	Non logique
!	Non logique
-xor	OU exclusif

Pour mieux comprendre l'utilisation de ces opérateurs, voici quelques exemples d'applications :

```
PS > (5 -eq 5) -and (8 -eq 9)
False
```

Faux, car 5 est bien égal à 5, mais 8 n'est pas égal à 9.

```
PS > (5 -eq 5) -or (8 -eq 9)
True
```

Vrai, car l'une des deux expressions est vraie, 5 est bien égal à 5.

```
PS > -not (8 -eq 9)
True

PS > !(8 -eq 9)
True
```

Vrai, car 8 n'est pas égal à 9.

i. Les opérateurs binaires

Les opérateurs binaires sont utilisés pour effectuer des opérations entre nombres binaires. Pour rappel, le système binaire est un système en base 2, contrairement au système décimal qui lui est en base 10. C'est-à-dire que la notation ne comporte que des « 0 » et des « 1 ».

Exemple de conversion de nombres décimaux en base binaire :

Décimal	Binaire
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101

Lorsque nous faisons appel à l'un des opérateurs binaires suivant, les bits des valeurs sont comparés les uns après les autres, puis selon que nous appliquons un ET ou un OU nous obtiendrons un résultat différent.

Opérateur	Signification
-band	Opérateur ET
-bor	Opérateur OU
-bnot	Opérateur NON
-bxor	Opérateur OU Exclusif

➤ Le résultat retourné après une comparaison binaire est automatiquement converti en système décimal et non pas en système binaire.

Imaginons que pour une application quelconque nous souhaitions savoir si le bit de poids faible d'une variable est égal à 1. Prenons pour exemple la valeur décimale 13, soit 1101 en binaire. Alors évidemment on voit clairement que le bit de poids faible est bien à 1, mais pour vérifier cette affirmation via PowerShell, utilisons plutôt notre opérateur binaire *-band*.

En utilisant cet opérateur, nous allons en fait réaliser ce que l'on appelle un masque sur le bit de poids faible. Si le résultat est conforme au masque appliqué alors le bit de poids faible est bien à la valeur 1. Voici ce que donnerait graphiquement la comparaison :

$$\begin{array}{r}
 1101 \\
 0001 \\
 \hline
 0001
 \end{array}$$

Opérateur ET

Masque sur le bit de poids faible

Résultat :

```
PS > $var = 13
PS > $var -band 1
1
```

➤ PowerShell 1.0 utilise des opérateurs de bits travaillant sur des entiers de 32 bits (valeurs comprises entre -2 147 483 648 et 2 147 483 647). La version 2.0 quant à elle, permet de travailler sur 64 bits (couvrant les valeurs allant de -9223372036854775807 à 9223372036854775807).

j. Les opérateurs d'affectation

Vous savez donc maintenant comment affecter une valeur à une variable et réaliser une opération sur cette dernière. Maintenant nous allons vous montrer comment faire les deux en même temps en une seule opération.

Les opérations qui sont décrites dans ce tableau donnent strictement le même résultat.

Notation classique	Notation raccourcie
<code>\$i=\$i+8</code>	<code>\$i+=8</code>
<code>\$i=\$i-8</code>	<code>\$i-=8</code>
<code>\$i=\$i*8</code>	<code>\$i*=8</code>
<code>\$i=\$i/8</code>	<code>\$i/=8</code>
<code>\$i=\$i%8</code>	<code>\$i%=8</code>
<code>\$i=\$i+1</code>	<code>\$i++</code>
<code>\$i=\$i-1</code>	<code>\$i--</code>

➤ La notation `$i++` ou `$i--` est très utilisée dans les conditions de boucle de façon à incrémenter `$i` de 1 à chaque passage.

Ainsi, par exemple, voici comment ajouter une valeur avec l'opérateur d'affectation « += ».

```
PS > $i = 0
PS > $i += 15
PS > $i
15
```

Si on affiche la valeur de la variable `$i` on obtient bien 15, car cette instruction est équivalente à : `$i = $i + 15`.

Poursuivons avec cette fois le calcul des factoriels des chiffres allant de 1 à 10.

Pour cela, nous allons créer une boucle et, pour chaque valeur de `$i`, nous la multiplierons par la valeur de `$var` avant de la lui réaffecter :

```
PS > $var = 1
PS > foreach($i in 1..10){$var *= $i ; $var}
1
2
6
24
120
720
5040
40320
362880
3628800
```

➤ Comme nous n'avons pas encore abordé la notion de boucle `foreach`, n'y prêtez pas trop attention dans cet exemple. L'essentiel est que ayez compris qu'il s'agit d'une boucle, allant de 1 à 10, dans laquelle pour chaque valeur de `i`, on multiplie la valeur de `$i` par la valeur de `$var` et on enregistre le tout dans `$var`.

k. Les opérateurs de redirection

Ce qu'il faut savoir, c'est que les interpréteurs de commandes traitent les informations selon une entrée, une sortie et une erreur standard, chaque élément étant identifié par un descripteur de fichier. L'entrée standard, se voit attribuer le descripteur 0, la sortie standard le 1 et l'erreur standard le 2. Par défaut, c'est le clavier qui est utilisé comme entrée standard, et l'affichage dans la console l'est pour la sortie. Mais de façon à rediriger ces flux d'information avec plus de souplesse, PowerShell dispose d'une batterie d'opérateurs, identiques à ceux utilisés dans l'interface en ligne de commande d'Unix :

Opérateur	Signification
-----------	---------------

>	Redirige le flux vers un fichier, si le fichier est déjà créé, le contenu du fichier précédent est remplacé.
>>	Redirige le flux dans un fichier, si le fichier est déjà créé, le flux est ajouté à la fin du fichier.
2>&1	Redirige les messages d'erreurs vers la sortie standard.
2>	Redirige l'erreur standard vers un fichier, si le fichier est déjà créé, le contenu du fichier précédent est remplacé.
2>>	Redirige l'erreur standard vers un fichier, si le fichier est déjà créé, le flux est ajouté à la fin du fichier.

Supposons que nous souhaitions envoyer le résultat d'une commande dans un fichier texte plutôt qu'à l'intérieur de la console, pour cela utilisons l'opérateur « > » :

```
PS > Get-Process > c:\temp\process.txt
```

Nous obtenons un fichier texte dans c:\temp du nom de process.txt qui contient le résultat de la commande.

```
PS > Get-Content c:\temp\process.txt
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
66	4	1768	3520	57	0,14	4200	acrotray
62	2	940	2924	31		1424	audiodg
297	76	4832	452	72	0,70	2096	ccApp
851	14	13008	7532	105		548	CcmExec
497	11	9528	5320	87		1800	ccSvcHst
34	2	796	3464	36	0,05	5152	conime
582	5	1712	3124	87		768	csrss
301	10	2732	12728	135		4784	csrss
202	6	2256	5720	70		540	DefWatch
82	3	1388	4436	45	0,09	2636	dwm
678	27	27960	41488	189	20,42	368	explorer
0	0	0	16	0		0	Idle

Maintenant, si nous écrivons de nouveau une commande dont la sortie serait dirigée dans le même fichier via l'opérateur « > », les données seront écrasées. Le contenu du fichier est effacé et remplacé par la nouvelle sortie. Pour éviter cela, il faut utiliser l'opérateur « >> » qui indique à PowerShell d'ajouter la sortie de la commande à la fin du fichier spécifié.



Pour rediriger un flux vers un fichier, vous pouvez aussi utiliser la commandelette Out-File à la place des opérateurs de redirection, notamment si vous souhaitez utiliser des paramètres tels que l'encodage, le nombre de caractères dans chaque ligne de sortie, etc. Mais tout cela sera expliqué en détail dans le chapitre Maîtrise du Shell.

Dernier exemple, la redirection de l'erreur standard vers un fichier. Pour cela utilisons simplement une commande susceptible de retourner un message d'erreur, comme Get-ChildItem sur un répertoire inexistant. Puis envoyons le tout dans un fichier via l'opérateur « 2> ».

```
PS > Get-ChildItem c:\temp\RepInexistant 2> c:\err.txt
```

Aucun message n'est affiché dans la console. Mais en récupérant le contenu du fichier err.txt, on s'aperçoit qu'il contient bien le message d'erreur relatif à la commande saisie.

```
PS > Get-Content c:\err.txt
Get-ChildItem : Impossible de trouver le chemin d'accès
« C:\temp\RepInexistant », car il n'existe pas.
```

I. Opérateurs de fractionnement et de concaténation

Les opérateurs de fractionnement et de concaténation sont uniquement disponibles dans la version 2.0 de PowerShell. Ils permettent de combiner ou bien de fractionner à volonté des chaînes de caractères.

Opérateur	Signification
-split	Fractionne une chaîne en sous-chaînes.
-join	Concatène plusieurs chaînes en une seule.

Ainsi, par exemple, voici comment fractionner une chaîne en plaçant l'opérateur -split en début de ligne.

```
PS > -split 'PowerShell c'est facile'
PowerShell
c'est
facile
```

Par défaut, le fractionnement est réalisé avec pour délimiteur l'espace blanc. Pour changer ce délimiteur, il convient de placer l'opérateur en fin de ligne et de le faire suivre du caractère délimiteur souhaité.

Exemple :

```
PS > 'Nom:Prenom:Adresse:Date' -split ':'
Nom
Prenom
Adresse
Date
```

L'opérateur -join permet de réaliser la concaténation de différentes chaînes de caractères d'un même tableau.

Exemple :

```
PS > $tableau = 'Lundi', 'Mardi', 'Mercredi', 'jeudi', 'Vendredi',
'Samedi', 'Dimanche'
PS > -join $tableau
LundiMardiMercredijeudiVendrediSamediDimanche
PS > $tableau -join ', puis '
Lundi, puis Mardi, puis Mercredi, puis jeudi, puis Vendredi,
puis Samedi, puis Dimanche
```

m. Récapitulatif sur les opérateurs

Dans cette liste vous retrouverez tous les opérateurs déjà énoncés au cours de ce chapitre (les opérateurs signalés d'une étoile ne sont disponibles qu'avec PowerShell v2).

Opérateur	Signification
-eq	Égal.
-lt	Inférieur à.
-gt	Supérieur à.
-le	Inférieur ou égal à.
-ge	Supérieur ou égal à.
-ne	Différent de.
-not	Non logique.
!	Non logique.
-match	Comparaison d'égalité entre une expression et une expression régulière.

-notmatch	Comparaison d'inégalité entre une expression et une expression régulière.
-like	Comparaison d'égalité d'expression générique.
-notlike	Comparaison d'inégalité d'expression générique.
-replace	Opérateur de remplacement.
-and	ET logique.
-or	OU logique.
-bor	Opérateur de bits OU.
-band	Opérateur de bits ET.
-bxor	Opérateur de bits OU EXCLUSIF.
-xor	OU EXCLUSIF.
-is	Opérateur d'égalité de type.
-isnot	Opérateur d'inégalité de type.
-ceq	Égal (respecte la casse).
-clt	Inférieur à (respecte la casse).
-cgt	Supérieur à (respecte la casse).
-cle	Inférieur ou égal à (respecte la casse).
-cge	Supérieur ou égal à (respecte la casse).
-cne	Différent de (respecte la casse).
-cmatch	Comparaison d'égalité entre une expression et une expression régulière (respecte la casse).
-cnotmatch	Comparaison d'inégalité entre une expression et une expression régulière (respecte la casse).
-clike	Comparaison d'égalité d'expression générique (respecte la casse).
-cnotlike	Comparaison d'inégalité d'expression générique (respecte la casse).
-creplace	Opérateur de remplacement (respecte la casse).
>	Redirige le flux vers un fichier, si le fichier est déjà créé, le contenu du fichier précédent est remplacé.
>>	Redirige le flux dans un fichier, si le fichier est déjà créé, le flux est ajouté à la fin du fichier.
2>&1	Redirige les messages d'erreurs vers la sortie standard.
2>	Redirige l'erreur standard vers un fichier, si le fichier est déjà créé, le contenu du fichier précédent est remplacé.

2>>	Redirige l'erreur standard vers un fichier, si le fichier est déjà créé, le flux est ajouté à la fin du fichier.
-split (*)	Fractionne une chaîne en sous-chaînes.
-join (*)	Concatène plusieurs chaînes en une seule.

Les alias

Les alias sont ce que l'on pourrait appeler « surnoms d'une commande », ils sont souvent utiles lorsque l'on utilise des commandes un peu longues à taper. Ainsi, l'alias « commande » pourrait par exemple être attribué à la commande « commandevraimenttrophlongue ».

Pour ceux qui sont déjà habitués au Shell Unix ou au CMD.exe et qui ont leurs petites habitudes, PowerShell a pensé à eux et leur facilite la tâche grâce à des alias de commande mode « Unix » / « CMD » de façon à ne pas les déstabiliser. Par exemple les utilisateurs Unix peuvent utiliser quelques commandes comme : ls, more, pwd, etc.

Ces commandes sont des alias de commande préenregistrées dans PowerShell. Par exemple, ls est un alias de la commande **Get-ChildItem** qui liste les fichiers et les répertoires.

1. Lister les alias

Pour rechercher tous les alias de votre session, aussi bien ceux déjà prédéfinis que ceux que vous avez créés, tapez tout simplement : **Get-Alias**

PS > 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
...		

Exemple :

PS > Get-Alias -Name cd		
CommandType	Name	Definition
-----	----	-----
Alias	cd	Set-Location

Le nom de paramètre -Name est facultatif.



Retrouvez la liste complète de tous les alias et commandes associées en annexe Liste des alias.

2. Les commandes appliquées aux alias

Vous vous en doutiez sûrement, il est possible de créer de nouveaux alias, tout comme il est possible d'en modifier et d'en supprimer. Pour cela, PowerShell met à disposition cinq commandelettes pour agir sur les alias :

Get-Alias : comme nous venons de le voir, cette commandelette permet d'obtenir tous les alias de la session active.

New-Alias : cette commande permet de créer un alias.

Exemple :

```
PS > New-Alias -Name Grep -Value Select-String
```

Set-Alias : cette commande permet de créer ou de modifier un alias.

Exemple :

```
PS > Set-Alias -Name write -Value Write-Host
```

La différence avec la commande new-alias est que si l'alias existe déjà, elle modifie les valeurs de ce dernier.

Export-Alias : exporte un ou plusieurs alias vers un fichier, si le fichier de sortie spécifié n'existe pas, la commandelette le crée.

Exemple :

```
PS > Export-Alias -Path c:\temp\alias.txt
```

Et voici le résultat contenu dans le fichier texte :

```
PS > Get-Content c:\temp\alias.txt

# Fichier d'alias
# Exporté par : Edouard Bracame
# Date/heure : vendredi 10 septembre 2009 23:14:47
# Ordinateur : WIN7-BUREAU
"ac","Add-Content","", "ReadOnly, AllScope"
"asnp","Add-PSSnapIn","", "ReadOnly, AllScope"
"clc","Clear-Content","", "ReadOnly, AllScope"
"cli","Clear-Item","", "ReadOnly, AllScope"
"clp","Clear-ItemProperty","", "ReadOnly, AllScope"
"clv","Clear-Variable","", "ReadOnly, AllScope"
```

Import-Alias : importe un fichier d'alias dans Windows PowerShell.

Exemple :

```
PS > Import-Alias -Path c:\temp\alias.txt
```



Les alias peuvent être utilisés sur des commandes, des fichiers ou des fichiers exécutables, mais il est impossible d'y faire figurer des paramètres. Mais rien ne vous empêche d'écrire un script ou une fonction qui utilise des commandes avec arguments.



Les créations et modifications d'alias faites en cours de session sont perdues une fois cette session fermée. Pour retrouver vos alias personnalisés à chaque session, vous devrez les déclarer dans un fichier script particulier, appelé profil, qui est chargé automatiquement au démarrage de chaque session PowerShell. Nous aborderons la notion de profil dans le chapitre Maîtrise du Shell.

L'info en plus

Le lecteur attentif que vous êtes, se rappellera qu'au chapitre "À la découverte de PowerShell" nous vous avons parlé des fournisseurs. Et bien l'un d'eux s'appelle « alias ». Et contient, comme son nom l'indique, la liste des alias.

Pour rappel, afin d'obtenir la liste et les détails associés aux fournisseurs, tapez la commande **Get-PsProvider**. La navigation à l'intérieur de ces lecteurs se fait exactement de la même manière que pour explorer un système de fichiers sur un disque dur. Exemple, si vous souhaitez obtenir tous les alias commençant par la lettre « f », tapez :

```
PS > Get-ChildItem alias:f*
```

CommandType	Name	Definition
-----	----	-----
Alias	fc	Format-Custom
Alias	fl	Format-List
Alias	foreach	ForEach-Object
Alias	ft	Format-Table
Alias	fw	Format-Wide

Pour plus d'informations sur les fournisseurs, reportez-vous au chapitre "À la découverte de PowerShell".

Tableaux

1. Tableaux à une dimension

Le tableau à une dimension est le cas le plus simple, les valeurs sont mises les unes après les autres, et il suffit d'indiquer le numéro d'indice pour utiliser le contenu. Un tableau à une dimension est parfois appelé liste.

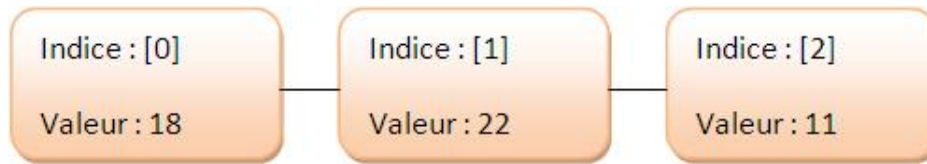


Illustration d'un tableau à une dimension

Par exemple ici :

La valeur 18 est contenue dans le tableau à l'indice 0.

La valeur 22 est contenue dans le tableau à l'indice 1.



Avec PowerShell les indices de tableau commencent à 0 et non pas à 1 comme avec d'autres langages.

a. Initialiser un tableau à une dimension

Pour à la fois créer un tableau et l'initialiser, il suffit de lui affecter plusieurs valeurs séparées par une virgule. Par exemple : `$tab = 1,5,9,10,6` est un tableau de type entier qui va contenir 1 à l'indice 0, puis 5 à l'indice 1, puis 9 à l'indice 2, etc.



Un tableau peut aussi s'initialiser avec l'opérateur de plage, exemple : `$tab = 1..20` est un tableau d'entier qui va contenir toutes les valeurs allant de 1 à 20.

À noter que le type d'objet rentré dans le tableau est attribué de façon automatique, mais comme pour les variables simples, vous pouvez forcer le type des données contenues dans le tableau.

Exemple :

```
PS > [int[]]$tab = 1,2,3
```

Vous noterez les crochets `[]` immédiatement après le nom du type. Ces crochets symbolisent le fait qu'il s'agit d'un tableau de valeurs du type en question. Dans cet exemple, le tableau `$tab` ne peut contenir que des entiers.

Mais un tableau peut aussi être hétérogène, et dans ce cas, l'affectation des types se fait valeur par valeur.

Exemple :

```
PS > $tab = [int]1,[double]2.5,[char]'A'
```

b. Lire les tableaux à une dimension

Pour lire un tableau à une dimension plusieurs méthodes sont possibles.

La plus simple étant de saisir son nom dans la console, dans ce cas, tous les éléments du tableau seront donc affichés. Mais pour lire une valeur à un indice précis, il suffit d'indiquer entre crochets l'indice voulu.

```
PS > $tab[0]  
1
```


Pour lire plusieurs valeurs à des indices précis, il suffit, cette fois-ci, d'indiquer entre crochets les indices séparés par des virgules.

```
PS > $tab[0,2]
1
3
```

Ici, seules les valeurs à l'indice 0 et 2 sont obtenues, la valeur à l'indice 1 ne l'est pas.



Vous pouvez aussi afficher plusieurs valeurs avec l'opérateur de plage, exemple : `$tab[1..20]` ceci affichera les valeurs de l'indice 1 à 20.

Maintenant, supposons que nous souhaitions uniquement lire la valeur contenue au dernier indice. Une des méthodes consiste à savoir combien de valeurs sont contenues dans notre tableau. Ceci se fait grâce à la propriété `Length` :

```
PS > $tab[$tab.Length-1]
3
```

Notez que nous enlevons une unité à la propriété `Length` parce que les indices commencent à 0 et non à 1.

Mais il y a une autre méthode plus simple : les indices négatifs.

Lorsque vous utilisez un indice négatif, vous faites référence au nombre d'indices depuis la fin du tableau.

Exemple :

```
PS > $tab[-1]
3
PS > $tab[-3..-1]
1
2
3
```

La méthode la plus courante pour lire un tableau reste toutefois le parcours de tableaux avec des boucles (*While*, *For*, *Foreach*). Pour en savoir plus, reportez-vous à la section Les boucles (*While*, *For* et *Foreach*) de ce chapitre sur les boucles et conditions.

c. Opérations sur les tableaux à une dimension

Concaténer deux tableaux

Avec PowerShell la concaténation de tableaux se fait avec l'opérateur « + ». Supposons que pour un motif quelconque nous ayons besoin de concaténer deux tableaux (ou plus). Pour cela, il suffit d'additionner les tableaux par l'opérateur « + ».

Exemple avec l'addition de deux tableaux de caractère nommés `$chaine1` et `$chaine2` :

```
PS > $chaine1 = 'P','o','w','e','r'
PS > $chaine2 = 'S','h','e','l','l'
PS > $chaine1 + $chaine2
P
o
w
e
r
S
h
e
l
l
```

Ajouter un élément à un tableau

En PowerShell, l'ajout d'une valeur à un tableau se fait avec l'opérateur « += ». Ainsi en tapant la ligne suivante, nous ajoutons la valeur 4 à notre tableau :

```
PS > $tab = 1, 2, 3
PS > $tab += 4
PS > $tab
1
2
3
4
```

Modifier la valeur d'un élément

La modification d'un élément dans un tableau se fait avec l'opérateur « = ».

Exemple, en tapant \$tab[2]=1 nous allons modifier la valeur contenue à l'indice 2 (la troisième valeur, donc). En réalité, c'est une nouvelle affectation qui est réalisée, et celle-ci écrasera l'ancienne valeur par la nouvelle.

Exemple :

```
PS > $tab = 'A', 'B'
PS > $tab[0] = 'C'
PS > $tab
C
B
```

Il existe une deuxième technique pour modifier une valeur existante. Pour cela, il nous faut faire appel à une méthode spécifique aux objets de type tableau : SetValue.

En utilisant SetValue et en lui indiquant en paramètre la nouvelle valeur puis l'indice du tableau nous réalisons une affectation.

```
PS > $tab = 'A', 'B'
PS > $tab.SetValue('C', 0)
PS > $tab
C
B
```

d. Supprimer un élément

Avec PowerShell, il n'est pas possible de supprimer un élément d'un tableau. Enfin en y réfléchissant bien, il y a une explication logique : à chaque suppression d'élément, cela entraînerait un réajustement des indices pour chaque valeur, et on serait vite perdu. Cependant, il existe une solution alternative, permettant de contourner le problème. Celle-ci consiste à effectuer une recopie d'un tableau en y excluant un ou plusieurs indices.

Exemple : Suppression d'éléments dans un tableau

Prenons l'exemple de vos dernières notes à l'examen de fin d'étude.

```
PS > $notes = 12, 18, 10, 14, 8, 11
```

N'ayant pas brillé en algorithmique (8) vous décidez de supprimer cette note qui ne vous satisfait pas du tout.

Et bien pour cela, procédons tout simplement à la recopie des éléments du tableau à l'exception de la valeur à l'indice 4 :

```
PS > $notes = $notes[0..3 + 5]
PS > $notes
12
18
10
14
11
```

Si l'on ne connaît pas les indices, ou si le nombre de notes à supprimer est trop important, on peut aussi procéder par ce que l'on appelle un filtre. Bien que nous n'ayons pas encore abordé les filtres, voici comment grâce à un filtre et à un opérateur de comparaison, nous pouvons obtenir une recopie de tableau avec uniquement les valeurs supérieures à 10.

```
PS > $notes2 = $notes | where-object {$_ -ge 10}
PS > $notes2
12
18
10
14
11
```

2. Tableaux à plusieurs dimensions

Lorsque l'on parle de tableaux à plusieurs dimensions, on parle de tableaux à plusieurs index, avec autant d'index que de dimensions. Ainsi, pour passer d'un tableau à une dimension à un tableau à deux dimensions, il suffit d'ajouter un indice permettant de se repérer dans cette nouvelle dimension.

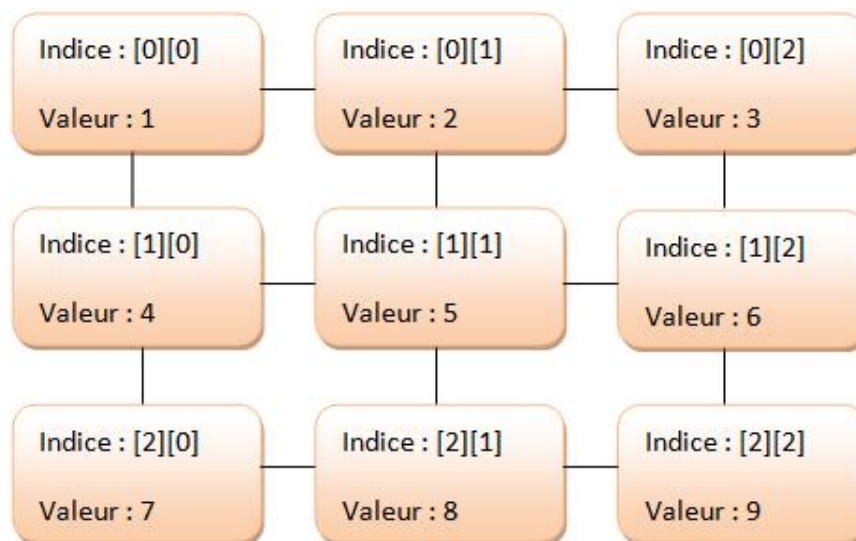


Illustration d'un tableau à deux dimensions

La lecture des tableaux à plusieurs dimensions est semblable à ceux à une dimension. La seule contrainte est de jouer avec les indices. Prenons le cas du tableau ci-dessus.

La lecture du tableau avec l'indice « 0 », nous donnera la première ligne de ce tableau :

```
PS > $tab[0]
1
2
3
```

Pour obtenir une valeur précise, nous devons tout simplement fixer l'indice de la dimension horizontale et celui de la verticale.

```
PS > $tab[0][2]
3
```

Tableaux associatifs

Un tableau associatif est un tableau dans lequel chaque valeur n'est pas référencée par un indice mais par une clé. Jusque là, nous avons vu que dans un tableau, chaque valeur était indexée numériquement. Et bien dans un tableau associatif cette notion d'indexation numérique n'existe plus, on utilise des clés qui sont utilisées comme identifiant. Par exemple, voici un tableau associatif dans lequel chaque valeur est un prix à laquelle est associée une clé, qui représente un produit.

Clé	Valeur
Vidéo_projecteur	1600
Télévision	1400
Console_de_jeux	400

 Avec les tableaux associatifs, tout comme les tableaux classiques, vous pouvez utiliser des types de données hétérogènes.

Pour initialiser un tableau associatif il vous faut utiliser la syntaxe suivante :

```
$<nom_tableau> = @{<clé1 = élément1>; <clé = élément2>;...}
```

Notez que la création d'un tableau associatif nécessite de bien insérer le signe « @ » en tête, de séparer toutes les valeurs par des points-virgules ainsi que d'affecter une clé à chaque élément.

Reprenons notre exemple avec les produits décrits précédemment. Voici à quoi ressemble l'initialisation du tableau :

```
PS > $catalogue = @{ Video_projecteur = 1600 ;  
Television = 1400 ;  
Console_de_jeux = 400 }
```

Pour ensuite pouvoir lire les valeurs contenues dans le tableau, il existe deux méthodes, soit nous tapons simplement le nom du tableau dans la console :

```
PS > $catalogue  
  
Name                Value  
----                -  
Console_de_jeux     400  
Television           1400  
Video_projecteur     1600
```

Soit nous choisissons d'afficher élément par élément, dans ce cas il nous faut utiliser la notation par point ou crochet :

```
PS > $catalogue['Console_de_jeux']  
400  
  
PS > $catalogue.Television  
1400
```

 Si votre clé ou votre valeur contient des espaces, n'oubliez pas d'insérer des simples guillemets ' '.

Redirections et Pipeline

1. Le pipeline

Avec PowerShell, il est possible de connecter des commandes, de telle sorte que la sortie de l'une devienne l'entrée de l'autre. C'est ce qu'on appelle le pipeline.

Ce canal de communication établit entre un émetteur et un récepteur une liaison sur laquelle sont transportées les données sous forme d'objet.

Explication : Le pipeline, signifie « canalisation » en anglais, et sert à établir une liaison entre deux commandes. Matérialisé par le caractère « | » [Alt Gr] 6 (ASCII décimal 124), il transfère la sortie de la commande qui le précède vers l'entrée de la commande qui le succède. Par exemple :

```
PS > Get-Command | Out-File -FilePath C:\temp\fichier.txt
```

Dans la commande précédente, la sortie de la commande **Get-Command**, qui renvoie la liste des commandes disponibles, est envoyée à la commande **Out-File** qui va se charger à son tour de l'envoyer dans un fichier texte.

Toujours dans le même registre, la commande **Out-null** supprime immédiatement toute entrée qu'elle reçoit.

```
PS > Get-Command | Out-null
```

Bien évidemment, plusieurs pipelines peuvent être utilisés sur une même ligne de commande. Dans ce cas, chaque commande, à l'exception de celles aux extrémités, recevra un objet en entrée à travers le pipeline, et fournira l'objet retourné vers le pipeline suivant. Prenons par exemple le cas de ligne suivante :

```
PS > Get-ChildItem C:\temp | ForEach-Object  
{$_ .Get_extension().ToLower()} | Sort-Object | Get-Unique |  
Out-File -FilePath C:\temp\extensions.txt -Encoding ASCII
```

Cinq instructions sur une ligne le tout passant par des pipelines. Alors certes l'expression devient un peu chargée, mais en revanche, une seule ligne aura suffi pour faire tout ça.

Voici le contenu de la commande en détail :

1^{ère} instruction : grâce au **Get-ChildItem C:\temp** on va lister tous les éléments du répertoire C:\temp,

2^{ème} instruction : le **ForEach-object** nous permet pour chaque élément, d'afficher son extension et la convertir en minuscules,

3^{ème} instruction : **Sort-Object** trie par ordre alphabétique les éléments,

4^{ème} instruction : **Get-Unique** supprime les occurrences en doublon,

5^{ème} instruction : et enfin, **Out-File -FilePath C:\temp\extensions.txt -Encoding ASCII**, envoie le tout dans un fichier texte en mode ASCII.

Reste maintenant à vérifier le contenu du fichier C:\temp\extensions.txt par le moyen de la commande **Get-Content** :

```
PS > Get-Content C:\temp\extensions.txt  
  
.doc  
.gzip  
.lnk  
.pdf  
.ppt  
.ps1  
.rnd  
.txt
```

a. Filtre Where-Object

La commande **Where-Object** (alias : **Where**) est très utilisée dans les pipelines. Elle fait référence aux objets retournés par la commande précédente et permet d'agir dessus de façon à ne garder que ceux qui nous intéressent. Par exemple, supposons que nous utilisons la commande **Get-Service** pour lister les services, jusque là tout va bien. Maintenant imaginons que l'on souhaite lister uniquement les services stoppés ! C'est là qu'intervient

l'instruction **Where-Object**. C'est en passant le résultat de la commande **Get-Service** au travers du pipeline, que la commandelette **Where-Object** associée à une expression de comparaison, va récupérer les sous ensembles qui correspondent aux services arrêtés :

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

Status	Name	DisplayName
Stopped	Alerter	Avertissement
Stopped	aspnet_state	Service d'état ASP.NET
Stopped	ATI Smart	ATI Smart
Stopped	AutoExNT	AutoExNT

Vous noterez l'utilisation de la variable `$_` représentant l'objet courant passé par le pipe, ici en l'occurrence `$_` fait référence aux services.

Exemple : Liste des fichiers dont la taille excède 500 octets

Pour lister les fichiers dont la taille est supérieure à 500 octets, nous allons utiliser un filtre sur la propriété **Length** de chaque élément retourné par la commandelette **Get-ChildItem**.

```
PS > Get-ChildItem | Where-Object {$_.length -gt 500}
```

Répertoire : Microsoft.PowerShell.Core\FileSystem::C:\Temp

Mode		LastWriteTime	Length	Name
-a---	11/12/2007	09:57	9444	Fichier1.txt
-a---	11/12/2007	10:46	19968	Fichier2.txt
-a---	11/12/2007	10:49	9892	Fichier3.txt

Exemple :

Liste des processus dont le temps d'occupation processeur est supérieur à 300 millisecondes. Toujours dans le même esprit, pour récupérer les processus dont le temps d'occupation processeur est supérieur à 300 millisecondes, nous allons filtrer tous les objets renvoyés par la commandelette **Get-Process** :

```
PS > Get-Process |
Where-Object {$_.TotalProcessorTime.totalmilliseconds -gt 300}
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
458	12	14976	13884	76	10,25	3828	explorer
373	11	5568	9744	97	1,06	2396	OUTLOOK
319	6	42152	30900	156	8,22	632	powershell
95	10	3388	4516	44	0,53	3724	RTNotify
531	29	49148	62856	346	348,41	284	WINWORD



Il est également possible de faire des filtres sous forme de fonction, pour cela reportez-vous à la partie sur les fonctions de ce chapitre.

Enfin, pour terminer, sachez que toutes les commandelettes n'acceptent pas l'entrée de pipeline. Seules celles ayant au moins un de leurs paramètres acceptant l'entrée de pipeline peuvent être utilisées ainsi. Pour connaître toutes les propriétés relatives aux paramètres d'une commande, tapez la commande **Help** avec le paramètre **-Full**.

Les boucles (While, For et Foreach)

Une boucle est une structure répétitive qui permet d'exécuter plusieurs fois les instructions qui se trouvent à l'intérieur du bloc d'instruction.

1. Boucle While

Cette boucle décrit un déroulement précis. Les instructions de cette boucle sont répétées tant que la condition de la boucle est satisfaite.

La syntaxe d'une boucle while est la suivante :

```
While (<condition>)  
{  
    #bloc d'instructions  
}
```

Et son fonctionnement est le suivant :

- La boucle évalue la condition,
- Si la condition est fausse, le bloc d'instruction n'est pas exécuté et la boucle se termine,
- Si la condition est vraie, alors cette fois le bloc d'instruction est exécuté,
- Retour à l'étape 1.

Voici un exemple basique d'une boucle while qui va lister les valeurs contenues dans un tableau. Dans cette boucle while, tant que la valeur \$nombre est strictement inférieure à la taille du tableau, le bloc d'instruction lit la valeur du tableau à l'indice \$nombre.

```
$nombre = 0  
$tab = 0..99  
  
While($nombre -lt $tab.Length)  
{  
    Write-Host $tab[$nombre]  
    $nombre++  
}
```

2. Boucle Do-While

La boucle Do-While s'apparente à la boucle while, à la différence près que le test de condition est effectué à la fin. La boucle Do-While se structure de la façon suivante :

```
Do  
{  
    #bloc d'instructions  
}  
While (<condition>)
```

Le test de condition étant à la fin, le bloc d'instruction est toujours exécuté au moins une fois, même si le test est faux. Par exemple, lorsque vous utiliserez la boucle suivante, l'utilisateur sera amené à saisir un nombre entre 0 et 10 une première fois. Si le nombre saisi ne s'avère pas compris entre ces valeurs, alors le bloc d'instruction sera exécuté de nouveau.

```
Do  
{  
    Write-host 'Entrez une valeur entre 0 et 10'  
  
    [int]$var = read-host
```

```
}  
While( ($var -lt 0 ) -or ($var -gt 10))
```

a. Boucle For

La boucle For permet d'exécuter un certain nombre de fois un bloc d'instruction.

Lorsque l'on utilise une boucle For, on y indique sa valeur de départ, la condition de répétition de la boucle et son pas d'incréméntation, c'est-à-dire la valeur dont elle est augmentée à chaque itération.

La syntaxe de la boucle For est la suivante :

```
For (<initial> ;<condition> ;<incrément>)  
{  
    #bloc d'instructions  
}
```

Son fonctionnement est le suivant :

1. L'expression initiale est évaluée, il s'agit en général d'une affectation qui initialise une variable,
2. La condition de répétition est évaluée,
3. Si la condition est fausse, l'instruction For se termine,
4. Si la condition est vraie, les instructions du bloc d'instruction sont exécutées,
5. L'expression est incrémentée avec le pas choisi et l'exécution reprend à l'étape 2.

Reprenons l'exemple du parcours d'un tableau, mais cette fois-ci avec une boucle For.

```
$tab = 0..99  
For($i=0 ;$i -le 99 ;$i++)  
{  
    Write-Host $tab[$i]  
}
```

Notez que l'incréméntation de la variable \$i peut également être faite dans le bloc d'instruction. Cet exemple donne le même résultat que le précédent.

```
$tab = 0..99  
For($i=0 ;$i -le 99)  
{  
    Write-Host $tab[$i]  
    $i++  
}
```

3. Boucle Foreach-Object

À proprement parler **Foreach-Object** est une commandelette et non une instruction de boucle. Cette commandelette également disponible sous l'appellation Foreach en raison d'un alias, permet de parcourir les valeurs contenues dans une collection. Sa syntaxe est la suivante :

```
Foreach ($<élément> in $<collection>)  
{  
    #bloc d'instructions  
}
```

Par exemple, si nous appliquons une boucle Foreach sur un Get-Process de la manière suivante, Foreach(\$element in get-process). Lors de la première itération, la variable \$commande représentera le premier objet que Get-Process va renvoyer. Chaque élément de la collection étant un objet, nous allons pouvoir agir sur leurs propriétés et méthodes. Puis au passage suivant \$element représentera le second objet que Get-Process va renvoyer, et ainsi de suite. La boucle ne s'arrête que lorsque toutes les valeurs contenues dans la collection ont été atteintes.

Exemple :

```
Foreach ($element in Get-Process)
```



```
{  
    Write-Host "$($element.Name) démarré le : $($element.StartTime)"  
}
```

Résultat :

```
Notepad      démarré le : 2/10/2008 09:57:00  
Powershell   démarré le : 2/10/2008 09:09:24  
WINWORD      démarré le : 2/10/2008 08:57:40
```

La commandelette **Foreach-Object** est également applicable aux pipelines. C'est-à-dire qu'elle va accepter de travailler avec une collection qui lui est passée au travers d'un pipe.

Par exemple :

```
PS > Get-Process | Foreach{$_Name} | Sort -unique
```

Cette ligne de commande affiche uniquement le nom des processus (grâce au Foreach) puis les trie dans l'ordre et supprime les doublons.

À la différence de Where-Object, Foreach-object ne fait aucun filtrage.

Par contre, Foreach-object permet une segmentation entre les tâches à effectuer avant le premier objet (paramètre begin), les tâches à effectuer pour chaque objet (paramètre process) et les tâches à effectuer après le dernier objet (paramètre end).

Exemple :

```
PS > Get-Process | Foreach-Object -begin {  
Write-Host "Début de liste des processus`n"} `~  
-process {$_Name } -End {  
Write-Host "`nfin de liste des processus `n"}
```

Ainsi, dans cette commande, nous effectuons un affichage de chaîne au début et à la fin du traitement qui précise les actions réalisées :

```
Début de liste des  processus  
  
acrotray  
alg  
ati2evxx  
...  
  
fin de liste des  processus
```



Remarquez l'utilisation des doubles guillemets dans l'exemple à cause du caractère d'échappement `n (saut de ligne). Sans ces doubles guillemets `n n'aurait pas été interprété en tant que tel.

Structure conditionnelle If, Else, ElseIf

Une structure conditionnelle permet, via une évaluation de la condition, d'orienter l'exécution d'instructions. La syntaxe d'une structure conditionnelle est la suivante :

```
If (condition)
{
    #bloc d'instructions
}
```

Pour mieux comprendre l'utilisation d'une structure conditionnelle, voici quelques exemples d'applications :

Imaginons que nous souhaitions déterminer si une valeur entrée par l'utilisateur est la lettre « A ». Pour cela, nous allons utiliser une structure conditionnelle avec une condition sur la valeur de la variable testée. En utilisant un opérateur de comparaison, la structure sera la suivante :

```
$var = Read-Host

If($var -eq 'A')
{
    Write-Host "Le caractère saisi par l'utilisateur est un 'A' "
}
```

Si la variable entrée par l'utilisateur est un « A », alors la commandelette **Write-Host** sera traitée, sinon, l'exécution poursuivra son cours.

À l'instruction If, peut être associée la clause Else. Cette clause, permet en cas de retour d'une valeur False d'orienter le traitement vers un second bloc d'instructions. Prenons l'exemple suivant :

```
If (($var1 -eq 15) -and ($var2 -eq 18))
{
    # Bloc d'instructions 1
}
Else
{
    # Bloc d'instructions 2
}
```

Dans un premier temps, PowerShell va évaluer la première condition, à savoir si la variable \$var1 est égale à 15. Si le test est bon alors la première condition prend la valeur true.

Puis, il va évaluer la seconde condition (\$var2 -eq 18). Si le test est bon alors la seconde condition \$var2 prend également la valeur true.

Puis, si les deux valeurs sont vraies, l'opérateur logique -and de la condition retournera la valeur true (vrai ET vrai = vrai), et ainsi le bloc d'instruction 1 sera exécuté, sinon, si la condition est fausse, ce sera le bloc d'instruction 2 qui sera exécuté.

Toujours dans le même registre voici un autre exemple :

```
[int]$var1 = Read-Host 'Saisissez un nombre'
[int]$var2 = Read-Host 'Saisissez un nombre'

If($var1 -ge $var2)
{
    Write-Host "$var1 est plus grand ou égal que $var2"
}
Else
{
    Write-host "$var1 est plus petit que $var2"
}
```

Dans ce second exemple, l'utilisateur initialise deux variables, puis PowerShell va tester si la première valeur est plus grande ou égale à la seconde. Si la condition est vraie, alors dans la console sera affiché un message indiquant que la première valeur est plus grande ou égale à la seconde. Si la condition est fausse, c'est le message se situant dans le bloc d'instruction de la clause Else qui sera affiché.

Enfin, pour finir sur les structures conditionnelles voici comment les améliorer avec l'instruction ElseIf. L'instruction ElseIf va nous permettre, si la condition précédente est fausse, de tester une autre condition. Ainsi, en utilisant une structure conditionnelle avec des ElseIf, nous ne nous limitons plus à une orientation binaire, mais nous augmentons

les possibles orientations d'exécution.

Exemple :

```
[int]$val = read-host 'Entrez une valeur : 1,2 ou 3'

If($val -eq 1)
{ Write-Host 'la valeur saisie est égale à 1 '}
ElseIf($val -eq 2)
{ Write-Host 'la valeur saisie est égale à 2 '}
ElseIf($val -eq 3)
{ Write-Host 'la valeur saisie est égale à 3 '}
Else
{Write-Host "la valeur saisie n'est pas égale à 1 ni à 2, ni à 3 "}
```

De cette manière, on aurait pu créer autant de ElseIf que voulu. Mais l'utilisation intensive des ElseIf est une solution viable mais un peu lourde. Le fait qu'il y ait autant de conditions que de blocs d'instruction, ne rend pas le code très souple, et l'on préférera s'orienter vers l'instruction Switch.

Switch

L'instruction Switch permet de remplacer avantageusement toute une série de If et de ElseIf. À la différence de l'instruction If qui, pour une expression donnée, va orienter la suite de l'exécution vers un des deux blocs d'instructions, l'instruction Switch va vous permettre d'orienter l'exécution vers plusieurs blocs d'instructions. Et ce, avec une seule expression. Ce qui lui confère une utilisation nettement plus souple. La syntaxe de Switch est la suivante :

```
Switch (<Expression>)
{
    <Valeur_1> {<instructions>}
    <Valeur_2> {<instructions>}
    <Valeur_3> {<instructions>}
    Default    {<instructions>}
}
```

La valeur « default » est facultative, son bloc d'instruction n'est exécuté uniquement dans le cas où l'expression ne correspond à aucune des valeurs du Switch.

Prenons pour exemple d'application, le cas basique où l'utilisateur doit saisir un nombre entre 1 et 5, et PowerShell détermine quel nombre a été saisi. Le code est le suivant :

```
$Nombre = Read-Host 'Entrez un nombre compris entre 1 et 5 '

Switch($Nombre)
{
    1 {Write-Host 'Vous avez saisi le nombre 1 '}
    2 {Write-Host 'Vous avez saisi le nombre 2 '}
    3 {Write-Host 'Vous avez saisi le nombre 3 '}
    4 {Write-Host 'Vous avez saisi le nombre 4 '}
    5 {Write-Host 'Vous avez saisi le nombre 5 '}
    Default {Write-Host "Le nombre saisi n'est pas compris
entre 1 et 5"}
}
```

L'instruction Switch accepte également les expressions régulières, pour cela il suffit de spécifier le paramètre -regex :

```
$chaine = Read-Host 'Entrez une chaine'

Switch -regex ($chaine)
{
    '[aeiouy]' {Write-Host 'La chaine saisie commence par une voyelle'}
    '^[^aeiouy]' {Write-Host 'La chaine saisie ne commence pas par une voyelle'}
}
```

-
-  Si plusieurs correspondances sont trouvées, chacune d'elle provoquera l'exécution du bloc d'instruction correspondant. Pour éviter cela, utilisez le mot clé break permettant d'exercer une sortie d'exécution.
-

Les fonctions

En PowerShell et comme dans de nombreux langages, une fonction est un ensemble d'instructions auquel on va donner un nom. Le principal intérêt des fonctions est que vous pouvez y faire référence à plusieurs reprises, sans avoir à ressaisir l'ensemble des instructions à chaque fois. Une fonction est constituée des éléments suivants :

- un nom ;
- un type de portée (facultatif) ;
- un ou plusieurs arguments (facultatifs) ;
- un bloc d'instruction.

En ce qui concerne le type de portée, nous aborderons cette notion plus tard dans ce chapitre. L'écriture d'une fonction nécessite la syntaxe suivante :

```
Function [<portée> :] <nom de fonction> (<argument>)  
{  
    param (<liste des paramètres>)  
    # bloc d'instructions  
}
```

Prenons par exemple la fonction suivante :

```
Function Bonjour  
{  
    $date = Get-Date  
    Write-Host "Bonjour, nous sommes le $date"  
}
```

Cette fonction est la plus basique qui soit. À chaque appel, elle affiche un message dans la console. Pour appeler une fonction il suffit tout simplement de taper son nom :

```
PS > bonjour  
Bonjour, nous sommes le 09/06/2009 17:07:09
```

Utilisation des arguments

Une notion importante dans l'utilisation de fonctions et de scripts, est le passage de valeurs. Pour ce faire, une technique consiste à utiliser les arguments. Les arguments sont les valeurs placées derrière le nom de la fonction lorsque celle-ci est appelée. Voici la syntaxe de l'appel d'une fonction avec plusieurs arguments :

```
<Nom de fonction> <Argument1> < argument2> <argumentN>
```

Imaginons que nous venions de créer une fonction qui affiche un message dans une boîte de dialogue. La question est, comment faire pour récupérer les arguments de façon à les insérer dans une boîte de dialogue ? Et bien la réponse est toute simple, lorsque nous passons des arguments à une fonction, tous se retrouvent stockés dans un tableau d'arguments appelé \$args. Et c'est ce tableau que nous allons réutiliser à l'intérieur de la fonction ou du script. \$args[0] correspond à votre premier argument, \$args[1] au second, etc.

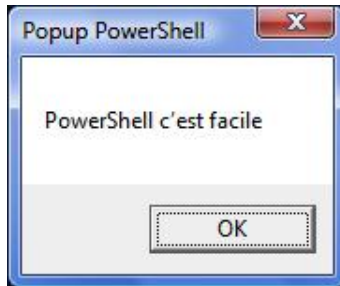
Prenons par exemple une fonction capable de créer une fenêtre pop-up et d'y insérer un titre et un texte :

```
Function Set-Popup
{
    $WshShell = New-Object -ComObject wscript.Shell
    $WshShell.Popup($args[0], 0, 'Popup PowerShell')
}
```

➤ Cette fonction fait appel à un objet COM du nom de wscript.shell. L'accès, la création et toutes autres manipulations sur les objets COM sont décrites dans le chapitre Objets COM.

Lorsque nous ferons appel à cette fonction avec un ou plusieurs arguments, seul le premier sera pris en compte et affiché dans une boîte de dialogue :

```
PS > Set-Popup "PowerShell c'est facile"
```



Affichage de l'argument dans la boîte de dialogue

Utilisation des paramètres

La deuxième façon de transmettre des variables à une fonction ou à un script, est d'utiliser les paramètres. La syntaxe d'appel d'une fonction avec un paramètre est la suivante :

```
<Nom de fonction> -<Paramètre> <Valeur du paramètre>
```

Pour ensuite que PowerShell les interprète, il suffit de spécifier au début de votre fonction ou script, les paramètres d'entrée grâce à l'instruction `param(<type du paramètre><nom du paramètre>)`.

Par exemple :

```
Function Set-Popup
{
    param([string]$message, [string]$titre)

    $WshShell = New-Object -ComObject wscript.Shell
    $WshShell.Popup($message,0,$titre)
}
```

Avec ce principe, contrairement aux arguments, l'ordre n'a aucune importance à partir du moment où l'on spécifie le nom du paramètre. Cela signifie que les deux expressions suivantes donneront le même résultat :

```
PS > Set-Popup -titre 'Mon titre' -message 'Bonjour'
```

```
PS > Set-Popup -message 'Bonjour' -titre 'Mon titre'
```

Si on ne souhaite pas utiliser les noms des paramètres quand on appelle la fonction, dans ce cas-là c'est leur position qui est prise en compte.

On peut également attribuer une valeur par défaut à un paramètre donné.

```
Function Set-Popup
{
    param([string]$message='Message...', [string]$titre='Titre')

    $WshShell = New-Object -ComObject wscript.Shell
    $WshShell.Popup($message, 0, $titre)
}
```

Ainsi, lors d'un appel de la fonction, si les valeurs des paramètres *Titre* et *Message* ne sont pas renseignés, alors l'exécution se fera avec les valeurs définies par défaut.

Exemple :

```
PS > Set-Popup
```



Boîte de dialogue avec les paramètres par défaut

Notez, que lors de la déclaration des paramètres, on peut utiliser l'instruction « `throw` » pour lancer une exception et informer l'utilisateur qu'il manque un paramètre. Vous pourrez le remarquer à travers de nombreux exemples à venir tout au long de cet ouvrage.

 PowerShell permet également l'appel des fonctions ou de scripts, en utilisant une partie d'un nom de paramètre. On peut raccourcir les noms des paramètres tant que l'on veut dans la mesure où il n'y a pas d'ambiguïté entre plusieurs noms de paramètres.

Par exemple :

```
PS > Set-Popup -t 'Mon titre' -m "PowerShell c'est facile"
```

Chose importante à noter, il n'est pas obligatoire d'indiquer le nom des paramètres (appel de la fonction comme s'il s'agit d'argument) pourvu que l'ordre dans lequel ils sont définis soit respecté, voir exemple ci-dessous.

```
PS > Set-Popup 'Mon titre' "PowerShell c'est facile"
```

1. Retourner une valeur

Une fonction retourne tout objet qui est émis. Par conséquent, il suffit d'insérer l'objet en fin de fonction ou script pour que son résultat soit transmis à l'appelant. Prenons l'exemple d'une fonction qui fait la moyenne de deux nombres.

```
Function moyenne
{
    param ([double]$nombre1, [double]$nombre2)
    ($nombre1 + $nombre2) /2
}
```

Pour affecter à une variable la valeur retournée par la fonction, il suffit de faire une affectation de variable, avec pour valeur l'appel à la fonction suivie de ses paramètres.

```
PS > $resultat = moyenne -nombre1 15 -nombre2 20
PS > $resultat
17,5
```

2. Les fonctions filtre

À la différence d'une fonction standard qui bloque l'exécution jusqu'à ce que toutes les informations en entrée aient été reçues, la « fonction filtre » qui s'utilise après un pipe, traite les données à mesure de leur réception (pour en permettre le filtrage). C'est-à-dire que le bloc d'instructions est exécuté pour chaque objet provenant du pipe. La syntaxe est la suivante :

```
Filter <nom du filtre>
{
    # Bloc d'instructions
}
```

Prenons pour exemple, la création d'un filtre qui ne retourne que les répertoires. La composition est la suivante :

```
Filter Filtre-Repertoire
{
    If($_.Mode -like "d*"){$_}
}
```

Ainsi, si ce filtre est appliqué à Get-ChildItem, il procédera à un traitement des objets passés par le pipe pour en filtrer les éléments correspondant à un répertoire :

```
PS > Get-ChildItem | Filtre-Repertoire
```

Répertoire :

Mode		LastWriteTime	Length	Name
----		-----	-----	-----
d----	10/2/2008	08:58		Repertoire1
d----	10/2/2008	13:46		Repertoire2

De façon à mieux structurer leur exécution, les fonctions filtre (comme les boucles Foreach) disposent de trois sections : Begin, Process et End.

Les sections Begin et End, sont respectivement exécutées de façon unique avant et après le bloc contenu dans Process, qui lui, peut être exécuté de une à plusieurs fois selon l'utilisation de la fonction.

Exemple :

```
Filter Filtre-fichier
{
    Begin
    {
        # Bloc d'instructions exécuté une seule fois au début de la fonction
        $taille = 0
    }

    Process
    {

        # Bloc d'instructions exécuté pour chaque objet passé depuis le pipe
        If($_.Mode -Like "-a*"){
            $_
            $taille += $_.length
        }

    }

    End
    {
        Write-host "`n La taille cumulée de tous les fichiers est de $taille octets"
    }
}
```

Le résultat obtenu est le suivant.

```
PS > Get-ChildItem | Filtre-Fichier

    Répertoire : D:\Scripts
Mode                LastWriteTime         Length Name
----                -
-a---          12/09/2009      17:53           2497 cesar.ps1
-a---          08/09/2009      12:10           1118 chaine_c1.txt
-a---          14/09/2009      23:32           452 chaine_c2.txt
-a---          08/09/2009      12:14           1118 chaine_c3.txt
-a---          14/09/2009      23:30              0 Essai.txt
-a---          10/09/2009      16:27           562 MonCertificat.cer
-a---          10/09/2009      17:25           576 MonCertificat2.cer
-a---          14/09/2009      23:40          1804 MonScript.ps1
-a---          14/09/2009      23:42           171 Script.txt
La taille cumulée de tous les fichiers est de 8298 octets
```

Création d'objets personnalisés

Comme nous venons de le voir précédemment, retourner une valeur en fin de fonction ou de script est très simple. Dans le cas de scripts conséquents ayant de nombreux résultats en retour, l'affichage des résultats sans un formatage particulier n'est pas toujours aisément interprétable car il ne garantit pas une uniformisation des résultats. Concrètement, qu'est-ce cela signifie ? Prenons l'exemple du script suivant qui pour un fichier donné retourne son nom, la date de création et la date du dernier accès.

```
Function Info-File
{
    param([string]$nom_fichier)
    $fichier = Get-Item $nom_fichier
    Write-Output "Nom : $($fichier.Name)"
    Write-Output "Date de création : $($fichier.CreationTime)"
    Write-Output "Dernier accès : $($fichier.LastAccessTime)"
}
```

Le résultat de cette fonction est une liste des différents fichiers avec les propriétés sollicitées.

```
PS > Info-File .\Abonnés.txt

Nom : Abonnés.txt
Date de création : 04/09/2009 18:17:00
Dernier accès : 04/09/2009 18:17:00
```

Bien que le résultat soit celui attendu, son interprétation et sa réutilisation n'est guère satisfaisante. Il est préférable d'utiliser un formalisme différent sous forme d'objet personnalisé. Un objet personnalisé est un objet sur lequel on va pouvoir insérer nos propres propriétés afin d'adopter un formalisme clair et réutilisable. En voici un exemple avec le script ci-dessous qui réalise le même traitement que le précédent.

```
Function Info-File
{
    param([string]$nom_fichier)
    $fichier = Get-Item $nom_fichier
    # Construction de l'objet qui contiendra tous les résultats
    $result = New-Object PSObject
    # Ajout de membres à notre objet:
    $result | Add-Member NoteProperty Nom $fichier.Name
    $result | Add-Member NoteProperty Date_Creation $fichier.CreationTime
    $result | Add-Member NoteProperty Dernier_Acces $fichier.LastAccessTime

    # Retour des résultats
    $result
}
```

Cette fois nous avons créé notre propre objet PowerShell (cf. chapitre Maîtrise du Shell - Objets PSBase et PSObject) auquel nous avons ajouté des propriétés grâce à la commande **add-member**. Chacune de ces propriétés se voit attribuer une valeur. Ainsi, lorsque l'objet est retourné, il prend la forme d'un tableau dans lequel chaque colonne est une propriété.

```
PS > Info-File .\Abonnés.txt

Nom                Date_Creation      Dernier_Acces
---                -
Abonnés.txt        04/09/2009 18:17:00  04/09/2009 18:17:00
```

Il est donc beaucoup simple de lire les informations et de réutiliser l'objet ultérieurement.

```
PS > $resultat = Info-File .\Abonnés.txt
PS > $resultat.Nom
Abonnés.txt

PS > $resultat.Dernier_Acces
04/09/2009 18:17:00
```

Autre exemple, prenons cette fois-ci l'exemple d'un script qui affiche pour un répertoire donné, le nombre et les noms des scripts dont la taille est comprise entre 0 et 50 octets, 50 et 500 octets, et plus de 500 octets.

```

#Information-taille.ps1
param([string]$chemin)

#Création des variables
$liste = Get-ChildItem $chemin
$Taille_0_50 = 0
$liste_0_50 = ''
$Taille_50_500 = 0
$liste_50_500 = ''
$Taille_500 = 0
$liste_500 = ''

#Boucle sur l'ensemble des éléments contenus dans la liste
foreach($selement in $liste){
    $type=$selement.gettype()
    $size=0
    if($type.Name -eq "FileInfo"){
        $size=$selement.length
        if($size -lt 50){
            $Taille_0_50++
            $liste_0_50 += "$selement;"
        }
        elseif(($size -ge 50) -and ($size -lt 500)){
            $Taille_50_500++
            $liste_50_500 += "$selement;"
        }
        elseif($size -ge 500){
            $Taille_500 ++
            $liste_500 += "$selement;"
        }
    }
}

# Construction de l'objet qui contiendra tous les résultats liés
# aux fichiers de taille 0 à 50 octets
$Result_0_50 = New-Object PSObject
# Ajout de membres à notre objet:
$Result_0_50 | Add-Member NoteProperty 'Taille (octets)' '0 - 50'
$Result_0_50 | Add-Member NoteProperty Nombre $Taille_0_50
$Result_0_50 | Add-Member NoteProperty Noms $liste_0_50

# Construction de l'objet qui contiendra tous les résultats liés
# aux fichiers de taille 50 à 500 octets
$Result_50_500 = New-Object PSObject
# Ajout de membres à notre objet:
$Result_50_500 | Add-Member NoteProperty 'Taille (octets)' '50 - 500'
$Result_50_500 | Add-Member NoteProperty Nombre $Taille_50_500
$Result_50_500 | Add-Member NoteProperty Noms $liste_50_500

# Construction de l'objet qui contiendra tous les résultats liés
# aux fichiers de taille 500 octets et plus
$Result_500 = New-Object PSObject
# Ajout de membres à notre objet:
$Result_500 | Add-Member NoteProperty 'Taille (octets)' '500+'
$Result_500 | Add-Member NoteProperty Nombre $Taille_500
$Result_500 | Add-Member NoteProperty Noms $liste_500

#Affichage des objets personnalisés
$Result_0_50
$Result_50_500
$Result_500

```

La construction des objets personnalisés ainsi que leur affichage sous forme d'un tableau permet d'obtenir un résultat clair et précis.

```
PS .\Information_taille.ps1 D:\Scripts
```

Taille (octets)	Nombre	Noms
-----	-----	----

0 - 50	2	Essai.txt;Script.txt;
50 - 500	1	chaine_c2.txt;
500+	6	cesar.ps1;chaine_c1.txt;chaine_...

La portée des variables

La portée d'une variable détermine la visibilité d'une variable dans PowerShell, cette notion de portée est très importante, puisqu'elle garantit une indépendance des variables, et évite ainsi les probables interférences de valeurs.

Imaginez un instant que vous exécutez un script et qu'une fois le script terminé vous souhaitiez vérifier une valeur en tapant son nom dans la console, aie ! Ce n'est pas possible, la variable n'est pas disponible, c'est tout simplement dû à la portée des variables.

PowerShell utilise la notion de portée Parent et de portée Enfant. Une portée Enfant étant une portée créée à l'intérieur d'une portée Parent. C'est-à-dire qu'à chaque fois que nous exécutons une fonction, un script, ou un bloc d'instructions, une nouvelle portée est créée. Et sauf spécification contraire de votre part, PowerShell définit qu'une variable peut être lue dans sa propre portée, ainsi que dans les portées enfants. Mais elle ne peut être modifiée que dans la portée où elle a été créée. De plus, les portées parentes ne peuvent ni lire, ni modifier les variables définies dans leurs portées enfants.

Le schéma ci-dessous illustre l'accessibilité d'une variable à travers une portée enfant.



Illustration de l'accessibilité d'une variable

Admettons qu'une variable `$valeur` soit initialisée à 0 dans la console. Puis, créons la fonction « lire » suivante :

```
function Lire
{
    $valeur
}
```

Jusque-là tout va bien, nous pouvons lire la variable `$valeur` à chaque fois que nous appelons la fonction « lire ». Nous sommes bien dans le cas d'une variable créée dans une portée parent, accessible en lecture dans une portée enfant.

Maintenant, si nous créons la fonction « ajouter » qui permet d'incrémenter de 1 la variable `$valeur` à chaque fois que la fonction est appelée :

```
function Ajouter
{
    $valeur++
}
```

Si vous avez bien suivi jusque-là, vous savez qu'une fois la fonction terminée, votre variable ne sera pas incrémentée, la variable `$valeur` sera toujours égale à 0. Tout simplement parce que nous n'avons rien spécifié dans les portées de variables, et que par conséquent, nous ne sommes pas autorisé à modifier cette valeur dans une portée enfant.

```
PS > $valeur = 0
PS > Ajouter
PS > $valeur
0
```

Pour remédier à cela, il faut adapter la portée des variables selon trois types :

- la portée globale,
- la portée locale,
- la portée script.

La portée globale (global) :

La portée globale est celle appliquée au démarrage de PowerShell, c'est-à-dire que si au démarrage nous initialisons une variable, par défaut sa portée sera globale.

Les variables de portée globale, sont accessibles en lecture dans une portée enfant sans spécifier quoi que ce soit.

Cependant, pour pouvoir modifier une variable de portée globale depuis une portée enfant, il faut impérativement spécifier un libellé « global : » avant le nom de variable.

Par exemple :

```
function Ajouter
{
    $global:valeur++
}
```

Ainsi, lors du retour dans la portée Parent, la variable \$valeur a bien été modifiée.

```
PS > $valeur = 0
PS > Ajouter
PS > $valeur
1
```

La portée locale (local) :

La portée locale c'est la portée dans laquelle nous nous trouvons à un instant T (portée actuelle).

Une nouvelle portée locale est créée chaque fois que nous exécutons une fonction, un script ou un bloc d'instructions.

La portée locale répond aux règles de base, à savoir qu'une variable créée dans une portée parent, ne peut pas être modifiée dans une portée enfant.

La portée script :

La portée script est, comme son nom l'indique, une portée créée durant le temps d'exécution du script, et cesse d'exister une fois le script terminé. À l'instar de la console, un script peut être amené à créer plusieurs portées enfants, elles-mêmes susceptibles de créer des variables. La portée script permet alors d'accéder à ces variables, mais à l'extérieur de la fonction. Exemple, prenons le script suivant :

```
#script1.ps1
#Script sur le type de portée "script"

Function Nouvelle_valeur
{
    $valeur = 10
}

$valeur = 0           #initialisation de la variable
Write-Host "La valeur d'origine est : $valeur"
Nouvelle_valeur       #Appel de la fonction
Write-Host "La nouvelle valeur est : $valeur"
```

Dans ce script, nous initialisons une variable à 0, puis nous appelons une fonction qui va assigner la valeur 10 à notre variable, puis pour finir nous affichons la variable.

Si aucune portée n'est spécifiée, en exécutant le script, on s'aperçoit que la variable n'a pas été modifiée :

```
PS > ./Script1.ps1

Valeur d'origine : 0
Nouvelle valeur : 0
```

Si maintenant, nous intégrons le libellé « script : » devant le nom de la variable dans la fonction, la variable est alors identifiée comme faisant partie de la portée script :

```
#script1.ps1
#Script sur le type de portée "script"

Function Nouvelle_valeur
{
    $script:valeur = 10
}

$valeur = 0                #initialisation de la variable
Write-Host "Valeur d'origine : $valeur"
Nouvelle_valeur            #Appel de la fonction
Write-Host "Nouvelle valeur : $valeur"
```

Le résultat ne sera donc pas le même. Cette fois, c'est bien la variable créée dans la portée script qui va être modifiée.

```
PS > ./Script1.ps1

Valeur d'origine : 0
Nouvelle valeur : 10
```

À noter aussi que nous pouvons également utiliser le libellé « private : » lors de l'affectation de la variable dans une fonction. Ceci aura pour effet de faire prendre à notre variable une valeur uniquement durant la période de vie de la fonction.

Le DotSourcing

On appelle « DotSourcing », le procédé qui consiste à placer un point et un espace avant le nom d'un script, ou d'un bloc d'instructions. Cette technique permet d'exécuter le contenu du script dans l'étendue courante. De cette manière, toute variable ou fonction se retrouve par conséquent réutilisable, et ce, durant toute la vie de l'étendue. Prenons par exemple le script suivant qui ne contient rien d'autre que des fonctions.

```
# fonctions.ps1

Function Reveil
{
    Write-Host 'Bonjour et bon réveil '
}

Function Liste-Temp
{
    Get-ChildItem -Path c:\temp
}

Function CPU-Time
{
    Get-Process | Where-Object {$_.CPU -gt 500}
}
```

En exécutant ce script sans aucune spécification, aucune des trois fonctions ne sera réutilisable, tout simplement parce que l'étendue créée par l'ouverture du script s'est terminée avec lui.

```
PS > ./fonctions.ps1
PS > Reveil

Le terme « reveil » n'est pas reconnu en tant qu'applet
de commande, fonction, programme exécutable ou fichier de script.
Vérifiez le terme et réessayez. Au niveau de ligne :
1 Caractère : 6 + reveil <<<<
```

Cependant, si maintenant nous appelons ce script par la méthode du DotSourcing, c'est-à-dire avec un point devant et un espace, les méthodes seront encore disponibles même après exécution du script.

```
PS > . ./fonctions.ps1
PS > Reveil

Bonjour et bon réveil
```


Les fonctions avancées

Les fonctions avancées (*advanced functions*) apportent de nouvelles fonctionnalités très intéressantes de PowerShell v2. Ces dernières permettent un fonctionnement très similaire aux commandelettes, et ce de façon simple et rapide.

Dans la version 1.0 de PowerShell, la seule façon de procéder à la création d'une commandelette était de compiler son propre code développé en C# ou Visual Basic .NET et de le faire prendre en compte dans Windows PowerShell par le biais d'un snap-in. La version 2 de PowerShell, simplifie cette façon de procéder pour permettre à l'utilisateur moyen de créer ses propres « commandelettes » directement depuis la console. Désormais il n'est plus nécessaire d'avoir des compétences de développeur pour y arriver.

Les fonctions avancées nécessitent le mot clé « CmdletBinding » pour les identifier en tant que tel. La syntaxe d'utilisation est la suivante :

```
function <nom de fonction> (<argument>)
{
    [CmdletBinding()]
    param (<liste des paramètres>)
    # Bloc d'instructions
}
```

Exemple :

Fonction avancée du nom de Map-Drive qui permet de connecter un lecteur réseau.

```
function Map-Drive
{
    [CmdletBinding()]
    Param([string]$Lettre, [string]$Partage)
    $obj = New-Object -Com Wscript.Network
    $obj.MapNetworkDrive("$Lettre:", $Partage)
}
```

Toutefois, nous pouvons observer que ces fonctions avancées ne sont pas listées lors d'un **Get-Command -Commandtype cmdlet**. En réalité, il existe des différences entre une commandelette « classique » et une fonction avancée. La principale est qu'une commandelette éditée dans la console est considérée comme n'étant pas réellement du même type. C'est la raison pour laquelle la commande **Get-Command -Commandtype cmdlet** ne retourne pas les fonctions avancées, mais uniquement celles compilées en C# ou VB .NET.

Pour obtenir les fonctions avancées que nous avons créées, il nous faut saisir la ligne suivante :

```
PS > Get-Command -Commandtype function
```

Lorsque nous éditons une fonction avancée, nous pouvons lui définir des attributs qui vont agir sur son comportement. En voici la liste non exhaustive :

Attributs	Description
SupportsShouldProcess	Cet attribut indique que la fonction avancée permet les appels à la méthode ShouldProcess . La méthode ShouldProcess informe l'utilisateur sur le résultat de l'action avant que cela ne modifie le système. C'est-à-dire que lorsque cet attribut est spécifié, le paramètre WhatIf est activé.
DefaultParameterSet <paramètre>	Cet attribut spécifie le nom du ou des paramètres que la fonction doit utiliser lorsqu'elle ne sait pas déterminer lequel elle doit prendre.
ConfirmImpact <Valeur>	Cet attribut permet de définir à quel moment l'action de la fonction doit être confirmée par un appel à la méthode ShouldProcess . Cette dernière est appelée uniquement lorsque la valeur associée au paramètre de ConfirmImpact (par défaut, il s'agit de la valeur medium) est supérieure ou égale à la valeur de la variable \$ConfirmPreference . Les valeurs possibles sont : low, medium, high.

Snapin <Nom du Snap-in>

Cet attribut spécifie le nom du composant logiciel enfichable qui est utilisé pour faire fonctionner la fonction.

Exemple :

```
Function Nom-Verbe
{
    [CmdletBinding(SupportsShouldProcess=$true, ConfirmImpact="medium")]
    Param ([string]$Parametre)
    Begin
    {
        # Bloc d'instructions
    }
    Process
    {
        # Bloc d'instructions
    }
    End
    {
        # Bloc d'instructions
    }
}
```

Le gros avantage que présente une fonction avancée vis-à-vis d'une fonction classique, est qu'elle dispose de plus de contrôle sur ses paramètres, et ce grâce à l'utilisation d'attributs et d'arguments (les arguments permettant de définir les attributs). Par exemple, pour spécifier que la valeur passée en attribut est de type string et qu'elle provient d'un pipeline, il suffit de le spécifier l'attribut parameter avec pour argument ValueFromPipeline=\$true :

```
function Get-Result
{
    [CmdletBinding()]
    Param(
        [parameter(ValueFromPipeline=$true)]$valeur
    )
    write-host "le résultat du pipe est : $valeur"
}
```

Ou encore, si cette valeur est nécessaire au fonctionnement de la fonction, alors, en spécifiant l'argument Mandatory à ce même attribut parameter, celui-ci est rendu obligatoire pour l'exécution du script.

```
function Get-Result
{
    [CmdletBinding()]
    Param(
        [parameter(Mandatory=$true, ValueFromPipeline=$true)]$valeur
    )
    write-host "le resultat du pipe est : $valeur"
}
```

L'attribut le plus utilisé se nomme parameter (voir ci-dessus). C'est lui, qui via les arguments qui lui sont données, va permettre d'agir sur le comportement du paramètre souhaité. L'ensemble des arguments utilisables pour l'attribut parameter sont listés ci-dessous.

Argument de l'attribut "parameter"	Description
Mandatory	L'argument Mandatory indique que le paramètre est obligatoire si la valeur est égale à \$true. <i>Syntaxe :</i> Param([parameter(Mandatory=\$true)]\$valeur)
Position	L'argument Position spécifie la position du paramètre lors de l'appel à la fonction ou au script. <i>Syntaxe :</i>

	Param([parameter(Position=0)]\$valeur)
ParameterSetName	L'argument ParameterSetName spécifie le jeu de paramètres auquel un paramètre appartient. <i>Syntaxe :</i> Param([parameter(ParameterSetName='chiffre')]\$valeur)
ValueFromPipeline	L'argument ValueFromPipeline spécifie que le paramètre accepte les entrée de pipe, si la valeur est égale à \$true. <i>Syntaxe :</i> Param([parameter(ValueFromPipeline=\$true)]\$valeur)
ValueFromPipelineBy PropertyName	L'argument valueFromPipelineByPropertyName spécifie que le paramètre accepte l'entrée provenant d'une propriété d'un objet de pipeline. Cela signifie par exemple, que si la fonction comporte un paramètre nommé « valeur » et que l'objet redirigé comporte une propriété du même nom (« valeur »), et bien le paramètre en question se voit acquérir le contenu de la propriété « valeur » de l'objet transmis. <i>Syntaxe :</i> Param([parameter(ValueFromPipeline=\$true)]\$valeur)
ValueFromRemaining Arguments	Au contraire de l'argument précédent, ValueFromRemainingArguments spécifie que le paramètre accepte les arguments de la fonction. <i>Syntaxe :</i> Param([parameter(ValueFromRemainingArguments =\$true)]\$valeur)
HelpMessage	L'argument HelpMessage permet d'indiquer une description du contenu du paramètre. <i>Syntaxe :</i> Param([parameter(HelpMessage="Un chiffre entre 0 et 9999")]\$valeur)

Il existe bien entendu d'autres attributs que parameter. Ces derniers, qui sont listés ci-dessous, agissent non pas sur le comportement du paramètre mais sur son contenu. En voici la liste :

Argument de l'attribut "parameter"	Description
Alias	Permet d'indiquer un alias sur le paramètre. <i>Syntaxe :</i> Param([alias("CN")]\$valeur)
AllowNull	Permet d'indiquer que l'on autorise une valeur nulle comme valeur de paramètre. <i>Syntaxe :</i> Param([AllowNull()])\$valeur)
AllowEmptyString	Permet d'indiquer que l'on autorise une chaîne vide comme valeur de paramètre. <i>Syntaxe :</i> Param([AllowEmptyString()])\$valeur)
AllowEmptyCollection	Permet d'indiquer que l'on autorise une collection vide comme valeur de paramètre. <i>Syntaxe :</i> Param([AllowEmptyCollection()])\$valeur)

ValidateCount	Permet d'indiquer le nombre minimal et le nombre maximal d'arguments que l'on peut fournir au paramètre en question. <i>Syntaxe :</i> Param([ValidateCount(1,3)]\$valeur)
ValidateLength	Permet de définir la longueur minimale et la longueur maximale de la valeur passée en paramètre (nombre de caractères par exemple). <i>Syntaxe :</i> Param([ValidateLength(1,5)]\$valeur)
ValidatePattern	Permet de définir la valeur passée en paramètre selon un modèle établi avec les expressions régulières. <i>Syntaxe :</i> Param([ValidatePattern("[A*]")\$chaine)
ValidateRange	Permet de définir une gamme de valeur (valeur min et valeur max). <i>Syntaxe :</i> Param([ValidateRange(0,20)]\$valeur)
ValidateScript	Permet de spécifier qu'un bloc de script est utilisé pour valider la valeur fournie en paramètre. Pour que la valeur soit acceptée, le bloc de script doit retourner la valeur \$true. <i>Syntaxe :</i> Param([ValidateScript({\$_ -le 99 })]\$valeur)
ValidateSet	Permet de spécifier une ou plusieurs valeurs auxquelles la valeur du paramètre doit correspondre. <i>Syntaxe :</i> Param([ValidateSet("Rouge", "Bleu", "Vert")]\$couleur)
ValidateNotNull	Permet de spécifier que la valeur passée en argument ne doit pas être null. <i>Syntaxe :</i> Param([ValidateNotNull()])\$valeur)
ValidateNotNullOrEmpty	Permet de spécifier que la valeur passée en argument ne doit pas être null ou vide. <i>Syntaxe :</i> Param([ValidateNotNullOrEmpty])\$valeur)

Personnaliser PowerShell en modifiant son profil

Vous connaissez certainement déjà la notion de profil car il en est question depuis longtemps dans Windows avec, entre autres, le fameux «profil Windows» (qui peut être local ou itinérant), ainsi que le profil Outlook. Un profil est simplement un fichier (ou un ensemble de fichiers) qui contient les préférences de l'utilisateur et qui lui permet de personnaliser son environnement.

Il faudra désormais composer avec des profils supplémentaires, ceux de PowerShell. Et ils peuvent être nombreux car il en existe quatre différents.

Il faut tout d'abord distinguer deux sortes de profils :

- Les profils utilisateurs (au nombre de deux) qui s'appliquent à l'utilisateur courant.
- Les profils machines (au nombre de deux également) qui s'appliquent aux utilisateurs d'une machine en particulier.

Une autre notion qu'il faut connaître avec PowerShell est la notion de «Shell» ou «environnement» en français. La console installée d'origine avec PowerShell constitue un environnement. Vous n'êtes pas sans savoir que Microsoft Exchange 2007 (plate-forme d'entreprise de messagerie Microsoft) ainsi que System Center Operation Manager 2007 (anciennement MOM (*Microsoft Operation Manager*) est la solution de supervision des systèmes) et tous les autres produits de la gamme Microsoft System Center parus depuis 2009, possèdent déjà ou posséderont leur propre console PowerShell ; il est là aussi question de nouveaux environnements. Microsoft offre donc, en toute logique, la possibilité de créer un profil PowerShell propre à chaque environnement.

1. Profils utilisateurs

Si vous êtes plusieurs administrateurs systèmes dans votre société à utiliser PowerShell, vous aurez certainement envie que chacun de vous puisse personnaliser son environnement de travail, et ce sans modifier celui de son voisin. Dans ce cas, ce type de profil est fait pour vous.

Il existe deux profils utilisateurs portant chacun un nom distinct :

- %UserProfile%\Mes documents\WindowsPowerShell\profile.ps1
- %UserProfile%\Mes documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1

Le premier profil est un profil commun à tous les environnements alors que le second est propre à l'environnement PowerShell installé par défaut. En d'autres termes, si vous créez un fichier **profile.ps1**, toutes les modifications faites dans celui-ci seront valables aussi bien dans la console Exchange que dans la console SCOM, ainsi que dans la console par défaut.



C'est parce que l'identifiant de l'environnement PowerShell installé par défaut se nomme « Microsoft.PowerShell » que le nom du profil commence ainsi. Pour le vérifier, tapez la commande suivante :
Get-Item variable:ShellId



Certaines consoles et notamment PowerShell ISE, peuvent prendre en compte leur propre profil utilisateur. Exemple du profil PowerShell ISE : %UserProfile%\Mes documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1

2. Profils machines

Tous les changements que vous pourrez apporter à ces profils seront effectifs uniquement sur un ordinateur mais ils s'appliqueront à tous les utilisateurs.

Il existe deux profils machines portant chacun un nom distinct :

- %windir%\system32\WindowsPowerShell\v1.0\profile.ps1

- %windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1

Pour la plate-forme Windows 64 bits, l'emplacement de ces fichiers est différent :

- %windir%\syswow64\WindowsPowerShell\v1.0\profile.ps1
- %windir%\syswow64\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1

Le principe est le même que pour les profils utilisateurs, à savoir que le fichier **profile.ps1** s'appliquera à tous les environnements installés sur la machine et à tous les utilisateurs, tandis que le second sera spécifique à l'environnement **Microsoft.PowerShell**.



Il est préférable de manipuler en priorité les profils utilisateurs plutôt que les profils machines car les premiers peuvent vous suivre si vous utilisez les profils Windows itinérants ou si avez mis en place une stratégie de groupe qui redirige votre répertoire Mes documents vers un partage réseau. Si vous vous trouvez dans ce dernier cas, et que vous utilisez PowerShell sur un serveur, n'oubliez pas de désactiver la configuration de sécurité renforcée d'Internet Explorer. Sans quoi en fonction de votre stratégie d'exécution de script courante, PowerShell peut vous empêcher d'exécuter votre profil.



Tout comme pour le profil utilisateur, certaines consoles comme PowerShell ISE peuvent prendre en compte leur propre profil machine. Exemple du profil machine PowerShell ISE : %windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1 et %windir%\syswow64\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1

3. Ordre d'application des profils

L'ordre d'application des profils est important, PowerShell les applique dans cet ordre :

- %windir%\system32\WindowsPowerShell\v1.0\profile.ps1
- %windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
- %UserProfile%\Mes documents\WindowsPowerShell\profile.ps1
- %UserProfile%\Mes documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1

Comme d'habitude ce sont les paramètres les plus proches de l'utilisateur qui sont prioritaires et donc qui s'appliquent en dernier. Par exemple, si vous définissez plusieurs fois la même variable dans vos profils, la dernière définition qui s'applique aura le dernier mot.

4. Création du profil

Par défaut, aucun profil n'est créé. La méthode la plus simple pour créer son profil consiste à s'appuyer sur la variable prédéfinie **\$profile**. Cette variable contient le chemin complet vers votre profil utilisateur de l'environnement par défaut **Microsoft.PowerShell**, et ce même si vous ne l'avez pas encore créé.

Voyons ce que contient **\$profile** :

```
PS > $profile
C:\Users\Arnaud\Documents\WindowsPowerShell\
Microsoft.PowerShell_profile.ps1
```

Pour créer votre profil, tapez la commande :

```
PS > New-Item -Path $profile -ItemType file -Force
```

Félicitations, votre profil est maintenant créé mais il ne fait que zéro octet car il est vide. Pour le modifier avec le bloc-notes, tapez la commande suivante :

```
PS > notepad $profile
```

Vous êtes maintenant prêt à personnaliser votre environnement préféré. Vous pourriez par exemple changer la couleur de fond de la fenêtre, sa taille, la couleur des caractères, ajouter de nouveaux alias, ou de nouvelles fonctions, etc.

Voici par exemple le contenu de notre profil du moment :

```
# profile.ps1 version 0.7
# Définition de l'alias Out-Clipboard pour envoyer un flux
# dans le presse-papier.
#
Set-Alias -name Out-Clipboard -value 'c:\windows\system32\clip.exe'
Set-alias -name grep -value select-string

# Définition des fonctions
Function cd.. {cd ..}

# Modification des variables de preference
$VerbosePreference = 'continue' # par défaut "silentlycontinue"
$DebugPreference = 'continue'
$WarningPreference = 'continue'

# Message d'accueil personnalisé
#
$UserType = 'Utilisateur'
$CurrentUser = [System.Security.Principal.WindowsIdentity]::GetCurrent()
$principal =
    new-object System.Security.principal.windowsprincipal($CurrentUser)
if ($principal.IsInRole('Administrateurs'))
{
    $UserType = 'Administrateur'
    $host.ui.RawUI.BackgroundColor = 'DarkMagenta'
    Clear-Host
}
else
{
    $host.ui.RawUI.BackgroundColor = 'DarkMagenta'
    Clear-Host
}

Write-Host '+-----+'
Write-Host "+- Bonjour $($CurrentUser.Name).split('\')[1])"
Write-Host "+- Vous êtes connecté en tant que : $UserType"
Write-Host '+-----+'

# Modification de la couleur du prompt en jaune, remplacement
# du Prompt par PS > et affichage du chemin courant dans la barre de titre
# de la fenetre de la console
function prompt
{
    Write-Host ('PS ' + '>') -newline -fore yellow
    $host.ui.RawUI.Set_windowtitle("$(get-location) ($UserType)")
    return ' '
}
```

Nous allons voir dans la partie suivante, un éventail de ce qu'il est possible de faire pour personnaliser sa fenêtre PowerShell.

5. Personnalisation de l'environnement

Tout ce que nous allons voir maintenant est fait pour être inclus dans votre profil. À vous de choisir quel sera le fichier de profil le plus approprié à votre besoin.

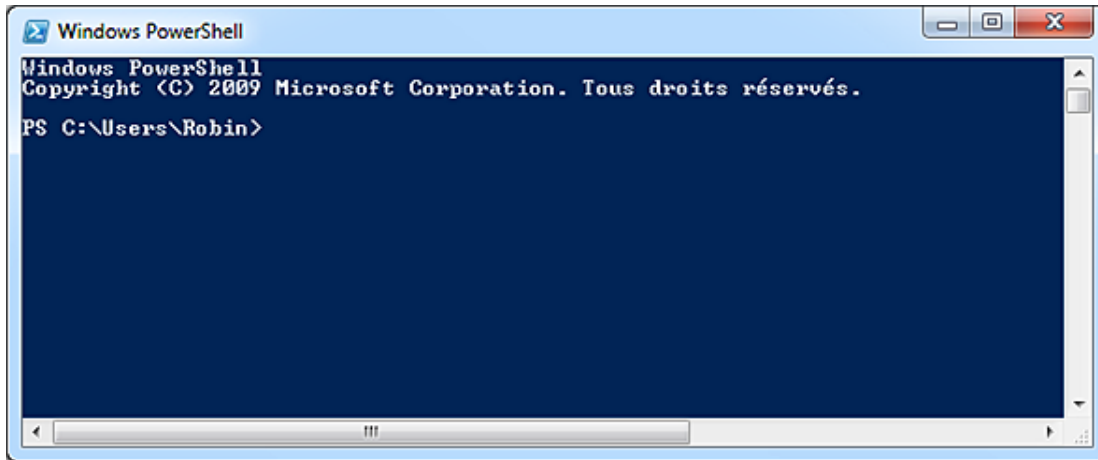
a. Modification du prompt

Le prompt ou invite est l'ensemble des caractères qui indique que l'ordinateur est prêt à recevoir une saisie au clavier.

Par défaut il est de la forme suivante : `PS CHEMIN_EN_COURS>`

Vous vous trouvez, au démarrage de PowerShell, dans le répertoire racine de votre profil utilisateur Windows (sous Windows 7 et Vista : `C:\Users\NomDuProfil`, sous Windows XP : `C:\Documents and Settings\NomDuProfil`).

Voici ce que donne sous Windows 7 le prompt par défaut :



Affichage du prompt par défaut

Pour le changer, il suffit de modifier la fonction **Prompt** intrinsèque à PowerShell. Voyons d'abord ce qu'elle contient dans sa configuration d'origine. Tapez la commande `Get-Content function:prompt`. Voici le résultat obtenu avec PowerShell v1 :

```
PS > Get-Content function:prompt
'PS ' + $(Get-Location) + $(if ($nestedpromptlevel -ge 1) { '>' }) + '> '
```

Et voici celui obtenu avec PowerShell v2 :

```
PS > Get-Content function:prompt
$(if (test-path variable:/PSDebugContext) { '[DBG]: ' } else { '' }) `
+ 'PS ' + $(Get-Location) + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
```

La fonction **Prompt** par défaut peut vous sembler un peu barbare de prime abord mais en la regardant de plus près on peut comprendre les choses suivantes :

- Elle concatène quatre chaînes de caractères séparées par l'opérateur d'addition « + ».
- `$(Get-Location)` retourne le chemin courant.
- `$nestedpromptlevel` indique si nous nous trouvons dans un environnement imbriqué ou non (voir chapitre Gestion des erreurs et débogage, section Le débogage - Les points d'arrêts (break points)). Si cette variable contient un nombre supérieur à zéro, nous nous trouvons dans un environnement imbriqué alors dans ce cas on ajoute au prompt un caractère « > » supplémentaire.
- Enfin on ajoute au prompt le caractère final « > » suivi d'un espace pour que la saisie ne soit pas accolée au prompt.
- À noter que dans PowerShell v2, un test sur les conditions de débogage est effectué en début de fonction (cf. chapitre Gestion des erreurs et débogage pour connaître la signification de ce test).

Il est très facile de redéfinir cette fonction, ainsi nous pourrions par exemple décider de supprimer du prompt le chemin en cours car souvent à cause de cela, le prompt est infiniment long lorsque l'on explore des arborescences où de nombreux répertoires sont imbriqués. Néanmoins, pour ne pas se priver de cette information intéressante, nous allons l'afficher dans le titre de la fenêtre, à la place de l'habituel titre « Windows PowerShell ».

```
function prompt
{
```



```
'PS > '  
$host.ui.RawUI.set_windowtitle($(get-location))  
}
```

Vous remarquerez que le titre de la fenêtre est rafraîchi chaque fois que nous changeons de répertoire courant. La réalité est un peu différente car la fonction **Prompt** est en fait réévaluée chaque fois que PowerShell nous redonne la main pour saisir une nouvelle ligne de commandes.

\$host est l'objet qui correspond à notre environnement. Il possède un grand nombre de propriétés et méthodes qui peuvent servir à personnaliser notre fenêtre PowerShell.

Un prompt haut en couleur

Pour donner une petite touche sympathique à notre invite, nous pouvons lui ajouter un peu de couleur, comme ceci :

```
function prompt  
{  
    Write-Host ('PS ' + $(get-location) + '>') `   
-NoNewLine -ForegroundColor yellow  
    ' '  
}
```

En procédant de la sorte, nous affichons une chaîne de caractères en couleur avec la commandelette **Write-Host**, à laquelle nous disons de ne pas retourner à la ligne avec le commutateur **-NoNewLine**. Puis nous redéfinissons notre invite à sa plus simple expression : un espace. Il est impératif que la fonction **prompt** renvoie une chaîne de caractères, sans quoi le prompt par défaut « PS> » apparaît. Au lieu d'écrire « ' ' » dans la fonction, ce qui peut paraître un peu bizarre, nous aurions pu écrire **return ' '**. Pour plus d'informations concernant le retour des fonctions, veuillez vous référer au chapitre Fondamentaux - Les fonctions.

Un prompt toujours à l'heure

Vous pourriez peut-être avoir envie d'afficher la date et l'heure à la place du chemin courant ?

Rien de plus simple, essayons cela :

```
function prompt  
{  
    Write-Host ('PS ' + $(get-date) + '>') -NoNewLine -Foreg yellow  
    return ' '  
}  
  
PS 09/18/2009 23:45:46>
```



Vous pouvez faire toute sorte de choses dans la fonction **Prompt**, mais retenez ceci : votre fonction doit toujours retourner une valeur de type **String**, sans quoi PowerShell affichera le prompt par défaut "PS>" ; pour plus de lisibilité essayez de limiter votre prompt à une seule ligne, la plus courte de préférence ; à chaque retour au prompt, autrement dit à la fin de chaque commande, PowerShell réévalue la fonction **Prompt**. Essayez donc de ne pas faire trop de choses compliquées dans votre fonction, ce qui pourrait avoir comme conséquence un certain ralentissement du système.

b. Modification de la taille de la fenêtre

Vous pouvez agir sur la fenêtre de la console pour en modifier sa taille, sa couleur, son titre, sa position, etc.

PowerShell vous permet d'agir sur la console à travers l'objet **host.ui.RawUI**. Listons ses propriétés pour voir celles sur lesquelles nous pouvons agir :

```
PS > $host.UI.RawUI  
  
ForegroundColor      : DarkYellow  
BackgroundColor      : DarkMagenta  
CursorPosition      : 0,2999  
WindowPosition      : 0,2948  
CursorSize          : 25  
BufferSize           : 140,3000  
WindowSize           : 140,52
```

```
MaxWindowSize      : 140,81
MaxPhysicalWindowSize : 182,81
KeyAvailable       : False
WindowTitle        : www.PowerShell-Scripting.com : Utilisateur
```

Si nous voulons ajuster horizontalement notre fenêtre il va nous falloir agir à la fois sur la taille de celle-ci mais également sur la taille de la mémoire tampon (le buffer) associée. Cela se fait ainsi :

```
PS > $buff = $host.ui.RawUI.BufferSize # init. de la variable $buff
PS > $buff.width = 150                 # déf. du nb. de car. par ligne
PS > $buff.Height = 3000                # déf. du nb. de lignes verticales
PS > $host.ui.RawUI.BufferSize = $buff
PS > $taille = $host.ui.RawUI.WindowSize # on initialise la variable
PS > $taille.Width = $buff.width        # nb. de caractères à l'horizontal
PS > $taille.Height = 60                # nombre de lignes verticales
PS > $host.ui.RawUI.WindowSize = $taille
```



La taille de la mémoire tampon et celle de la fenêtre doivent être rigoureusement identiques si vous ne voulez pas avoir d'ascenseur horizontal.

c. Modification des couleurs

Vous avez le loisir de choisir les couleurs de votre environnement préféré, et ce aussi bien pour les caractères que pour la couleur de fond de la fenêtre.

Voici la liste des couleurs possibles :

Black	Blue	Cyan	DarkBlue
DarkCyan	DarkGray	DarkGreen	DarkMagenta
DarkRed	DarkYellow	Gray	Green
Magenta	Red	White	Yellow

Pour les affecter, faites comme ceci :

```
PS > $host.ui.RawUI.ForegroundColor = 'White' # Couleur du texte
PS > $host.ui.RawUI.BackgroundColor = 'Black' # Couleur du fond
```



Lorsque nous changeons la couleur de fond de la fenêtre avec **\$host.ui.RawUI.BackgroundColor**, il faut que nous fassions ensuite un **Clear-Host** ou **cls**. Si vous ne le faites pas, la couleur de fond ne s'appliquera qu'aux nouveaux caractères ; ce qui n'est pas forcément du plus bel effet. Vous pouvez aussi affecter des couleurs différentes que celles par défaut aux messages d'erreur et de débogage. Pour les consulter, tapez **\$host.privatedata**.

Voici la liste des propriétés et des couleurs par défaut :

```
PS > $host.privatedata
ErrorForegroundColor      : Red
ErrorBackgroundColor      : Black
WarningForegroundColor    : Yellow
WarningBackgroundColor    : Black
DebugForegroundColor      : Yellow
DebugBackgroundColor      : Black
VerboseForegroundColor    : Yellow
VerboseBackgroundColor    : Black
ProgressForegroundColor   : Yellow
ProgressBackgroundColor   : DarkCyan
```

d. Modification du titre de la fenêtre

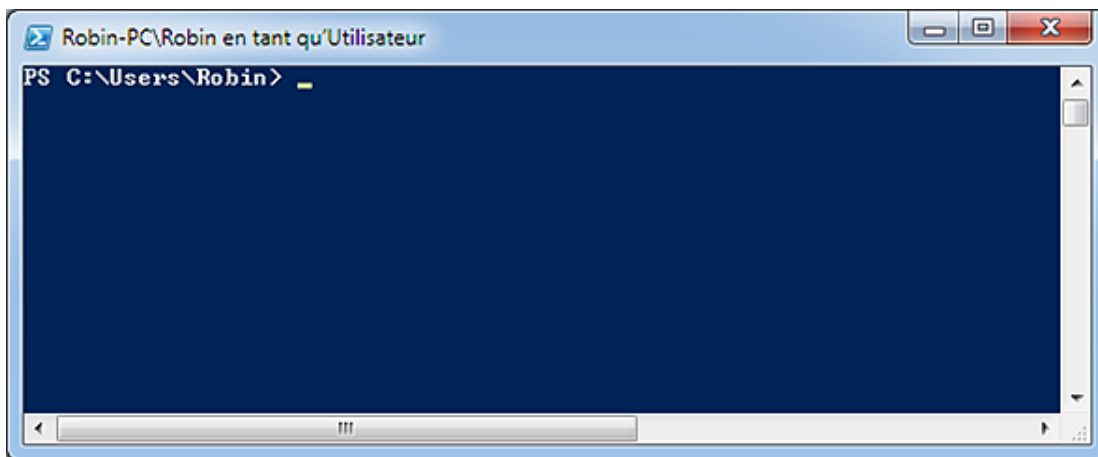
Le titre de la fenêtre se modifie grâce à la propriété **WindowTitle**, comme cela :

```
PS > $host.ui.RawUI.WindowTitle = 'www.PowerShell-Scripting.com'
```

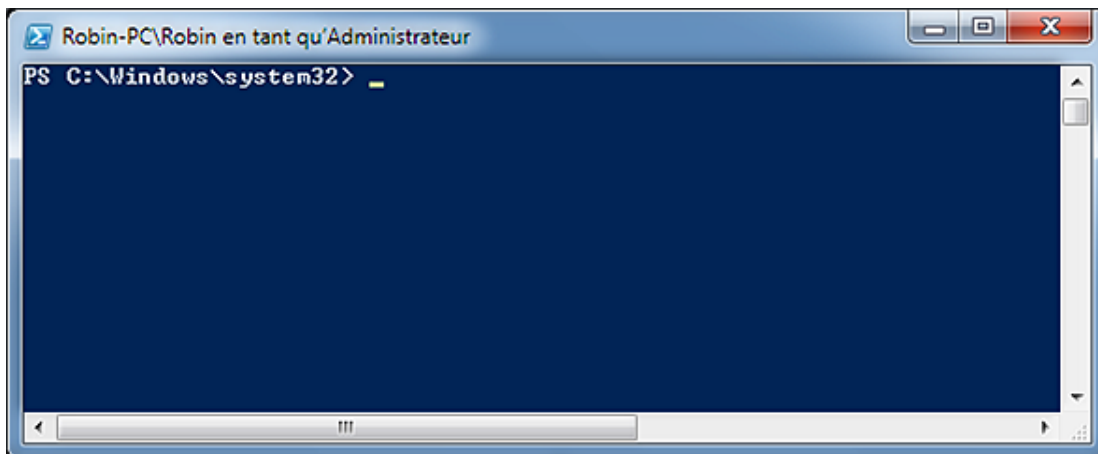
Veuillez noter que cette propriété n'est pas dynamique. Vous ne pourrez donc pas afficher l'heure du système et la voir se rafraîchir en temps réel. Par contre, vous pouvez utiliser la fonction prompt qui se chargera d'actualiser le titre de la fenêtre régulièrement.

Prenons un exemple où nous allons, en fonction de l'utilisateur connecté, afficher son rôle (utilisateur ou administrateur) dans le titre de la fenêtre. Cet exemple n'a aucun intérêt sous Windows XP ou Windows Server, mais il prend tout son sens avec Windows 7 et Vista dans la mesure où même connecté Administrateur, vous lancez par défaut PowerShell en tant que simple utilisateur.

```
$UserType = 'Utilisateur'
$CurrentUser = [System.Security.Principal.WindowsIdentity]::GetCurrent()
$principal =
new-object System.Security.principal.windowsprincipal($CurrentUser)
if ($principal.IsInRole('Administrators'))
{
    $UserType = 'Administrateur'
}
$host.ui.RawUI.WindowTitle = "$($CurrentUser.Name) en tant qu'$UserType"
```



Modifier le titre de la console : Mode Utilisateur



Modifier le titre de la console : Mode Administrateur



Sous Windows 7 et Vista, pour lancer PowerShell en mode administrateur, vous devez faire un clic droit sur l'icône PowerShell et choisir « exécuter en tant qu'administrateur ».

e. Ajout d'un message d'accueil personnalisé

Au lieu de modifier le titre de votre fenêtre pour indiquer le statut de l'utilisateur connecté, vous pouvez tout aussi bien décider de l'afficher uniquement au lancement de votre console PowerShell en ajoutant le code adéquat à votre profil.

```
$UserType = 'Utilisateur'
$CurrentUser =[System.Security.Principal.WindowsIdentity]::GetCurrent()
$principal =
New-Object System.Security.principal.windowsprincipal($CurrentUser)
if ($principal.IsInRole('Administrators'))
{
    $UserType = 'Administrateur'
}
Write-Host '+-----+'
Write-Host "+- Bonjour $($CurrentUser.Name)"
Write-Host "+- Vous êtes connecté en tant que : $UserType"
Write-Host '+-----+'
```

Résultat :

```
+-----+
+- Bonjour Robin-PC\Robin
+- Vous êtes connecté en tant que : Administrateur
+-----+
```

Vous pouvez également faire en sorte que le fond de la fenêtre s'affiche en rouge, en ajoutant le code suivant dans le bloc if :

```
$host.ui.RawUI.BackgroundColor='Red'
Clear-Host
```

f. Prise en compte de scripts externes

À force d'utiliser PowerShell vous allez vite vous constituer une bibliothèque de scripts importante que vous aurez envie de réutiliser. Vous pourriez avoir envie d'ajouter vos créations à votre profil, mais cela risque vite de le surcharger et le rendre difficilement lisible. Nous vous proposons donc un petit bout de code pour les importer facilement dans votre environnement.

```
Write-Host 'Importation des scripts externes'
Get-ChildItem "$home\Scripts" | Where {$_.extension -eq '.ps1'} |
    Foreach {$_.fullname; . $_.fullname}
```

Ce script recherche tous les fichiers dont l'extension se termine par « .ps1 » dans le répertoire **\$home\Scripts**, puis les exécute un à un dans la portée courante.

Get-ChildItem "\$home\Scripts" liste tous les fichiers du répertoire **Scripts** et les passe un à un à la clause **where** à travers le pipe. La clause **where** sert de filtre. Elle examine l'extension du fichier et regarde si elle est « .ps1 » (le test n'est pas sensible à la casse). Si la clause **where** est vraie alors notre objet fichier est passé à la commande suivante du pipe. L'instruction **Foreach** prend alors le relais et pour chaque objet reçu, elle va retourner son nom, puis exécuter le script dans la portée courante (grâce au point et à l'espace ". " placés devant le nom du script - c'est la technique du *DotSourcing*).



Pour que ce script soit pleinement utilisable, vous devez écrire vos scripts sous forme de fonction (ou de filtre). Ainsi une fois vos scripts chargés en mémoire, vous n'aurez plus qu'à appeler le nom de leur fonction (ou filtre).

g. Prise en compte de fichiers de définitions de types personnalisés

Pour le moment si vous ignorez ce que sont les fichiers de types personnalisés, vous pouvez sauter ce paragraphe et y revenir ultérieurement. Pour les autres, on continue...

Exactement sur le même principe que précédemment, nous allons rechercher tous les fichiers dont le nom se termine par *.types.ps1xml, puis nous allons les importer grâce à la commande **Update-TypeData**.

```
Write-Host 'Importation des types personnalisés'
```

```
gci "$home\Scripts" | ? {$_name -match 'types.ps1xml'} |  
% {$_fullname; update-typedata $_fullname}
```



Nous avons cette fois-ci remplacé les instructions **Where** et **Foreach** par leurs alias respectifs **?** et **%**.

h. Prise en compte de fichiers de formatage personnalisés

Tant que nous y sommes allons jusqu'au bout des choses et faisons de même pour importer les fichiers de formatage personnalisés. Cette fois-ci nous allons chercher les fichiers dont le nom se termine par ***.format.ps1xml**.

```
Write-Host 'Importation des affichages personnalisés'  
gci "$home\Scripts" | ? {$_name -match 'format.ps1xml'} |  
% {$_fullname; update-formatdata -prepend $_fullname}
```

Ajout de méthodes et propriétés personnalisées

Comme nous vous le disions en introduction, PowerShell est extensible. Nous allons voir à présent comment ajouter de nouvelles propriétés et méthodes à des types de données. Car qu'il y a-t-il de plus frustrant que de lister un grand nombre de propriétés et de ne pas trouver celle que l'on cherche ? Qu'à cela ne tienne, grâce à PowerShell vous allez pouvoir les rajouter vous-même !

Prenons un exemple pour illustrer nos propos. Lorsque vous utilisez la commande **Get-Member** sur un fichier ou sur un dossier, vous avez en retour une liste conséquente de propriétés et méthodes associées. Vous en avez exactement 77 (69 avec PowerShell v1) pour un fichier, et 65 (58 avec PowerShell v1) pour un dossier (merci aux commandes `Get-Item monFichier|Get-Member -force |Measure-Object`).

Bien sûr, la propriété que nous recherchons n'y est pas (c'est toujours comme ça ! ☹) : nous aurions bien aimé connaître le propriétaire d'un fichier.

Pas de panique ! Commençons par lister les méthodes et propriétés d'un fichier en tapant la commande suivante :

```
PS > Get-Item monFichier.txt | Get-Member -Force
```

Name	MemberType	Definition
-----	-----	-----
Mode	CodeProperty	System.String Mode{get=Mode;}
pstypenames	CodeProperty	System.Collections.ObjectModel...
psadapted	MemberSet	psadapted {Name, Length, Direct..
PSBase	MemberSet	PSBase {Name, Length, Directory...
psextended	MemberSet	psextended {PSPath, PSParentPat...
psobject	MemberSet	psobject {Members, Properties,...
PSStandardMembers	MemberSet	PSStandardMembers {DefaultDispl...
AppendText	Method	System.IO.StreamWriter AppendTe...
CopyTo	Method	System.IO.FileInfo CopyTo(strin...
Create	Method	System.IO.FileStream Create()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef...
CreateText	Method	System.IO.StreamWriter CreateTe...
Decrypt	Method	System.Void Decrypt()
Delete	Method	System.Void Delete()
Encrypt	Method	System.Void Encrypt()
Equals	Method	bool Equals(System.Object obj)
GetAccessControl	Method	System.Security.AccessControl.F...
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeServic...
GetObjectData	Method	System.Void GetObjectData(Syste...
GetType	Method	type GetType()
get_Attributes	Method	System.IO.FileAttributes get_At...
get_CreationTime	Method	System.DateTime get_CreationTim...
get_CreationTimeUtc	Method	System.DateTime get_CreationTim...
get_Directory	Method	System.IO.DirectoryInfo get_Dir...

En y regardant de plus près, nous pouvons observer la méthode **GetAccessControl**. Celle-ci possède un nom fort intéressant, et en cuisinant un peu cette méthode, elle va bien finir par nous donner l'information que l'on recherche...

À présent listons les propriétés et méthodes commençant par « get » associées à la classe **getAccessControl** :

```
PS > (Get-Item monFichier.txt).getAccessControl() | Get-Member |
where {$_.name -like "get*"}
```

Name	MemberType	Definition
-----	-----	-----
GetAccessRules	Method	System.Security.AccessContro...
GetAuditRules	Method	System.Security.AccessContro...
GetGroup	Method	System.Security.Principal.Id...
GetHashCode	Method	System.Int32 GetHashCode()
GetOwner	Method	System.Security.Principal.Id...
.....		

On touche au but... Nous voyons qu'une méthode (**GetOwner**) possède un nom qui ressemble à ce que nous cherchons. Maintenant essayons ceci :

```
PS > (Get-Item monFichier.txt).getAccessControl().GetOwner()

Surcharge introuvable pour « GetOwner » et le nombre d'arguments « 0 ».
Au niveau de ligne : 1 Caractère : 54
+ (Get-Item monFichier.txt).getAccessControl().GetOwner( << )
```

Malheureusement cela aurait été trop simple, et cette ligne de commandes nous renvoie un message d'erreur pas très sympathique ! En effet, si l'on regarde de plus près la définition de cette méthode :

```
PS > (Get-Item monFichier.txt).getAccessControl() | Get-Member |
where {$_.name -eq "getOwner"} | format-list

TypeName      : System.Security.AccessControl.FileSecurity
Name           : GetOwner
MemberType     : Method
Definition     : System.Security.Principal.IdentityReference GetOwner(Type
targetType)
```

On s'aperçoit qu'elle s'attend à ce qu'on lui passe un paramètre de type **targetType**.

Il va nous falloir un peu d'aide pour trouver les types attendus, car ceux-ci ne se trouvent pas dans l'aide standard de PowerShell. C'est un peu normal car nous sommes en train de manipuler directement des objets du Framework .NET.

À ce stade, il ne nous reste qu'une seule chose à faire : aller consulter l'aide directement chez Microsoft et en particulier la base de connaissances MSDN.



Pour obtenir de l'aide sur les classes d'objets du framework .NET, utilisez l'URL suivante : <http://msdn2.microsoft.com> et collez le nom de la classe recherchée dans le champ **Recherche**, en haut à droite de la page.

Après avoir pris de l'information sur le site MSDN nous avons découvert que la classe **IdentityReference** attendait en paramètre les classes **NTAccount** ou **SecurityIdentifier**.

■ Essayons maintenant ceci :

```
PS > (Get-Item monFichier.txt).getAccessControl().GetOwner(`
[System.Security.Principal.NTAccount])

Value
-----
Robin-PC\Robin
```

Ouf, cela fonctionne ! Nous récupérons le nom du propriétaire du fichier en question, ainsi que le nom du domaine associé à son compte (ici Robin-PC).



Lorsque nous faisons appel à un type ou à une classe d'objet .NET, n'oubliez pas de le spécifier entre crochets, comme dans l'exemple ci-après : `[System.Security.Principal.NTAccount]`

Tant que nous y sommes, voyons ce que donne la commande si on lui spécifie la classe **SecurityIdentifier** :

```
PS > (Get-Item monFichier.txt).getAccessControl().GetOwner(`
[System.Security.Principal.SecurityIdentifier]) | Format-List

BinaryLength      : 28
AccountDomainSid   : S-1-5-21-2069618812-4153402021-1334178849
Value              : S-1-5-21-2069618812-4153402021-1334178849-1002
```

Nous avons cette fois récupéré deux SID (*Security Identifier*) : le SID correspondant au domaine d'appartenance de l'utilisateur, ainsi que le SID de l'utilisateur.

Bon, recentrons-nous sur le sujet de cette partie qui, nous vous le rappelons, concerne l'extension du jeu de propriétés et méthodes d'un type donné. Nous savons désormais comment obtenir l'information « propriétaire d'un fichier », mais celle-ci est tellement longue à taper et compliquée que nous risquons de ne pas nous en servir tous les

jours. Nous allons donc en faire une propriété supplémentaire pour le type fichier (**System.IO.FileInfo**).

Cela se fait en plusieurs étapes :

- Création d'un fichier XML décrivant les nouvelles propriétés (et méthodes s'il y a lieu).
- Importation de ce fichier dans PowerShell (utilisation de la commande **Update-TypeData**).

1. Création du fichier de définition de type

Avant de commencer, vous devez savoir que dans PowerShell tous les types existants sont définis dans le fichier `types.ps1xml`. Vous pouvez trouver ce fichier dans le répertoire `%windir%\system32\windowspowershell\v1.0` pour les environnements 32 bits et dans `%windir%\syswow64\WindowsPowerShell\v1.0` pour les systèmes 64 bits (les plus attentifs noteront au passage que ce chemin n'est ni plus ni moins le contenu de la variable `$PSHOME`). Il s'agit d'un fichier XML que vous pouvez ouvrir dans le bloc-notes. Pour en visionner le contenu, nous vous conseillons d'en faire une copie et de changer l'extension en `.xml`. Ainsi il s'ouvrira automatiquement dans Internet Explorer, et vous bénéficierez de la coloration syntaxique et bien plus encore... Ce fichier XML possède une grammaire (ou schéma XML) qui lui est propre.



Bien qu'il soit possible de modifier directement le fichier `types.ps1xml`, il est très fortement déconseillé de le faire sous peine de créer un fonctionnement erratique de PowerShell.

Afin de rajouter notre propriété **Owner**, nous allons devoir créer un nouveau fichier `ps1xml`. Vous devez le créer au même endroit que le fichier de type par défaut. Nommons-le par exemple `proprietaire.types.ps1xml`.

Nous vous laissons le découvrir, puis nous vous expliquerons élément par élément comment est constitué ce dernier :

```
<?xml version="1.0" encoding="utf-8" ?>

<Types>
  <Type>
    <Name>System.IO.FileInfo</Name>
    <Members>
      <ScriptProperty>
        <Name>Owner</Name>
        <GetScriptBlock>
          $this.GetAccessControl().getOwner(`
[System.Security.Principal.NTAccount])
        </GetScriptBlock>
      </ScriptProperty>
    </Members>
  </Type>
</Types>
```

Celui-ci est tout droit inspiré du fichier `types.ps1xml` livré en standard dans PowerShell.

Comme vous pouvez le constater, il reste relativement simple à faire et à comprendre. Et vu les services qu'un tel fichier peut rendre, nous aurions tort de nous en priver.

Quelques explications sur sa structure :

La toute première ligne contient l'entête standard d'un fichier XML.

Vient ensuite l'élément racine **Types**. Puis pour chaque nouveau type ou type à étendre vous devez créer un élément **Type**. Vous indiquez ensuite le nom du type visé dans l'élément **Name** et ouvrez une balise **Members**. Celle-ci contiendra chaque nouvelle propriété ou méthode personnalisée. Pour définir une propriété, utilisez l'élément **ScriptProperty**, et pour une méthode **ScriptMethod**. Arrive ensuite le nom de la propriété, puis « l'intelligence » de celle-ci dans un élément **GetScriptBlock**.

Vous pouvez voir que dans un bloc de code, nous utilisons la variable `$this` pour faire référence à l'objet et ainsi accéder à ses propriétés et méthodes.

a. Utilisation de la propriété **Owner**

Nous venons à présent de définir la propriété **Owner**. Pour la tester, rien de plus simple, utilisez la commande suivante pour que PowerShell charge le nouveau fichier de définition de type : **Update-TypeData proprietaire.types.ps1xml**.

➤ Attention toutefois à la stratégie d'exécution de script choisie (cf. chapitre Sécurité). Les fichiers *.ps1xml sont des fichiers de description, mais ces fichiers sont signés numériquement. Attention donc au possible message d'erreur concernant la nom signature de ce type de fichier lors de leur chargement avec la commande Update-TypeData.

Maintenant, si vous utilisez la commande **Get-Member** pour obtenir la liste des propriétés vous devriez voir apparaître **Owner**.

```
PS > Get-Item monFichier.txt | Get-Member -Type ScriptProperty

TypeName: System.IO.FileInfo

Name      MemberType      Definition
----      -
Owner     ScriptProperty  System.Object Owner {get=$this.GetAccessContr...
Mode      ScriptProperty  System.Object Mode {get=$catr = "";...
```

Pour tester notre nouvelle propriété, essayez ceci :

```
PS > (Get-Item monFichier.txt).Owner

Value
-----
Robin-PC\Robin
```

Ou bien, dans un autre genre :

```
PS > Get-ChildItem *.txt | Format-Table Name,Owner -autosize

Name      Owner
-----
donnees.txt  Robin-PC \Robin
MonFichier.txt  Robin-PC \Arnaud
test.txt     BUILTIN\Administrateurs
```

b. Ajout de la seconde propriété OwnerSID

Si nous avions voulu ajouter une deuxième propriété, par exemple la propriété **OwnerSID**, il aurait fallu ajouter un autre élément **ScriptProperty** de la même façon que précédemment. Comme ci-dessous :

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
  <Type>
    <Name>System.IO.FileInfo</Name>
    <Members>
      <ScriptProperty>
        <Name>Owner</Name>
        <GetScriptBlock>
          $this.GetAccessControl().getOwner(`
[System.Security.Principal.NTAccount])
        </GetScriptBlock>
      </ScriptProperty>
      <ScriptProperty>
        <Name>OwnerSID</Name>
        <GetScriptBlock>
          $this.GetAccessControl().getOwner(`
[System.Security.Principal.SecurityIdentifier])
        </GetScriptBlock>
      </ScriptProperty>
    </Members>
```

```
</Type>
</Types>
```

Tout comme dans l'exemple précédent, n'oubliez pas de charger votre fichier de type avec la commande **Update-TypeData**.

Pour tester votre nouvelle propriété, essayez ceci :

```
PS > (Get-Item monFichier.txt).OwnerSID | Format-List

BinaryLength      : 28
AccountDomainSid  : S-1-5-21-2069618812-4153402021-1334178849
Value             : S-1-5-21-2069618812-4153402021-1334178849-1002
```



Vous pouvez, au choix, créer un seul fichier de types personnalisés et mettre toutes vos extensions à l'intérieur (en créant un nouvel élément Type au même niveau que celui existant pour chaque nouveau type à étendre), ou bien créer un fichier (*.types.ps1xml) par type à étendre.

c. Ajout des méthodes personnalisées **SetOwner** et **GetMSDNHelp**

Poussons notre exemple encore un peu plus loin en ajoutant deux méthodes :

- **SetOwner** : celle-ci va nous permettre de changer le propriétaire d'un fichier.
- **GetMSDNHelp** : grâce à elle nous allons pouvoir demander de l'aide sur le type d'objet en cours d'utilisation. Cette méthode va nous ouvrir le site Internet de MSDN directement à la bonne page.



Cet exemple est tiré du « Blog de Janel » (cf. chapitre Ressources complémentaires - Ressources externes).

Pour implémenter la méthode **SetOwner**, ajoutez le morceau de code ci-dessous à la suite des éléments **ScriptProperty** de l'exemple précédent.

```
<ScriptMethod>
  <Name>SetOwner</Name>
  <Script>
    $argument = $args[0]
    $a = $this.GetAccessControl()
    $a.SetOwner([System.Security.Principal.NTAccount]$argument)
    $this.SetAccessControl($a)
  </Script>
</ScriptMethod>
```

Vous l'aurez deviné, **SetOwner** nécessite qu'on lui passe un argument en entrée pour fonctionner.

Pour l'utiliser, faites comme cela :

```
PS > (Get-Item monFichier.txt).SetOwner('monDomaine\monUtilisateur')
```



Pour ajouter une méthode, nous avons utilisé l'élément **ScriptMethod** au lieu de **ScriptProperty** qui sert à ajouter une propriété. De même qu'à l'intérieur d'une définition de méthode, il faut utiliser l'élément **Script** au lieu de **GetScriptBlock** pour une propriété.

d. Mise en œuvre de la méthode **GetMSDNHelp**

Pour tester cette méthode, nous allons devoir créer un nouveau fichier *.types. ps1xml, nommons-le par exemple **MSDN.types.ps1xml**.

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
```

```

<Type>
  <Name>System.Object</Name>
  <Members>
    <ScriptMethod>
      <Name>GetMSDNHelp</Name>
      <Script>
        $culture = $host.currentculture
        if ($args[0]) { $culture = $args[0] }
        if (($global:MSDNViewer -eq $null) -or
            ($global:MSDNViewer.HWND -eq $null))
        {
          $global:MSDNViewer =
            new-object -ComObject InternetExplorer.Application
        }
        $Uri = 'http://msdn2.microsoft.com/' + $culture `
              + '/library/' + $this.GetType().FullName + '.ASPX'
        $global:MSDNViewer.Navigate2($Uri)
        $global:MSDNViewer.Visible = $TRUE
        $ShellObj = new-object -com WScript.Shell
        $ShellObj.AppActivate((get-process |
            where {$_.MainWindowHandle -eq $global:MSDNViewer.HWND}).Id)
      </Script>
    </ScriptMethod>
  </Members>
</Type>
</Types>

```

Maintenant, comme d'habitude, utilisons la commande : **Update-TypeData MSDN.types.ps1xml**

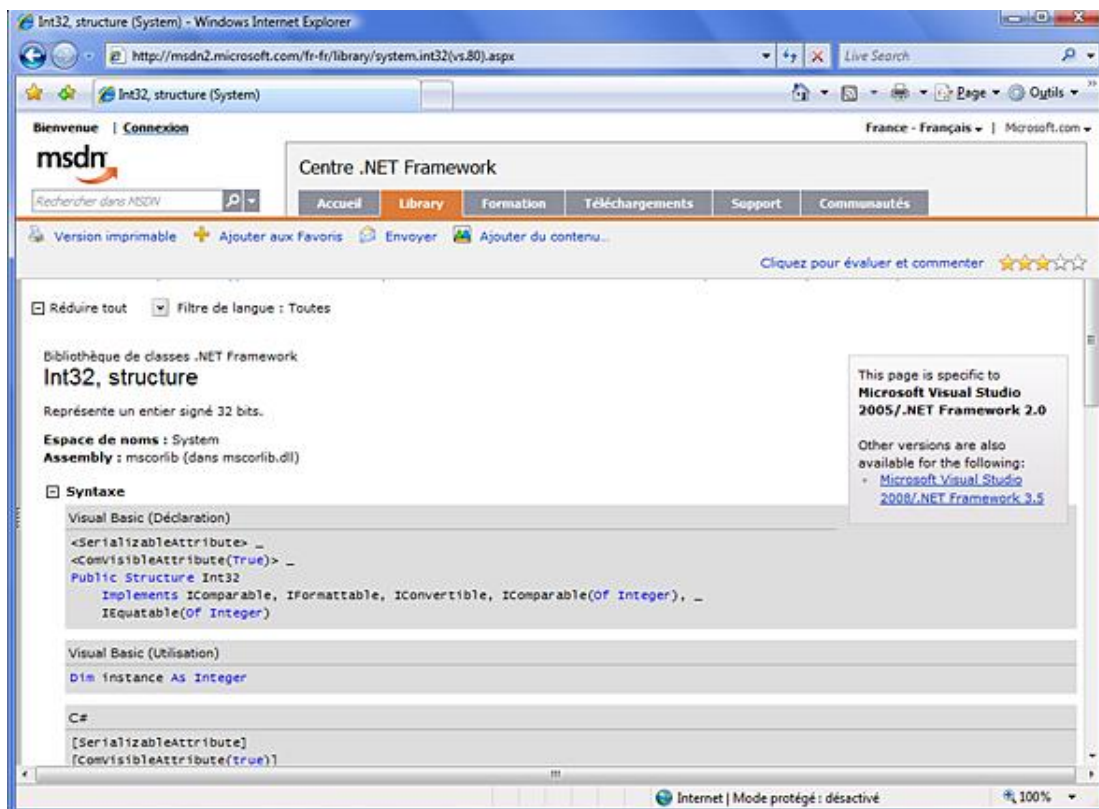
- Essayons notre nouvelle méthode :

```

PS > [int]$var = 66
PS > $var.GetMSDNHelp()

```

Notre méthode fonctionne : Internet explorer s'ouvre sur le site MSDN et nous donne de l'information sur le type « Int32 » de notre variable **\$var**.



Test de la méthode GetMSDNHelp



Pour obtenir de l'information encore plus détaillée sur l'extension des types, vous pouvez vous reporter à l'adresse suivante : <http://msdn2.microsoft.com/en-us/library/ms714665.aspx>

Formatage de l'affichage et personnalisation

Dans le chapitre À la découverte de PowerShell nous avons vu que le résultat d'une commande renvoie la plupart du temps un ensemble de propriétés (que nous appellerons « propriétés par défaut ») formaté généralement sous forme de tableau ou de liste.

Une des grandes forces de PowerShell est qu'il nous offre la possibilité de modifier le jeu de valeurs affiché par défaut ; et ce non pas pour chaque commande mais pour chaque type. En effet, on pourrait penser que les valeurs par défaut qui s'affichent lors de l'exécution d'une commande sont propres aux commandes ; mais ce n'est absolument pas le cas. C'est le type de l'objet (à afficher) qui déterminera son formatage.

En réalité, lorsque vous exécutez une commande dans la console PowerShell, le flux d'objets résultant de la commande est transmis à la commande **Out-Default** via le pipe. Cela est ainsi pour toutes les commandes (qui affichent quelque chose à l'écran) que vous pouvez saisir dans l'interpréteur.

Out-Default est responsable de l'affichage et du formatage des flux d'objets. Si le flux d'objets est de type chaîne, alors **Out-Default** passe directement celui-ci, toujours par le pipe, à la commande **Out-Host**. À l'inverse, si le flux ne contient pas de chaînes, alors **Out-Default** inspecte l'objet et détermine ce qu'il doit en faire.

Premièrement, Powershell va déterminer le type de l'objet et essayer de lui trouver une vue prédéfinie. Les vues prédéfinies sont décrites dans un fichier XML, dont le nom est de la forme *.**format.ps1xml**. Nous verrons juste après comment les modifier ou en créer de nouvelles. Donc, si une vue existe pour le type d'objet en question, alors celui-ci sera formaté en fonction de la définition de la vue. En d'autres termes, si la définition de la vue est un tableau, alors **Out-Default** transmettra le flux d'objet à la commande de formatage adéquat (telle que **Format-Table**, **Format-List** ou **Format-Wide**), soit dans ce cas **Format-Table**.

Remarquez que l'exécution de toutes ces commandes nous donne exactement le même résultat :

```
Get-ChildItem
Get-ChildItem | Out-Default
Get-ChildItem | Format-Table
Get-ChildItem | Format-Table | Out-Host
Get-ChildItem | Format-Table | Out-String | Out-Host
Get-ChildItem | Format-Table | Out-String | Out-Default
```

Maintenant s'il n'y a pas de vue prédéfinie pour l'objet que nous voulons afficher, **Out-Default** recherche le premier objet dans le flux et compte le nombre de ses propriétés. Si l'objet en possède cinq ou plus, **Out-Default** enverra alors le flux à **Format-List**, sinon il l'enverra à **Format-Table**. Lorsque le flux d'objets est transmis à **Format-Table**, cette commande va devoir générer des colonnes. Pour ce faire, elle va créer autant de colonnes (moins de cinq donc) que de propriétés que possède le premier objet du flux. Par exemple, si le premier objet du flux possède trois propriétés, alors le tableau aura trois colonnes, même si le second objet possède dix propriétés. Dans ce cas, il y aura donc un problème pour afficher les autres objets.



Pour en savoir plus sur le formatage des objets, vous pouvez consulter le lien suivant : <http://blogs.msdn.com/powershell/archive/2006/04/30/586973.aspx>. Il s'agit d'une explication détaillée de la part de **Jeffrey Snover**, l'architecte de PowerShell.

1. Découverte des fichiers de formatage par défaut

Comme vous avez pu le comprendre, PowerShell est livré avec un affichage prédéfini pour chaque type d'objet. Les fichiers de définition de formatage se trouvent dans le même répertoire que le fichier de définition des types. À savoir, le répertoire d'installation de PowerShell (%**systemroot%****system32\windowspowershell\v1.0** ou autrement dit **\$PSHOME**).

Ces fichiers sont les suivants (les fichiers pour lesquels figure une étoile sont spécifiques à la version 2 de PowerShell) :

- **Certificate.format.ps1xml**
- **Diagnostics.Format.ps1xml (*)**
- **DotNetTypes.format.ps1xml**
- **FileSystem.format.ps1xml**

- **Getevent.type.ps1xml (*)**
- **Help.format.ps1xml**
- **PowerShellCore.format.ps1xml**
- **PowerShellTrace.format.ps1xml**
- **Registry.format.ps1xml**
- **WSManFormat.ps1xml (*)**

Ce sont des fichiers XML qui possèdent une grammaire (ou schéma XML) propre à PowerShell. Nous vous invitons à en regarder un de près pour vous familiariser un peu avec leur syntaxe si particulière.

Voici la structure générale d'un tel fichier :

```

<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <ViewDefinitions>
    <View>
      <Name>Nom de la vue</Name>
      <ViewSelectedBy>
        <TypeName>Nom du type d'objet à définir</TypeName>
      </ViewSelectedBy>
      <GroupBy>
        <PropertyName>Nom d'une propriété de regroupement</PropertyName>
        <Label>Nom de la propriété</Label>
      </GroupBy>
      <TableControl>
        <TableHeaders>
          <TableColumnHeader>
            <Width>Taille de la colonne</Width>
            <Label>Titre de la colonne</Label>
            <Alignment>Position</Alignment>
          </TableColumnHeader>
          <TableColumnHeader/>
        </TableHeaders>
        <TableRowEntries>
          <TableRowEntry>
            <TableColumnItems>
              <TableColumnItem>
                <PropertyName>Nom de propriété</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName> Nom de propriété </PropertyName>
              </TableColumnItem>
            </TableColumnItems>
          </TableRowEntry>
        </TableRowEntries>
      </TableControl>
    </View>
  </ViewDefinitions>
</Configuration>

```

Définition du nom de la vue

Définition des entêtes de colonnes

Définition du contenu des colonnes

Quelques explications sur la structure :

La toute première ligne contient l'en-tête standard d'un fichier XML ; elle définit sa version et son encodage. Viennent ensuite les éléments racine **Configuration** et **ViewDefinitions**. Puis pour chaque nouvelle vue à définir apparaît un élément **View**. L'élément **name** contient le nom de la vue. **ViewSelectedBy** contient le nom du ou des types cibles. L'élément **TypeName** spécifie tous les types. Le nœud **GroupBy** indique la propriété de regroupement des objets.

Arrive ensuite la définition de la table. L'en-tête des colonnes est d'abord défini avec l'élément **TableColumnHeader** dans lequel est spécifié : la taille de la colonne (en nombre de caractères), son titre, ainsi que l'alignement des données à afficher (left ou right). Sont définies autant d'en-têtes de colonnes que de propriétés à afficher. Si rien n'est indiqué à l'intérieur d'un élément **TableColumnHeader** (comme ceci <TableColumnHeader/> équivaut à <TableColumnHeader> </TableColumnHeader>), alors le titre de la colonne prendra le nom de la propriété et la taille s'ajustera au contenu.



Il est recommandé de nommer les colonnes (le titre) exactement comme les noms des propriétés si elles correspondent à une propriété, de manière à ne pas perturber l'utilisateur. Par exemple, si une propriété s'appelle `ProcessName`, appeler la colonne `ProcessName` et non pas `Name`, ou `Process`, ou `Process Name` avec un espace. Le renommage des propriétés avec des noms plus conviviaux peut être laissé à la discrétion de l'utilisateur en traitement final, avec les options avancées des commandes de formatage.

Et pour finir, le contenu des colonnes est défini dans l'élément `TableColumnItem`. Pour ce faire, peuvent être utilisés les éléments `PropertyName` ou `ScriptBlock`.

`PropertyName` sert à indiquer simplement le nom de la propriété à afficher. Tandis que `ScriptBlock` permet de faire bien plus, comme par exemple appliquer un traitement sur une propriété. Par ce biais, il est possible (entre autres) de mettre en forme une date, convertir une taille de fichiers en kilo-octets ou même afficher une propriété en couleur, etc.

2. Création d'un fichier de formatage personnalisé

Afin de mieux comprendre le fonctionnement des fichiers de formatage, nous allons prendre un cas concret : l'affichage de la commande `Get-ChildItem`. Bien que celle-ci nous rende quotidiennement un précieux service, elle pourrait être grandement améliorée.

`Get-ChildItem` nous renvoie par défaut un certain nombre de propriétés : `Mode`, `LastWriteTime`, `Length` et `Name`.

```
PS > Get-ChildItem $PSHOME

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

Mode                LastWriteTime         Length Name
----                -
d----          14/07/2009          06:56      en-US
d----          14/07/2009          06:52    Examples
d----          04/09/2009          11:19      fr-FR
d----          14/07/2009          09:49    Modules
-a---          10/06/2009          23:24      27338 Certificate.format.ps1xml
-a---          14/07/2009          03:06     126976 CompiledComposition.Microsoft...
-a---          10/06/2009          23:24      27106 Diagnostics.Format.ps1xml
-a---          10/06/2009          23:24      72654 DotNetTypes.format.ps1xml
-a---          10/06/2009          23:24      24857 FileSystem.format.ps1xml
-a---          10/06/2009          23:24      15603 getevent.types.ps1xml
-a---          10/06/2009          23:24     257847 Help.format.ps1xml
-a---          14/07/2009          03:14     452608 powershell.exe
-a---          10/06/2009          23:24      89703 PowerShellCore.format.ps1xml
-a---          10/06/2009          23:24      18612 PowerShellTrace.format.ps1xml
-a---          14/07/2009          03:23     204800 powershell_ise.exe
-a---          14/07/2009          03:06      20480 PSEvents.dll
-a---          14/07/2009          03:23     154624 pspluginwkr.dll
-a---          14/07/2009          03:06       2048 pwrshmsg.dll
-a---          14/07/2009          03:15      24064 pwrshsip.dll
-a---          10/06/2009          23:24      20120 Registry.format.ps1xml
-a---          10/06/2009          23:24     168372 types - Copie.ps1xml
-a---          10/06/2009          23:24     168372 types.ps1xml
-a---          10/06/2009          23:24      24498 WMan.Format.ps1xml
```

Il serait particulièrement agréable d'avoir la taille des fichiers en kilo-octets (Ko), car celle-ci devient difficilement lisible dès que le chiffre devient très grand, ainsi que la date de création des fichiers et des répertoires. Et tant que nous y sommes, pourquoi ne pas essayer de traduire l'intitulé des colonnes en français (attention, ce n'est pas une bonne pratique - voir remarque précédente - mais cela permet de vous montrer tout ce que l'on peut faire) !

Pour ne pas partir de zéro, partons à la recherche du fichier de formatage qui définit les objets de type `FileSystem`. Par chance, il y en a justement un qui se nomme `FileSystem.format.ps1xml` dans le répertoire d'installation de PowerShell.

Pour nous faciliter la tâche, nous pourrions avoir envie de modifier directement ce fichier, mais ceci serait une très mauvaise idée. D'une part, cela pourrait nuire au bon fonctionnement général de PowerShell, et d'autre part, nous pourrions avoir des ennuis avec la sécurité (pas la Police rassurez-vous !☺). En effet, ce fichier a été signé numériquement et si nous apportons une quelconque modification, alors la signature ne correspondra plus au fichier

original et PowerShell pourrait refuser de fonctionner. Nous allons donc plutôt nous en inspirer, en travaillant sur une copie du fichier original. Bien sûr, nous devons supprimer la signature numérique. Pour le reste, nous allons nous contenter de le modifier.

Rajoutons dans la définition de l'entête de la table un élément **<TableColumnHeader>** (qui correspondra à la colonne **CreationTime**) entre celui qui définit la propriété **Mode** et la propriété **LastWriteTime**.

Avant :

```
<TableHeaders>
  <TableColumnHeader>
    <Label>Mode</Label>
    <Width>7</Width>
    <Alignment>left</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>LastWriteTime</Label>
    <Width>25</Width>
    <Alignment>right</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>Length</Label>
    <Width>10</Width>
    <Alignment>right</Alignment>
  </TableColumnHeader>
  <TableColumnHeader/>
</TableHeaders>
```

Après :

```
<TableHeaders>
  <TableColumnHeader>
    <Label>Mode</Label>
    <Width>7</Width>
    <Alignment>left</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>CreationTime</Label>
    <Width>25</Width>
    <Alignment>right</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>LastWriteTime</Label>
    <Width>25</Width>
    <Alignment>right</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>Length</Label>
    <Width>10</Width>
    <Alignment>right</Alignment>
  </TableColumnHeader>
  <TableColumnHeader/>
</TableHeaders>
```

Maintenant il nous faut modifier la définition du contenu des colonnes, en ajoutant - toujours juste après **Mode** - le contenu de la propriété que nous venons de créer. Remplaçons également la propriété **Length** par un bloc de script. Celui-ci va nous faire la conversion octets -> kilo-octets et nous ajouter « Ko » dans la valeur de la propriété. Comme ci-dessous :

Avant :

```
<TableRowEntries>
  <TableRowEntry>
    <Wrap/>
    <TableColumnItems>
      <TableColumnItem>
        <PropertyName>Mode</PropertyName>
      </TableColumnItem>
      <TableColumnItem>
        <ScriptBlock>
```

```

                [String]::Format("{0,10} {1,8}",
                $_.LastWriteTime.ToString("d"),
                $_.LastWriteTime.ToString("t"))
            </ScriptBlock>
        </TableColumnItem>
        <TableColumnItem>
            <PropertyName>Length</PropertyName>
        </TableColumnItem>
        <TableColumnItem>
            <PropertyName>Name</PropertyName>
        </TableColumnItem>
    </TableColumnItems>
</TableRowEntry>
</TableRowEntries>

```

Après :

```

<TableRowEntries>
  <TableRowEntry>
    <Wrap/>
    <TableColumnItems>
      <TableColumnItem>
        <PropertyName>Mode</PropertyName>
      </TableColumnItem>
      <TableColumnItem>
        <PropertyName>CreationTime</PropertyName>
      </TableColumnItem>
      <TableColumnItem>
        <ScriptBlock>
          [String]::Format("{0,10} {1,8}",
          $_.LastWriteTime.ToString("d"),
          $_.LastWriteTime.ToString("t"))
        </ScriptBlock>
      </TableColumnItem>
      <TableColumnItem>
        <ScriptBlock>
          $a = [math]::round($_.length/1024,0)
          if ($a -gt 0) {
            [string]$a += " Ko"
          }
          else {
            $a = ""
          }
          $a
        </ScriptBlock>
      </TableColumnItem>
      <TableColumnItem>
        <PropertyName>Name</PropertyName>
      </TableColumnItem>
    </TableColumnItems>
  </TableRowEntry>
</TableRowEntries>

```

À présent, il ne nous reste plus qu'à faire prendre en compte ce nouveau fichier de formatage à PowerShell. En supposant que vous ayez appelé votre fichier **perso.format.ps1xml**, utilisez la commande suivante :

```
PS > Update-FormatData -Prepend perso.format.ps1xml
```

Le paramètre **-Prepend** indique à PowerShell d'utiliser en priorité ce fichier par rapport à celui natif.

Allons-y, observons si nos modifications ont changé quelque chose :

```
PS > Get-Childitem $PSHOME
```

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

```
Mode                CreationTime         LastWriteTime Length Name
```

```

-----
d--- 14/07/2009 06:52:30 14/07/2009 06:52 Examples
d--- 14/07/2009 10:39:37 14/07/2009 10:39 fr-FR
d--- 14/07/2009 06:52:30 14/07/2009 11:01 Modules
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 27 Ko Certificate.format.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 26 Ko Diagnostics.Format.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 71 Ko DotNetTypes.format.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 24 Ko FileSystem.format.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 15 Ko getevent.types.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 252 Ko Help.format.ps1xml
-a--- 14/07/2009 01:32:37 14/07/2009 03:14 442 Ko powershell.exe
-a--- 13/07/2009 23:47:02 14/07/2009 03:23 200 Ko powershell_ise.exe
-a--- 14/07/2009 01:32:28 14/07/2009 03:06 20 Ko PSEvents.dll
-a--- 14/07/2009 01:32:33 14/07/2009 03:23 151 Ko pspluginwkr.dll
-a--- 14/07/2009 01:32:29 14/07/2009 03:06 2 Ko pwrshmsg.dll
-a--- 14/07/2009 01:32:28 14/07/2009 03:15 24 Ko pwrshsip.dll
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 20 Ko Registry.format.ps1xml
-a--- 10/06/2009 23:24:31 10/06/2009 23:24 164 Ko types.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 24 Ko WSMAN.Format.ps1xml

```

N'est-ce pas tout simplement fabuleux ?



Bien que faisable, il n'est vraiment pas recommandé de modifier la propriété `Length` tel que nous l'avons fait. En effet en ajoutant l'unité « Ko » dans la valeur, nous avons modifié son type. Auparavant la propriété `Length` était de type `int`, et à présent elle est de type `String`. Par conséquent nous ne pourrons plus désormais effectuer facilement des tests sur la taille des fichiers.

Comme convenu, nous pouvons changer l'intitulé des colonnes en modifiant l'élément `<Label>` contenu dans l'élément `<TableHeaders>`, comme ceci :

```

<TableHeaders>
  <TableColumnHeader>
    <Label>Mode</Label>
    <Width>7</Width>
    <Alignment>left</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>Date de creation</Label>
    <Width>25</Width>
    <Alignment>right</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>Date d'écriture</Label>
    <Width>25</Width>
    <Alignment>right</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>Longueur</Label>
    <Width>10</Width>
    <Alignment>right</Alignment>
  </TableColumnHeader>
</TableHeaders>

```

Résultat :

```
PS > Get-Childitem $PSHOME
```

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

```

Mode      Date de création      Date d'écriture Longueur Name
-----
d--- 14/07/2009 06:52:30 14/07/2009 06:52 Examples
d--- 14/07/2009 10:39:37 14/07/2009 10:39 fr-FR
d--- 14/07/2009 06:52:30 14/07/2009 11:01 Modules
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 27 Ko Certificate.format.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 26 Ko Diagnostics.Format.ps1xml

```

```

-a--- 13/07/2009 22:34:42 10/06/2009 23:24 71 Ko DotNetTypes.format.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 24 Ko FileSystem.format.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 15 Ko getevent.types.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 252 Ko Help.format.ps1xml
-a--- 14/07/2009 01:32:37 14/07/2009 03:14 442 Ko powershell.exe
-a--- 13/07/2009 23:47:02 14/07/2009 03:23 200 Ko powershell_ise.exe
-a--- 14/07/2009 01:32:28 14/07/2009 03:06 20 Ko PSEvents.dll
-a--- 14/07/2009 01:32:33 14/07/2009 03:23 151 Ko pspluginwkr.dll
-a--- 14/07/2009 01:32:29 14/07/2009 03:06 2 Ko pwrshmsg.dll
-a--- 14/07/2009 01:32:28 14/07/2009 03:15 24 Ko pwrshsip.dll
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 20 Ko Registry.format.ps1xml
-a--- 10/06/2009 23:24:31 10/06/2009 23:24 164 Ko types.ps1xml
-a--- 13/07/2009 22:34:42 10/06/2009 23:24 24 Ko WSMAN.Format.ps1xml

```



Pour que cela fonctionne et que les accents de nos propriétés s'affichent correctement, il faut modifier le type d'encodage dans la première ligne du fichier ps1xml, en précisant **UTF-16** au lieu de **UTF-8**. **<?xml version="1.0" encoding="utf-16" ?>**. N'oubliez pas non plus de sauvegarder votre fichier en Unicode **UTF-16**. Vous en apprendrez davantage sur le format Unicode dans la section suivante de ce chapitre.

Enfin, toujours avec les fichiers de formatage nous pourrions très bien afficher le nom des fichiers d'une couleur, et les répertoires d'une autre couleur, ou bien encore affecter une couleur en fonction de l'extension de fichiers. Bref, il n'y a vraiment pas de limites !

Nous avons basé tous nos exemples sur le type **FileSystem** mais sachez que vous pouvez créer des affichages personnalisés pour n'importe quel autre type.



Pour en savoir plus sur le formatage des objets, vous pouvez consulter le lien suivant (« extending object types and formatting ») : <http://msdn2.microsoft.com/ru-ru/library/ms714665.aspx>

La gestion de fichiers

La gestion de fichiers n'aura jamais été aussi simple... Ceux d'entre vous qui ont déjà eu l'occasion de s'y confronter avec VBScript seront grandement satisfaits d'apprendre cela. En effet, avec PowerShell, il n'est plus question d'instancier des objets de type filesystem, de les ouvrir en spécifiant le mode d'accès (lecture ou écriture), puis de les fermer. PowerShell apporte un jeu de commandelettes dédié à la gestion de fichiers et nous verrons que cela représente un énorme gain de productivité dans l'écriture des scripts.

Dans le chapitre À la découverte de PowerShell, nous nous étions intéressés au contenant (le fichier lui-même), et nous avons vu comment les créer, les déplacer, les renommer, etc. À présent, nous nous intéresserons au contenu, et nous verrons entre autres, comment en générer et comment le relire.

Il est important de noter que PowerShell traite généralement les fichiers texte en Unicode de façon native (à quelques exceptions près), contrairement à CMD.exe qui ne manipule que de l'ASCII et les pages de code de caractères. Cependant, pour des raisons de compatibilité, il est possible de forcer les commandelettes à utiliser d'autres encodages tels que ASCII, UTF8, UTF32, etc.

1. Envoi de données dans un fichier

Il y a deux façons essentielles de procéder pour écrire des données dans un fichier. Nous pouvons utiliser soit **Set-Content**, soit **Out-File**.

Bien que ces deux commandes servent à faire la même chose : créer des fichiers et des données, il y a cependant une différence notable qu'il est important de connaître mais qui n'est pas facilement décelable alors que l'on débute.

Lorsque **Out-File** est utilisée, elle va tenter, tout comme les autres commandes **out-***, de formater le flux avant de l'écrire dans le fichier.

Set-Content quant à elle, ne cherche pas à formater le flux mais elle lui applique seulement la méthode **ToString** afin d'être sûre d'écrire des caractères. C'est cela la principale différence. Cependant, bien qu'elle puisse sembler anodine au premier abord, vous aurez des surprises si vous tentez d'écrire un objet dans un fichier avec **Set-Content** sans l'avoir formaté au préalable.

Par exemple, le résultat de cette commande écrira dans un fichier le type de l'objet au lieu de son contenu :

```
PS > Get-Process powershell | Set-Content MonFichier.txt
PS > Get-Content MonFichier.txt

System.Diagnostics.Process (powershell)
```

Alors que la commande suivante nous donne le résultat attendu :

```
PS > Get-Process powershell | Out-File MonFichier.txt
PS > Get-Content MonFichier.txt

Handles   NPM(K)  PM(K)  WS(K)  VM(M)   CPU(s)    Id  ProcessName
-----
533       13    64608  65376   219    38,39    2080 powershell
```

Pour obtenir le même résultat avec **Set-Content**, il aurait fallu effectuer un « transtypage » préalable sur l'objet avant de l'écrire, comme ceci :

```
PS > Get-Process powershell | Out-String -Stream | Set-Content
MonFichier.txt
```

Out-String nous permet de convertir les objets émis en les représentant sous forme de chaîne. Le paramètre **-stream** permet d'envoyer au pipe autant de chaînes que d'objets reçus, au lieu d'envoyer une chaîne unique contenant la représentation de tous les objets.

Si nous souhaitons personnaliser le résultat, nous pourrions écrire ceci :

```
Get-Process powershell | Format-Table id, processname | Out-String |
Set-Content MonFichier.txt
```

Une autre différence intéressante est que **Set-Content** permet d'écrire directement des octets dans un fichier grâce au paramètre **-Encoding Byte**. La valeur « **Byte** » de ce paramètre est propre à **Set-Content**, il n'existe pas dans **Out-File**. Cela va permettre de manipuler des fichiers autres que des fichiers textes en écrivant directement des octets.

En résumé, on aura donc plutôt tendance à privilégier l'utilisation de **Out-File** pour créer des fichiers textes, et à

utiliser **Set-Content** pour des fichiers binaires.

a. Les fichiers textes avec Out-File

Cette commande très puissante va nous permettre de créer des fichiers et leurs contenus associés. Elle fait sensiblement la même chose que les opérateurs de redirection (que nous verrons dans la prochaine section), sauf que l'on peut spécifier à **Out-File** un certain nombre de paramètres supplémentaires.

Voici la liste des paramètres :

Paramètres	Description
FilePath <String>	Fichier destination.
Encoding <String>	Type d'encodage (défaut : unicode).
Append <Switch>	Ajoute du contenu à un fichier existant.
Width <Int>	Nombre de caractères maxi par ligne.
InputObject <PSObject>	Objet à écrire dans le fichier.
NoClobber <Switch>	Indique de ne pas remplacer de fichier existant.

Les valeurs possibles pour le paramètre d'encodage sont les suivantes :

Nom	Description
Ascii	Force l'encodage en ASCII de base (jeu de caractères 0 à 127, 7 bits).
UTF7	Force l'encodage en Unicode UTF7 (<i>Unicode Transformation Format</i>).
UTF8	Force l'encodage en Unicode UTF8.
Unicode	Force l'encodage en Unicode UTF16 LittleEndian.
BigEndianUnicode	Force l'encodage en Unicode UTF16 BigEndian.
UTF32	Force l'encodage en Unicode UTF32.
Default	Utilise le codage de la page de codes ANSI actuelle du système.
Oem	Utilise l'identificateur de la page de codes du fabricant d'ordinateurs OEM (<i>Original Equipment Manufacturer</i>) actuel pour le système d'exploitation.

 Microsoft Windows travaille en interne en Unicode UTF16 LittleEndian. LittleEndian signifie que dans un mot (2 octets), l'octet le moins significatif est positionné en premier. L'inverse est la notation BigEndian où l'octet significatif est en premier. Par exemple, si l'on souhaitait coder le chiffre 10 (base décimale) en hexadécimal sur 16 bits, cela donnerait : 00 0A en LittleEndian, 0A 00 en BigEndian.

 Il est généralement plus efficace d'utiliser l'ordre d'octet natif pour stocker des caractères Unicode. Ainsi il est préférable d'utiliser l'ordre d'octet LittleEndian sur les plates-formes little-endian de type Intel et l'ordre d'octet BigEndian sur les plates-formes Motorola.

Exemple :

Création d'un fichier ASCII contenant des informations sur un processus du système.

```
PS > Get-Process powershell |  
Out-File c:\temp\test\monfichier.txt -Encoding ascii
```

Cette commande va créer le fichier ASCII monfichier.txt dans le répertoire c:\temp\test. Ce fichier contiendra le résultat d'exécution de la commande précédente passée au travers du pipeline.

Exemple 2 :

Ajout de données à un fichier existant.

```
PS > Get-Date | Out-File c:\temp\test\monfichier.txt -Append -Encoding ascii
```

Dans cet exemple, nous ajoutons des données au fichier que nous avons créé dans l'exemple précédent. Faites bien attention de *toujours* spécifier le *même* format d'encodage lorsque vous ajoutez des données à un fichier. PowerShell ne vous préviendra pas, mais si les formats de vos données diffèrent votre fichier deviendra illisible.



Lorsque vous ajoutez des données à un fichier texte, n'oubliez jamais de tenir compte de l'encodage de celui-ci, sous peine de rendre votre fichier illisible. Une méthode simple quand vous ne connaissez pas l'origine d'un fichier et que vous avez des données à lui ajouter, est de l'ouvrir dans le bloc-notes et de faire comme si vous vouliez l'enregistrer avec « enregistrer sous ». Ainsi dans le bas de la fenêtre, vous pourrez voir une liste déroulante nommée « codage » vous permettant de choisir l'encodage désiré, sachant que le choix proposé par défaut est celui du fichier que vous avez ouvert.



Il existe un autre éditeur de texte très bien et freeware qui s'appelle « ConTEXT » que nous vous recommandons. Avec ConTEXT, dès que vous ouvrez un fichier, son type d'encodage est affiché dans la barre d'état située tout en bas de la fenêtre ; ce qui est pratique. Et bien entendu vous aurez droit, comme tout éditeur de textes digne de ce nom, à la coloration syntaxique, ainsi qu'à bien d'autres fonctions.

b. Redirection du flux standard

Création de fichiers

Nous avons vu dans le chapitre Fondamentaux qu'il existait un opérateur de redirection, l'opérateur *supérieur* à « > ». Cet opérateur représente la forme la plus simple pour créer un fichier. Il fonctionne à l'identique que sous CMD.exe (à l'exception près du type d'encodage par défaut qui est Unicode). À savoir que lorsqu'il est utilisé, le flux de sortie standard est redirigé dans un fichier texte.

Exemple :

```
PS > Get-childItem C:\temp > dir.txt
```

Cette ligne de commandes liste les fichiers et dossiers contenus dans le répertoire **C:\temp** dans le fichier **dir.txt**.

Pas de changement donc pour les habitués du CMD.exe, pour le fonctionnement de cet opérateur.

Ajout de données à un fichier

Pas de changement non plus pour l'ajout de données, qui se réalise toujours avec l'opérateur de redirection « >> ». Ainsi, grâce à cet opérateur, nous pouvons ajouter du contenu à la fin d'un fichier existant.

Exemple :

```
PS > Get-Date >> dir.txt
```

Cette ligne de commandes aura pour effet de rajouter la date courante à la fin du fichier **dir.txt**, et ce tout en préservant le contenu présent à l'intérieur du fichier.



Les opérateurs de redirection de flux « > » et « >> » font en réalité appel à la commandelette **Out-File**. Pour en avoir le cœur net, appelons à la rescousse **Trace-Command** (que nous détaillerons dans le prochain chapitre) pour tenter de découvrir ce qu'il y a à l'intérieur de la bête... Essayons cela :

```
PS > Trace-command -Name CommandDiscovery -Expression `
```

```
{get-date > test.txt} -PSHost
```

```
DÉBOGUER : CommandDiscovery Information: 0 : Looking up command: get-date
DÉBOGUER : CommandDiscovery Information: 0 :
    Attempting to resolve function or filter: get-date
DÉBOGUER : CommandDiscovery Information: 0 :
    Cmdlet found:
Get-Date  Microsoft.PowerShell.Commands.GetDateCommand,
    Microsoft.PowerShell.Commands.Utility,
    Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=31bf3856ad364e35
DÉBOGUER : CommandDiscovery Information: 0 : Looking up command: out-file
DÉBOGUER : CommandDiscovery Information: 0 :
    Attempting to resolve function or filter: out-file
DÉBOGUER : CommandDiscovery Information: 0 :
    Cmdlet found: Out-File Microsoft.PowerShell.
    Commands.OutFileCommand,
    Microsoft.PowerShell.Commands.Utility,
    Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=31bf3856ad364e35
```



Nous voyons apparaître sur la dernière ligne « Cmdlet found: Out-File », CQFD ! Mais nous pouvons encore faire mieux en regardant quelles sont les valeurs que PowerShell affecte aux différents paramètres de **Out-File**. Essayons cette ligne de commandes :

```
PS > Trace-command -Name ParameterBinding -Expression `
{get-date > test.txt} -PSHost
```



Pour des raisons d'encombrement dues à la verbosité de **Trace-Command**, nous n'afficherons pas l'intégralité du résultat mais seulement les lignes les plus significatives. Ainsi vous devriez voir ceci :

```
BIND arg [test.txt] to param [FilePath]
BIND arg [unicode] to parameter [Encoding]
BIND arg [16/08/2009 19:50:19] to parameter [InputObject]
```



Cela confirme bien ce que l'on vous disait plus haut, PowerShell encode par défaut ses fichiers en Unicode. On remarque également que le nom de notre fichier est passé au paramètre **-FilePath**. Cette mécanique d'association de paramètres s'applique également à toutes les commandelettes. Par conséquent, lorsque l'on se contente de passer une valeur à un paramètre facultatif (tel que **Get-Childitem monFichier** au lieu de **Get-Childitem -FilePath monFichier**), et bien l'association valeur/paramètre se fait automatiquement en interne.

c. Création de fichiers binaires avec Set-Content

Contrairement à **Out-File**, cette commandelette écrit les données telles qu'elle les reçoit. La grande force de **Set-Content** est de pouvoir écrire directement des octets dans un fichier, et ce quel que soit le type de fichier (texte ou binaire). Mais attention, **Set-Content** écrase le contenu du fichier de destination car elle ne possède pas de switch **-append** comme **Out-File**.

Il ne faut pas oublier que **Set-Content** fait partie de la famille des commandelettes ***-Content**, soit :

- **Add-Content** : ajoute des données à un fichier existant,
- **Clear-Content** : efface les données présentes dans un fichier, mais pas le fichier,
- **Get-Content** : lit le contenu d'un fichier. Nous étudierons cette commandelette en détail un peu plus loin.

Voici les paramètres de **Set-Content** :

Paramètres	Description

Path <String[]>	Fichier destination recevant les données.
Value <Object[]>	Données à écrire (remplaceront le contenu existant).
Include <String[]>	Modifie uniquement les éléments spécifiés.
Exclude <String[]>	Omet les éléments spécifiés.
Filter <String>	Spécifie un filtre dans le format ou le langage du fournisseur.
PassThru <Switch>	Passe l'objet créé par cette commande à travers le pipeline.
Force <Switch>	Force la commande à réussir sans compromettre la sécurité, par exemple en créant le répertoire de destination s'il n'existe pas.
Credential <PSCredential>	Utilise des informations d'identification pour valider l'accès au fichier.
Encoding <String>	Type d'encodage (valeur par défaut : « default », soit ANSI).

Les valeurs possibles pour le paramètre d'encodage sont les suivantes :

Nom	Description
ASCII	Force l'encodage en ASCII de base (jeu de caractères 0 à 127, 7 bits).
UTF7	Force l'encodage en Unicode UTF7.
UTF8	Force l'encodage en Unicode UTF8.
Unicode	Force l'encodage en Unicode UTF16 LittleEndian.
BigEndianUnicode	Force l'encodage en Unicode UTF16 BigEndian.
Byte	Force l'encodage en octet.
String	Utilise le codage de la page de codes ANSI actuelle du système.
Unknown	Idem Unicode.

Faites attention car ce ne sont pas les mêmes valeurs que pour la commande **Out-File**.

Bien qu'il soit quand même possible d'écrire des données textuelles avec **Set-Content** (moyennant de prendre les précautions énoncées en introduction), le plus intéressant est la possibilité d'écrire directement des octets dans un fichier.

 Si vous envoyez des données de type **String** dans un fichier sans spécifier explicitement l'encodage désiré, le fichier résultant sera un fichier ANSI. C'est-à-dire un fichier ASCII étendu avec votre page de code courante pour prendre en compte les caractères accentués.

Exemple :

Envoi de données textuelles dans un fichier.

```
PS > 'AAéBB' | set-content test.txt
```

Cette ligne de commandes crée le fichier **test.txt** au format ANSI. À présent regardons quelle est la taille de ce fichier :

```
PS > Get-ChildItem test.txt
```

Répertoire : C:\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	28/08/2009 23:53	7	test.txt

Pourquoi diable avons-nous un fichier de 7 octets alors que nous n'avons envoyé que cinq caractères à l'intérieur ?

Grâce à une petite fonction personnalisée de notre cru, nous allons pouvoir passer au peigne fin tous les octets qui composent notre fichier.

Notre fonction Get-Dump, comme son nom l'indique, « dump » le contenu d'un fichier en décimal, hexadécimal et ASCII :

```
function Get-Dump
{
    param ([string]$path=$(throw 'Chemin non trouvé'),
          [int]$taille=(gci $path).Length)

    $fic = Get-Content -Path $path -Encoding byte -TotalCount $taille
    [string]$strDest = ''
    [string]$strAsciiDest = ''
    [string]$strHexDest = ''
    for ($i=0; $i -lt $taille; $i++)
    {
        $strDest += $fic[$i]
        $strDest += ' '
        $strAsciiDest += [char]$fic[$i]
        $strHexDest += ('{0:x}' -f $fic[$i]).PadLeft(2, '0')
        $strHexDest += ' '
    }

    Write-host "DEC:  $strDest"
    Write-host "HEX:  $strHexDest"
    Write-host "ASCII: $strAsciiDest"
}
```

Cette fonction devrait nous aider à mieux comprendre d'où provient cette différence de taille.

```
PS > Get-Dump test.txt

DEC:  65 65 233 66 66 13 10
HEX:  41 41 e9 42 42 0d 0a
ASCII: AAéBB
```

65, 66, et 233 sont respectivement les codes ASCII des caractères « A », « B », et « é » ; jusque-là tout est normal. Seulement voilà, nous pouvons constater que nous avons deux octets supplémentaires en fin de fichier qui sont venus se rajouter automatiquement. Ces octets 13 et 10 en décimal ou 0D, 0A en hexadécimal correspondent aux caractères **CR** (*Carriage Return*) et **LF** (*Line Feed*). Autrement dit, un retour chariot et un retour à la ligne.

Ceci est tout à fait normal car sur la plate-forme Windows (c'était déjà le cas sous DOS), chaque ligne d'un fichier texte se termine par **CR** et **LF**. Alors que sous Unix (et autres dérivés) une ligne se termine uniquement par **LF**. C'est ce qui explique pourquoi il y a quelques problèmes de mise en forme lorsque l'on échange des fichiers textes entre ces plates-formes...



L'ajout des codes de contrôle **CR** et **LF** se produit également avec la commande **Out-File**.

Exemple :

*Écriture d'un flux d'octets dans un fichier sans **CR LF**.*

Nous allons dans cet exemple tenter d'écrire une chaîne de caractères dans un fichier mais cette fois-ci nous allons faire en sorte que **CR** et **LF** ne soient pas ajoutés en fin de ligne. Pour ce faire, nous allons envoyer des octets correspondant aux codes ASCII de la chaîne à écrire ; puis nous spécifierons le type d'encodage **byte** pour **Set-Content**.

```
PS > [byte[]][char[]]'AAéBB' | Set-Content test.txt -Encoding byte
```

En faisant cela, nous convertissons la chaîne « AAéBB » en un tableau de caractères, que nous convertissons ensuite en un tableau d'octets, puis nous passons le tout à **Set-Content** où nous prenons bien soin d'ajouter le paramètre -**encoding byte**.

Exemple :

Convertir un fichier texte Unix en **DOS**.

```
# convert-Unix2Dos.ps1

param ($path=$(throw 'fichier non trouvé'), $dest=$path)

$tab = get-content $path -encoding byte
for ($i=0;$i -lt $tab.length; $i++)
{
    if ($tab[$i] -eq 10)
    {
        $tab=$tab[0..($i-1)]+[byte]13+$tab[$i..$tab.length]
        $i++
    }
}
$tab | Set-Content $dest -encoding Byte
```

Ce petit script convertit un fichier de type Unix en un fichier compatible DOS/Windows en insérant le caractère de contrôle **CR** (13 Dec.) devant chaque caractère **LF** (10 Dec.).

La suite d'octets suivante : 68 74 57 98 102 **10** 65 66 48 **10** 125 139 78

sera transformée ainsi : 68 74 57 98 102 **13 10** 65 66 48 **13 10** 125 139 78

Grâce à l'instruction **param** et à l'initialisation automatique des paramètres, une exception sera levée si vous ne spécifiez pas de fichier source. De plus, si vous omettez de spécifier un fichier de destination, le fichier source sera utilisé comme fichier de destination et son contenu existant sera écrasé.

On stocke ensuite le contenu du fichier source sous forme d'une suite d'octets dans le tableau **\$tab**. Après, c'est un petit peu plus ardu : on parcourt l'intégralité du tableau **\$tab** à la recherche du caractère **LF**. Lorsqu'on en trouve un, on concatène le début de notre tableau avec CR et la fin de notre tableau, puis on réinjecte le nouveau contenu dans notre tableau **\$tab**. En somme, nous écrasons à chaque itération le contenu de **\$tab** par un nouveau contenu modifié. Nous faisons ceci car il n'existe pas de méthode pour insérer un élément dans un tableau à un emplacement donné. Enfin, nous incrémentons notre variable d'indice d'une position car nous avons ajouté un élément dans **\$tab** ; sans quoi le test est toujours vrai et nous tombons dans une boucle infinie. Enfin notre tableau d'octets est passé via le pipe à **Set-Content** sans oublier de spécifier le type d'encodage **byte**.

2. Lecture de données avec Get-Content

Comme vous vous en doutez et comme son nom l'indique **Get-Content** va nous permettre de lire le contenu d'un fichier. Ce dernier peut être soit de type texte, soit de type binaire, peu importe, **Get-Content** s'en accommode à partir du moment où on le lui précise. Par défaut cette commandelette s'attend à lire des fichiers textes.

Voici les paramètres de **Get-Content** :

Paramètres	Description
Path <String[]>	Fichier source contenant les données à lire.
TotalCount <Int64>	Nombre de lignes à lire. Par défaut toutes (valeur -1).
ReadCount <Int64>	Nombre de lignes de contenu envoyées simultanément au pipeline. Par défaut elles sont envoyées une par une (valeur 1). Une valeur de 0 indique qu'on veut envoyer toutes les lignes d'un coup.
Include <String[]>	Récupère uniquement les éléments spécifiés.
Exclude <String[]>	Omet les éléments spécifiés.
Filter <String>	Spécifie un filtre dans le format ou le langage du fournisseur.

Force <Switch>	Force la commande à réussir sans compromettre la sécurité.
Credential <PSCredential>	Utilise des informations d'authentification pour valider l'accès au fichier.
Encoding <String>	Spécifie le type de codage de caractères utilisé pour afficher le contenu.

Les valeurs possibles pour le paramètre d'encodage sont les suivantes :

Nom	Description
ASCII	Force l'encodage en ASCII de base (jeu de caractères 0 à 127, 7 bits).
UTF7	Force l'encodage en Unicode UTF7.
UTF8	Force l'encodage en Unicode UTF8.
Unicode	Force l'encodage en Unicode UTF16 LittleEndian.
BigEndianUnicode	Force l'encodage en Unicode UTF16 BigEndian.
Byte	Force l'encodage en octet.
String	Utilise le codage de la page de codes ANSI actuelle du système.
Unknown	Idem Unicode.

Exemple :

Fonctionnalités de base.

```
PS > Get-Date > mesProcess.txt
PS > Get-Process >> mesProcess.txt

PS > Get-Content mesProcess.txt -Totalcount 10

dimanche 20 septembre 2009 11:22:22

Handles   NPM(K)  PM(K)  WS(K) VM(M) CPU(s)    Id ProcessName
-----
      91         5   3280   1496    62    0,22   3408 ashDisp
     129        140   4340   2072    67    2380 ashMaiSv
     351         10  28156  15888   140    1676 ashServ
     140         40  16312  32156   112    2416 ashWebSv
      30         2    836    396    23    1664 aswUpdSv
```

Dans cet exemple, nous créons un fichier texte avec l'opérateur de redirection « supérieur à » (unicode, donc) qui contient la date et l'heure ainsi que la liste des processus en cours d'exécution. Puis, nous faisons appel à **Get-Content** pour lire et afficher à l'écran les dix premières lignes du fichier.

Exemple :

Manipuler un fichier comme un tableau.

```
PS > $fic = Get-Content FableLaFontaine.txt
PS > $fic[14]
La fourmi n'est pas prêteuse ;
```

En utilisant une variable pour recevoir le résultat de la commande **Get-Content**, nous créons en réalité un tableau de lignes. Et nous affichons ensuite la ligne située à l'indice 14 du tableau (en réalité la 15^{ème} ligne du fichier car n'oubliez pas que les indices de tableau commencent à zéro).

De plus, comme une chaîne est également un tableau de caractères, on peut lire n'importe quel caractère en utilisant la syntaxe des tableaux à deux dimensions. Par exemple, le « i » du mot fourni qui se trouve à l'index 8 :

```
PS > $fic[14][8]
i
```

Enfin pour terminer cet exemple, si nous appliquons la méthode **Length** sur notre tableau **\$fic**, nous obtiendrons le nombre d'éléments qui le composent, soit le nombre de lignes de notre fichier texte.

```
PS > $fic.Length
22
```

Vingt-deux est le nombre de lignes de notre fichier.

Exemple :

Lecture d'un fichier en mode « brut ».

Comme nous vous le disions en introduction de cette commande, **Get-Content** sait lire des octets. Cette fonctionnalité est particulièrement intéressante pour révéler le contenu réel des fichiers, c'est en quelque sorte un mode d'accès de bas niveau au contenu.

En effet, qu'est-ce qui différencie un fichier texte d'un fichier binaire ? La réponse est simplement : le contenu ou l'interprétation de celui-ci. Dans les deux cas, un fichier possède des attributs qui caractérisent son nom, son extension, sa taille, sa date de création, etc.

Un fichier texte contient, tout comme son homologue le fichier binaire, une suite d'octets possédant une certaine structure.

Essayons d'ouvrir en mode brut un fichier texte Unicode, mais auparavant nous allons créer un nouveau fichier :

```
PS > 'PowerShell' > test.txt
PS > Get-Content test.txt -Encoding byte

255 254 80 0 111 0 119 0 101 0 114 0 83 0 104 0 101 0 108 0 108 0 13 0 10 0
```



Les octets s'affichent en réalité verticalement, mais pour faciliter la lecture et la compréhension de l'exemple nous les avons retranscrits horizontalement.

Un œil averti avec les fichiers textes ASCII remarquerait les deux choses suivantes :

- Le fichier débute par deux octets bizarres : 255 et 254.
- Tous les caractères sont codés sur deux octets dont l'un des deux vaut zéro.

Vous remarquerez également la présence des octets 13 et 10 en fin de ligne correspondant à **CR** et **LF** (voir plus haut dans ce chapitre).

La présence des octets 255 et 254 s'explique par le fait que tout fichier Unicode commence par un en-tête dont la longueur varie entre 2 et 4 octets. Cela diffère selon le codage Unicode choisi (UTF8, UTF16, UTF32).

Dans le cas présent, 255 254 (FF FE en notation hexadécimale) signifie que nous avons affaire à un fichier **UTF16 Little Endian**.

La présence des zéros s'explique car dans un fichier UTF16 tous les caractères sont codés sur deux octets.

Exemple :

Déterminer le type d'encodage d'un fichier.

La question que nous nous posions déjà depuis quelques pages, à savoir : « comment reconnaître le type d'encodage d'un fichier texte ? » a enfin trouvé sa réponse dans l'exemple précédent. Les premiers octets d'un fichier texte nous donnent son encodage.

Réalisons donc un petit script utilitaire qui nous dira de quel type est l'encodage d'un fichier à partir de ses premiers octets.

```
# Get-FileTypeEncoding.ps1
```

```

param ([string]$path=$(throw 'Chemin non trouvé'))

# définition des variables et constantes
$ANSI=0
Set-Variable -Name UTF8      -Value 'EFBBBF' -Option constant
Set-Variable -Name UTF16LE   -Value 'FFFE'   -Option constant
Set-Variable -Name UTF16BE   -Value 'FEFF'   -Option constant
Set-Variable -Name UTF32LE   -Value 'FFFE0000' -Option constant
Set-Variable -Name UTF32BE   -Value '0000FEFF' -Option constant

$fic = Get-Content -Path $path -Encoding byte -TotalCount 4
# Mise en forme des octets lus sur 2 caractères et conversion hexadécimale
# ex : 0 -> 00, ou 10 -> 0A au lieu de A
# et concaténation des octets dans une chaîne pour effectuer la comparaison
[string]$strLue = [string]('{0:x}' -f $fic[0]).PadLeft(2, '0') +
                  [string]('{0:x}' -f $fic[1]).PadLeft(2, '0') +
                  [string]('{0:x}' -f $fic[2]).PadLeft(2, '0') +
                  [string]('{0:x}' -f $fic[3]).PadLeft(2, '0')
Switch -regex ($strLue){
    "^$UTF32LE" {write-host 'Unicode UTF32LE'; break}
    "^$UTF32BE" {write-host 'Unicode UTF32BE'; break}
    "^$UTF8"    {write-host 'Unicode UTF8'; break}
    "^$UTF16LE" {write-host 'Unicode UTF16LE'; break}
    "^$UTF16BE" {write-host 'Unicode UTF16BE'; break}
    default
    {
        # Recherche d'un octet dont la valeur est > 127
        $fic = Get-Content -Path $path -Encoding byte
        for ($i=0; $i -lt (gci $path).Length; $i++){
            if ([char]$fic[$i] -gt 127){
                $ANSI=1
                break
            }
            else {
                $ANSI=0
            }
        } #fin for
        if ($ANSI -eq 1){
            Write-Host 'Fichier ANSI'
        }
        else{
            Write-Host 'Fichier ASCII'
        }
    } #fin default
} #fin switch

```

Ce script lit les quatre premiers octets du fichier, les met en forme et les compare à la signature Unicode pour déterminer le type d'encodage. Si aucune signature n'a été trouvée, c'est que le fichier est soit de type ASCII pur (caractères US de 0 à 127), soit de type ANSI (ASCII étendu, soit ASCII + page de codes pour gérer les caractères accentués).



Information de dernière minute : en explorant en profondeur les classes du Framework .NET (que vous découvrirez dans le chapitre .NET) nous avons découvert qu'il existait une classe qui permettait de déterminer le type d'encodage d'un fichier !

Exemple :

```

PS > $sr = new-object system.io.streamreader c:\temp\monFichier.txt
PS > $sr.CurrentEncoding

BodyName      : utf-8
EncodingName   : Unicode (UTF-8)
HeaderName    : utf-8
WebName       : utf-8
WindowsCodePage : 1200
IsBrowserDisplay : True
IsBrowserSave  : True

```

```

IsMailNewsDisplay : True
IsMailNewsSave    : True
IsSingleByte      : False
EncoderFallback   : System.Text.EncodingReplacementFallback
DecoderFallback   : System.Text.DecoderReplacementFallback
IsReadOnly        : True
CodePage          : 65001

```



Cela nous simplifiera grandement la tâche. Voici la preuve qu'en prenant le temps de fouiller un peu dans le Framework .NET on peut largement gagner du temps ! L'exemple reste néanmoins intéressant, car vous en saurez finalement un peu plus sur l'encodage Unicode.

3. Recherche de contenu avec Select-String

Grâce à **Select-String** nous allons pouvoir passer en revue le contenu d'une variable de type chaîne, d'un fichier, ou d'un grand nombre de fichiers à la recherche d'une chaîne de caractères sous forme d'expression régulière. Les "Unixiens" connaissant la commande **Grep** ne seront pas trop dépayés.

Voici les paramètres de **Select-String** (les paramètres signalés d'une étoile ne sont disponibles qu'avec PowerShell v2) :

Paramètres	Description
Pattern <String[]>	Chaîne ou expression régulière à rechercher.
Path <String[]>	Cible de la recherche : chaîne(s) ou fichier(s).
InputObject <PSObject>	Accepte un objet comme entrée.
Include <String[]>	Récupère uniquement les éléments spécifiés.
Exclude <String[]>	Omet les éléments spécifiés.
SimpleMatch <Switch>	Spécifie qu'une correspondance simple, plutôt qu'une correspondance d'expression régulière, doit être utilisée.
CaseSensitive <Switch>	Rend les correspondances sensibles à la casse.
Quiet <Switch>	Remplace le résultat de la commande par une valeur booléenne.
List <Switch>	Spécifie qu'une seule correspondance doit être retournée pour chaque fichier d'entrée.
AllMatches (*) <Switch>	Recherche plusieurs correspondances dans chaque ligne de texte. Sans ce paramètre, Select-String recherche uniquement la première correspondance dans chaque ligne de texte.
Context (*) <Int32>	Permet de sélectionner un nombre spécifique de lignes avant et après la ligne contenant la correspondance (permettant ainsi de voir le contenu recherché dans son contexte).
Encoding (*) <String>	Indique l'encodage du flux texte auquel Select-String doit s'appliquer. Les valeurs peuvent être : UTF7, UTF8, UTF32, Ascii, Unicode, BigIndian, Default ou OEM.
NotMatch (*) <Switch>	Indique quel modèle la recherche ne retourne pas. Ce paramètre est très utile pour réaliser une recherche inversée (en ne sélectionnant pas les lignes basée sur le modèle). Équivalent à Grep -v.



Les caractères accentués ne sont pas pris correctement en compte dans les recherches à l'intérieur des

Exemple :

Recherche simple.

```
PS > select-string -Path c:\temp\*.txt -Pattern 'fourmi'

C:\temp\CigaleFourmi.txt:8:Chez la fourmi sa voisine,
C:\temp\CigaleFourmi.txt:15:La fourmi n'est pas prêteuse ;
C:\temp\fourmisUtiles.txt:1:Les fourmis sont très utiles.
```

Dans cet exemple, nous recherchons la chaîne « fourmi » parmi tous les fichiers textes du répertoire **c:\temp**.

Nous obtenons en retour le nom des fichiers (ou du fichier s'il n'y en avait eu qu'un seul) qui contiennent la chaîne recherchée. Les valeurs 8, 15 et 1 correspondent au numéro de la ligne dans le fichier où une occurrence a été trouvée.

Parfois lorsque les résultats sont nombreux, il est intéressant d'utiliser le commutateur **-List** pour spécifier à la commande de ne retourner que le premier résultat trouvé par fichier.

Regardons quel serait le résultat avec **-List** :

```
PS > select-string -Path c:\temp\*.txt -Pattern 'fourmi' -List

C:\temp\CigaleFourmi.txt:8:Chez la fourmi sa voisine,
C:\temp\fourmisUtiles.txt:1:Les fourmis sont très utiles.
```

Les résultats obtenus sont de type **Microsoft.PowerShell.Commands.MatchInfo**. Ainsi, il est possible d'obtenir et de manipuler un certain nombre d'informations complémentaires en passant par une variable intermédiaire, comme ceci :

```
PS > $var = Select-String -Path c:\temp\*.txt -Pattern 'fourmi'
PS > $var | Get-Member -MemberType property

TypeName: Microsoft.PowerShell.Commands.MatchInfo

Name      MemberType Definition
-----
Context    Property  Microsoft.PowerShell.Commands.MatchInfoContext Context {get;set;}
Filename   Property  System.String Filename {get;}
IgnoreCase Property  System.Boolean IgnoreCase {get;set;}
Line       Property  System.String Line {get;set;}
LineNumber Property  System.Int32 LineNumber {get;set;}
Matches    Property  System.Text.RegularExpressions.Match[] Matches {get;set;}
Path       Property  System.String Path {get;set;}
Pattern     Property  System.String Pattern {get;set;}
```

À présent, essayons de forcer un affichage sous forme de liste :

```
PS > $var | Format-List

IgnoreCase : True
LineNumber : 8
Line       : Chez la fourmi sa voisine,
Filename   : CigaleFourmi.txt
Path       : C:\temp\CigaleFourmi.txt
Pattern    : fourmi
Context    :
Matches    : {Fourmi}

IgnoreCase : True
LineNumber : 15
Line       : La fourmi n'est pas prêteuse ;
Filename   : CigaleFourmi.txt
Path       : C:\temp\CigaleFourmi.txt
Pattern    : fourmi
Context    :
```



```
Matches      : {Fourmi}

IgnoreCase : True
LineNumber  : 1
Line       : Les fourmis sont très utiles.
Filename   : fourmisUtiles.txt
Path       : C:\temp\fourmisUtiles.txt
Pattern    : fourmi
Context    :
Matches    : {Fourmi}
```

Ainsi nous pouvons demander le numéro de ligne de la première occurrence :

```
PS > $var[0].Linenumber
8
```

Exemple :

Autre recherche simple.

Nous pouvons également utiliser **Select-String** en lui passant les données cibles au travers du pipe comme cela :

```
PS > Get-Item c:\temp\*.txt | Select-String -Pattern 'fourmi'
```

Les résultats obtenus seront les mêmes que dans l'exemple précédent.



Ne vous trompez pas ! Utilisez bien **Get-Item** ou **Get-ChildItem** et non pas **Get-Content** car bien que cela fonctionne à peu près, il peut y avoir des effets de bords. En effet, vous passeriez au pipeline le contenu des fichiers et non pas les fichiers eux-mêmes, et le contenu est en quelque sorte concaténé. Ce qui aurait pour conséquence de fausser la valeur de la propriété **LineNumber**.

Exemple :

```
PS > $var = Get-Content c:\temp\*.txt | Select-String -Pattern 'fourmi'
PS > $var | Format-List
```

```
IgnoreCase : True
LineNumber : 8
Line       : Chez la fourmi sa voisine,
Filename   : InputStream
Path       : InputStream
Pattern    : fourmi
Context    :
Matches    : {Fourmi}

IgnoreCase : True
LineNumber : 15
Line       : La fourmi n'est pas prêteuse ;
Filename   : InputStream
Path       : InputStream
Pattern    : fourmi
Context    :
Matches    : {Fourmi}

IgnoreCase : True
LineNumber : 23
Line       : Les fourmis sont très utiles.
Filename   : InputStream
Path       : InputStream
Pattern    : fourmi
Context    :
Matches    : {Fourmi}
```

Dans cet exemple, on notera que :

- Le nom de fichier a disparu des résultats pour être remplacé par un « *InputStream* » qui indique la provenance des données.
- Tout lien avec le fichier d'origine ayant disparu, le numéro de ligne est relatif à l'ensemble du flux, ce qui donne un résultat potentiellement erroné si l'on s'attend à avoir la position dans le fichier (voir la troisième et dernière occurrence ci-dessus).

Exemple 3 :

Recherche à base d'expression régulière.

```
PS > Get-item $pshome/fr-FR/*.txt | Select-String -Pattern 'item$'

C:\...\fr-FR\about_Alias.help.txt:159:          get-childitem
C:\...\fr-FR\about_Core_Commands.help.txt:23:    Get-ChildItem
C:\...\fr-FR\about_Core_Commands.help.txt:36:    APPLETS DE COMMANDE ITEM
C:\...\fr-FR\about_Core_Commands.help.txt:45:    Set-Item
C:\...\fr-FR\about_Environment_Variable.help.txt:35: get-childitem
C:\...\fr-FR\about_Environment_Variable.help.txt:100: get-childitem
C:\...\fr-FR\about_Parameter.help.txt:35:        help Get-ChildItem
C:\...\fr-FR\about_Provider.help.txt:137:        get-childitem
C:\...\fr-FR\about_Special_Characters.help.txt:46: $a = Get-ChildItem
C:\...\fr-FR\about_Wildcard.help.txt:82:         help Get-ChildItem
```

Cette ligne de commandes va explorer tous les fichiers dont l'extension est « *.txt* » à la recherche d'une chaîne se terminant par « *item* ».



L'exemple précédent reposait également sur une expression régulière. Simplement, sa syntaxe ne le distinguait pas d'une expression littérale. Il faut employer le paramètre **-SimpleMatch** pour que **Select-String** fasse une recherche sur une expression littérale plutôt que sur une expression régulière.

Exemple :

Recherche dont le résultat est un booléen.

```
PS > Select-String C:\temp\CigaleFourmi.txt -Pattern 'fourmi' -Quiet
True

PS > Select-String C:\temp\CigaleFourmi.txt -Pattern 'elephant' -Quiet
False
```

Exemple :

Recherche d'une chaîne en affichant son contexte (2 lignes avant et 2 lignes après).

```
PS > Select-String Cigalefourmi.txt -Pattern 'Août' -Context 2

Cigalefourmi.txt:11:Jusqu'à la saison nouvelle
Cigalefourmi.txt:12:"Je vous paierai, lui dit-elle,
Cigalefourmi.txt:13:Avant l'août, foi d'animal,
Cigalefourmi.txt:14:Intérêt et principal."
Cigalefourmi.txt:15:La fourmi n'est pas prêteuse ;
```

Pour être rigoureux, nous aurions dû ajouter le paramètre **-SimpleMatch** afin de préciser que notre recherche porte sur une chaîne et non pas sur une expression régulière. Cela fonctionne correctement car il n'y a pas de caractères spéciaux dans notre chaîne de recherche.

4. Gestion des fichiers CSV : Export-CSV / Import-CSV

Les fichiers CSV (*Comma Separated Values*) sont des fichiers textes dont les valeurs sont séparées par des virgules. Généralement, la première ligne de ces fichiers est l'en-tête. Celle-ci comprend le nom de chaque colonne de données,

et les valeurs qui la composent sont elles aussi séparées par des virgules.

Voici un exemple de fichier CSV :

```
Sexe,Prenom,Annee_de_naissance
M,Edouard,1982
M,Joe,1974
F,Eléonore,2004
```

PowerShell comprend un jeu de deux commandelettes pour gérer ces fichiers : **Export-CSV** pour créer un fichier, **Import-CSV** pour le relire.

Voici les différents paramètres de **Export-CSV** (les paramètres signalés d'une étoile ne sont disponibles qu'avec PowerShell v2) :

Paramètres	Description
Path <String>	Chemin du fichier de destination.
InputObject <PSObject>	Accepte un objet comme entrée.
Force <Switch>	Remplace le fichier spécifié si destination déjà existante.
Encoding <String>	Type d'encodage du fichier à créer (cf. Out-File pour la liste des valeurs possibles). ASCII est le type par défaut.
NoTypeInfoInformation <Switch>	Par défaut, un en-tête contenant le type des données est écrite. Si ce commutateur est spécifié, cet en-tête ne sera pas écrite.
NoClobber <Switch>	Ne pas écraser le fichier s'il existe déjà.
Delimiter (*) <Char>	Très utile, ce paramètre permet de spécifier un caractère délimiteur pour séparer les valeurs de propriété. La valeur par défaut est une virgule (,)
UseCulture (*)<Switch>	En lieu et place du paramètre Delimiter, vous pouvez également utiliser UseCulture. En spécifiant une culture spécifique, PowerShell adaptera le délimiteur (le délimiteur pour la culture fr-FR est le point-virgule). Pour le vérifier, essayez : (Get-Culture).TextInfo.ListSeparator

Et voici celui de **Import-CSV** :

Paramètres	Description
Path <String[]>	Chemin du fichier source.
Delimiter (*) <Char>	Très utile, ce paramètre permet de spécifier un caractère délimiteur pour séparer les valeurs de propriété. La valeur par défaut est une virgule (,)
UseCulture (*)<Switch>	En lieu et place du paramètre Delimiter, vous pouvez également utiliser UseCulture. En spécifiant une culture spécifique, PowerShell adaptera le délimiteur (le délimiteur pour la culture fr-FR est le point-virgule). Pour le vérifier, essayez : (Get-Culture).TextInfo.ListSeparator
Header (*) <String[]>	Permet de spécifier une autre ligne d'en-tête de colonne pour le fichier importé

Exemple : Export-CSV

```
PS > Get-Eventlog system -Newest 5 |
    Select-Object TimeGenerated,EntryType,Source,EventID |
    Export-Csv c:\temp\EventLog.csv -Encoding Unicode
PS > Get-Content c:\temp\EventLog.csv

#TYPE System.Management.Automation.PSCustomObject
```

```
TimeGenerated,EntryType,Source,EventID
"20/09/2009 12:31:29","Information","Service Control Manager","7036"
"20/09/2009 12:21:29","Information","Service Control Manager","7036"
"20/09/2009 12:00:01","Information","EventLog","6013"
"20/09/2009 11:50:38","Information","VPCNetS2","12"
"20/09/2009 11:50:38","Information","VPCNetS2","5"
```

Nous venons de créer un fichier Unicode nommé **EventLog.csv**. Il contient les propriétés **timeGenerated,EntryType,source,EventID** d'un objet de type journal des événements. Veuillez noter que la première ligne du fichier commence par **#TYPE** suivi du type de l'objet contenu dans notre fichier ; autrement dit il s'agit du type généré par **Get-EventLog**.



Faites attention, car par défaut cette commande génère des fichiers de type ASCII.

Exemple : Import-CSV

```
PS > $journal = Import-Csv c:\temp\EventLog.csv
PS > $journal
```

TimeGenerated	EntryType	Source	EventID
-----	-----	-----	-----
20/09/2009 12:31:29	Information	Service Control Manager	7036
20/09/2009 12:21:29	Information	Service Control Manager	7036
20/09/2009 12:00:01	Information	EventLog	6013
20/09/2009 11:50:38	Information	VPCNetS2	12
20/09/2009 11:50:38	Information	VPCNetS2	5

À présent, observons les propriétés et méthodes de **\$journal** :

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

TypeName: CSV:System.Management.Automation.PSCustomObject

Name	MemberType	Definition
----	-----	-----
Equals	Method	System.Boolean Equals(Object obj)
GetHashCode	Method	System.Int32 GetHashCode()
GetType	Method	System.Type GetType()
ToString	Method	System.String ToString()
EntryType	NoteProperty	System.String EntryType=Information
EventID	NoteProperty	System.String EventID=1103
Source	NoteProperty	System.String Source=Dhcp
TimeGenerated	NoteProperty	System.String TimeGenerated=20/09/2009 12:31:29

Nous pouvons nous apercevoir que nous avons des propriétés qui correspondent au nom de notre ligne d'en-tête ; ce qui va être fort utile pour en récupérer les valeurs ! Par exemple :

```
PS > $journal[0].EventID
7036
```

Exemple 2 : Export-CSV et Import-CSV

Imaginons à présent que vous soyez confronté à la recherche de résultats dans un fichier .csv puis à leurs modifications. Prenons comme exemple le fichier .csv suivant :

```
Nom,Prenom,Domaine,Derniere_Connexion
Lemesle,Robin,powershell-scripting.com,20/09/2009
Petitjean,Arnaud,powershell-scripting.com,21/09/2009
Teixeira,Jessica,,20/09/2009
```

Dans ce fichier, trois personnes sont identifiées parmi elles, une n'est pas identifiée comme appartenant au domaine powershell-scripting.com.

Dans un premier temps, importons le fichier.

```
PS > $utilisateurs = Import-Csv ./utilisateurs.csv
PS > $utilisateurs
```

Nom	Prenom	Domaine	Derniere_Connexion
---	-----	-----	-----
Lemesle	Robin	powershell-scripting.com	20/09/2009
Petitjean	Arnaud	powershell-scripting.com	21/09/2009
Teixeira	Jessica		20/09/2009

```
PS > $utilisateurs[0]
```

Nom	Prenom	Domaine	Derniere_Connexion
---	-----	-----	-----
Lemesle	Robin	powershell-scripting.com	20/09/2009

Comme on peut le voir ci-dessus, la variable \$utilisateurs se comporte comme un tableau dans lequel chaque ligne correspond à un numéro d'index. De plus chaque objet défini dans le tableau dispose des propriétés Nom,Prenom,Domaine,Derniere_Connexion, les noms des colonnes de notre fichier CSV.

```
PS > $utilisateurs[0].Nom
Lemesle

PS > $utilisateurs[0].Domaine
powershell-scripting.com
```

Bien entendu, le tableau peut être parcouru avec une boucle et chacune des valeurs est modifiable, c'est le cas ci-dessous. Pour chaque utilisateur, si un domaine n'est pas spécifié alors on lui ajoute la valeur PowerShell-scripting.com.

```
PS > Foreach($utilisateur in $utilisateurs){
if(($utilisateur.Domaine) -eq ''){
    $utilisateur.Domaine = 'PowerShell-scripting.com'
    Write-Host "l'utilisateur $($utilisateur.Nom) à été ajouté au domaine"
}
}
l'utilisateur Teixeira à été ajouté au domaine

PS > $utilisateurs
```

Nom	Prenom	Domaine	Derniere_Connexion
---	-----	-----	-----
Lemesle	Robin	powershell-scripting.com	20/09/2009
Petitjean	Arnaud	powershell-scripting.com	21/09/2009
Teixeira	Jessica	powershell-scripting.com	20/09/2009

Enfin, pour prendre en compte les changements apportés, l'enregistrement de la variable \$utilisateurs dans le fichier utilisateurs.csv s'effectue avec la commande Export-Csv.

```
PS > $utilisateurs | Export-Csv utilisateurs.csv -Encoding unicode
```



Si vous préférez éditer ce genre de fichiers avec Microsoft Excel plutôt qu'avec un éditeur de textes, nous vous recommandons de forcer le délimiteur à la valeur point-virgule, soit en spécifiant le paramètre -Delimiter ';', soit en ajoutant le switch -UseCulture. Ainsi, lorsque vous double cliquerez sur le fichier CSV, Excel devrait automatiquement le convertir et l'afficher.

5. Gestion des fichiers XML : Import-Clixml / Export-Clixml

XML (*Extensible Markup Language*) est un langage basé sur une hiérarchisation des données sous forme de balise. Pour savoir à quoi ressemble du XML, le mieux est certainement d'en voir un exemple.

```
<Livre>
  <Titre>Windows PowerShell</Titre>
  <SousTitre>Guide d'administration de référene pour l'administration
```

```

système</SousTitre>
  <Auteur>
    <Nom>Arnaud Petitjean</Nom>
    <AnnéeDeNaissance>1974</AnnéeDeNaissance>
    <Distinction>MVP PowerShell</Distinction>
  </Auteur>

  <Auteur>
    <Nom>Robin Lemesle
    </Nom>
    <AnnéeDeNaissance>1985</AnnéeDeNaissance>
    <Distinction>MVP PowerShell</Distinction>
  </Auteur>
  <Chapitre>
    <Nom>Introduction</Nom>
    <NombrePage>100</NombrePage>
  </Chapitre>

  <Chapitre>
    <Nom>A la decouverte de PowerShell</Nom>
    <NombrePage>120</NombrePage>
  </Chapitre>
</Livre>

```

Pour connaître l'ensemble des commandes liées à l'utilisation de fichier XML, tapez la commande suivante :

```
PS > Get-Command -Type cmdlet *XML*
```

CommandType	Name	Definition
Cmdlet	ConvertTo-Xml	ConvertTo-Xml [-Input0...
Cmdlet	Export-Clixml	Export-Clixml [-Path]...
Cmdlet	Import-Clixml	Import-Clixml [-Path]...
Cmdlet	Select-Xml	Select-Xml [-XPath] <S...

Commande	Description
ConvertTo-Xml	Cette commandelette permet de convertir des données au format XML
Export-Clixml	Réalise la même chose de ConvertTo-Xml mais permet aussi de stocker le résultat dans un fichier qui pourra ensuite être lu par Import-Clixml.
Import-Clixml	Comme son nom le laisse entendre, cette commande permet d'importer dans une variable le contenu de d'un fichier XML. Mais pas n'importe qu'elle fichier XML. En effet, la commande Import-Clixml ne permet pas d'importer des modèles génériques de fichiers XML, mais seulement les fichiers XML générés par PowerShell par la commande Export-Clixml.
Select-Xml	Permet de réaliser des requêtes au sein de données XML.

Comme nous le disions précédemment, la commande Import-Clixml permet seulement l'importation de fichiers XML générés par PowerShell, c'est-à-dire ceux générés par la commande Export-Clixml. Pour importer notre fichier, nous allons devoir réaliser un transtypage (cf. chapitre Fondamentaux). Le fichier XML importé n'est autre que celui présenté ci-avant qui est contenu dans le fichier Livre.xml.

```

PS > $Livre = [xml](get-content Livre.xml)
PS > $Livre

Livre
-----
Livre

```

Maintenant que le fichier XML est importé, il est alors très simple de le parcourir, et ce en précisant chaque nœud choisi, exemples :

```
PS > $Livre.Livre
```

Titre	SousTitre	Auteur	Chapitre
Windows PowerShell	Guide d'admin...	{Auteur, Auteur}	{Chapitre, Chapitre}

```

PS > $Livre.Livre.Titre
Windows PowerShell

PS > $Livre.Livre.Titre
Windows PowerShell

PS > $Livre.Livre.auteur
Nom          AnnéeDeNaissance  Distinction
---          -
Arnaud Petitjean  1974             MVP PowerShell
Robin Lemesle    1985             MVP PowerShell

```

Des modifications peuvent également être effectuées, pour cela il suffit d'indiquer quelle nouvelle valeur doit prendre un nœud en question.

```

PS > $Livre.Livre.Titre = 'Le livre de PowerShell'
PS > $Livre.Livre

```

Titre	SousTitre	Auteur	Chapitre
Le livre de PowerShell	Guide d'admin...	{Auteur, Auteur}	{Chapitre, Chapitre}

Nous venons de voir comment explorer des fichiers XML personnels, ce qui est très pratique. Mais les commandes natives PowerShell à propos d'XML, sont principalement dédiées à un usage de stockage d'informations provenant et propre à PowerShell. Comme par exemple le stockage d'objets PowerShell. C'est ce que nous allons voir. Ce mécanisme est aussi appelé « sérialisation » / « désérialisation » d'objets.

Lors de récupération d'informations dans une session PowerShell, le seul moyen de les récupérer ultérieurement à travers une autre session PowerShell, consiste à stocker les informations dans un fichier. Seulement voilà, le stockage d'information dans un fichier, fait perdre toute l'interactivité liée à l'objet. Prenons le cas concret suivant. Imaginons que nous souhaitons sauvegarder les informations retournées par la commande `Get-Process` afin de les analyser plus tard. Une des méthodes qui peut venir à l'esprit consiste à stocker ces données dans un fichier.txt.

```
PS > Get-Process > Process.txt
```

Quelques temps plus tard, au moment choisi par l'utilisateur pour récupérer ses données, la commande `Get-Content` ne pourra fournir qu'une chaîne de caractères comme valeur de retour. Avec un objet de type `String` et non pas un tableau d'objet `Process` (comme le retourne la commande `Get-Process`). L'application des méthodes et l'accès aux propriétés de l'objet est tout simplement impossible.

```

PS > $Process = Get-Content Process.txt
PS > Foreach($Item in $Process){$Item.id}

<Vide>

```

C'est donc là qu'intervient la commande `Export-Clixml`. Avec elle, l'objet transmis sera stocké sous un format XML que PowerShell sait interpréter de façon à ce qu'il puisse « reconstruire » les données. Reprenons notre exemple :

```
PS > Get-Process | Export-Clixml Process.xml
```

Après le stockage de l'objet via `Export-Clixml`, son importation est réalisée avec la commande `Import-Clixml`. Et comme promis, une fois importé, les méthodes et propriétés de l'objet sont à nouveau utilisables.

```

PS > $process = Import-Clixml Process.xml
PS > Foreach($Item in $Process){$Item.id}

3900
2128
1652
2080
1612
884
2656

```

6. Export de données en tant que page HTML

Si la création de pages HTML vous tente pour présenter quelques rapports stratégiques ou autres, alors la commande **ConvertTo-HTML** est faite pour vous ! En effet, grâce à celle-ci la génération de pages Web devient presque un jeu d'enfant si l'on fait abstraction de la mise en forme.

Voici les paramètres disponibles de **ConvertTo-HTML** :

Paramètres	Description
Property <Object[]>	Propriétés de l'objet passé en paramètre à écrire dans la page HTML.
InputObject <PSObject>	Accepte un objet comme entrée.
Body <String[]>	Spécifie le texte à inclure dans l'élément <body>.
Head <String[]>	Spécifie le texte à inclure dans l'élément <head>.
Title <String>	Spécifie le texte à inclure dans l'élément <title>.

Un peu à la manière de la commande **Export-CSV**, le nom des propriétés servira de titre pour chaque colonne du fichier HTML.

Exemple :

Liste des services du système.

```
PS > Get-Service |
  ConvertTo-HTML -Property name, displayname, status -Title 'Services du système' |
  Out-File Services.htm
```

Cet exemple nous permet de créer une page HTML qui contient la liste des services, leurs noms ainsi que leurs états. Le paramètre **-Title** a été spécifié afin que la fenêtre ait un titre autre que celui par défaut. Le tout est passé à **Out-File** qui créera le fichier **Services.htm**.

Si nous ne spécifions pas de propriétés particulières, toutes celles de l'objet seront écrites ; le paramètre **-Property** joue donc en quelque sorte un rôle de filtre.

D'autre part, à la différence d'**Export-CSV**, **ConvertTo-HTML** a besoin pour fonctionner pleinement qu'on lui adjoigne une commande pour écrire le flux texte de génération de la page dans un fichier. Par conséquent, n'oubliez pas de faire attention au type d'encodage du fichier résultant.

name	displayname	status
AeLookupSvc	Expérience d'application	Running
ALG	Service de la passerelle de la couche Application	Stopped
Appinfo	Informations d'application	Running
AppMgmt	Gestion d'applications	Stopped
aswUpdSv	avast! iAVS4 Control Service	Running
AudioEndpointBuilder	Générateur de points de terminaison du service Audio Windows	Running
Audiosrv	Audio Windows	Running
avast! Antivirus	avast! Antivirus	Running
avast! Mail Scanner	avast! Mail Scanner	Running
avast! Web Scanner	avast! Web Scanner	Running
BFE	Moteur de filtrage de base	Running
BITS	Service de transfert intelligent en arrière-plan	Running
Browser	Explorateur d'ordinateurs	Stopped
CertPropSvc	Propagation du certificat	Stopped
clr_optimization_v2.0.50727_32	Microsoft .NET Framework NGEN v2.0.50727_X86	Stopped
COMSysApp	Application système COM+	Stopped
CryptSvc	Services de chiffrement	Running
CscService	Fichiers hors connexion	Running
DcomLaunch	Lanceur de processus serveur DCOM	Running
DFSR	Réplication DFS	Stopped
Dhcp	Client DHCP	Running

Création d'une page HTML



Pour ouvrir ce fichier directement dans Internet Explorer, il vous suffit de taper la commande suivante :
`./services.htm` ou `Invoke-Item Services.htm`.



Comme l'extension `.htm` est connue de Windows, celui-ci ouvre le fichier avec l'application qui lui est associée (par défaut, Internet Explorer).

Grâce au paramètre **-Body**, nous pouvons spécifier du contenu supplémentaire qui apparaîtra dans le corps de la page, juste avant les données de l'objet.

Exemple :

Liste des services du système avec *BODY*.

```
PS >Get-Service |
  ConvertTo-HTML -Property name,displayname,status -Title 'Services
du système' `
  -body '<CENTER><H2>Etat des services du système</H2></CENTER>' |
  Out-File Services.htm
```

Services du système - Windows Internet Explorer

C:\Temp\Services.htm

Live Search

Services du système

Etat des services du système

name	displayname	status
AeLookupSvc	Expérience d'application	Running
ALG	Service de la passerelle de la couche Application	Stopped
Appinfo	Informations d'application	Running
AppMgmt	Gestion d'applications	Stopped
aswUpdSv	avast! iAVS4 Control Service	Running
AudioEndpointBuilder	Générateur de points de terminaison du service Audio Windows	Running
Audiosrv	Audio Windows	Running
avast! Antivirus	avast! Antivirus	Running
avast! Mail Scanner	avast! Mail Scanner	Running
avast! Web Scanner	avast! Web Scanner	Running
BFE	Moteur de filtrage de base	Running
BITS	Service de transfert intelligent en arrière-plan	Running
Browser	Explorateur d'ordinateurs	Stopped
CertPropSvc	Propagation du certificat	Stopped
clr_optimization_v2.0.50727_32	Microsoft .NET Framework NGEN v2.0.50727_X86	Stopped
COMSysApp	Application système COM+	Stopped
CryptSvc	Services de chiffrement	Running
CscService	Fichiers hors connexion	Running
DcomLaunch	Lanceur de processus serveur DCOM	Running

Terminé

Ordinateur | Mode protégé : désactivé

100%

Création d'une page HTML avec titre

Exemple :

Liste des services du système formatée avec CSS.

Encore plus fort, nous allons cette fois encadrer notre tableau grâce aux feuilles de style en cascade (*Cascading Style Sheets*). Pour ce faire, nous avons créé la feuille de style suivante, que nous avons nommé *Style.css* :

```
<style type='text/css'>
  table {
    border: medium solid #000000;
    border-collapse: collapse ;
  }
  td, th {
    border: thin solid #6495ed;
  }
</style>
```

Nous allons devoir inclure ce fichier dans l'élément **HEAD** de notre fichier HTML, comme ceci :

```
PS > $CSS = Get-Content Style.css
PS > Get-Service |
  ConvertTo-HTML -Property name,displayname,status -Title 'Services du système' `
  -Head $CSS -Body '<CENTER><H2>Etat des services du système</H2></CENTER>' |
  Out-File Services.htm
```

name	displayname	status
AeLookupSvc	Expérience d'application	Running
ALG	Service de la passerelle de la couche Application	Stopped
Appinfo	Informations d'application	Running
AppMgmt	Gestion d'applications	Stopped
aswUpdSv	avast! iAVS4 Control Service	Running
AudioEndpointBuilder	Générateur de points de terminaison du service Audio Windows	Running
Audiosrv	Audio Windows	Running
avast! Antivirus	avast! Antivirus	Running
avast! Mail Scanner	avast! Mail Scanner	Running
avast! Web Scanner	avast! Web Scanner	Running
BFE	Moteur de filtrage de base	Running
BITS	Service de transfert intelligent en arrière-plan	Running
Browser	Explorateur d'ordinateurs	Stopped
CertPropSvc	Propagation du certificat	Stopped
clr_optimization_v2.0.50727_32	Microsoft .NET Framework NGEN v2.0.50727_X86	Stopped
COMSysApp	Application système COM+	Stopped
CryptSvc	Services de chiffrement	Running
CscService	Fichiers hors connexion	Running
DcomLaunch	Lanceur de processus serveur DCOM	Running

Création d'une page HTML - Affichage d'un tableau

Un dernier exemple pour terminer avec cette commande pourrait être de mettre de la couleur pour chaque ligne de notre table. Nous pourrions ainsi différencier les services en cours d'exécution des autres.

Exemple :

Liste des services du système avec analyse du contenu.

```
PS > Get-Service |
ConvertTo-Html -Property name,displayname,status -Title 'Services du système' `
-Body '<CENTER><H2>Etat des services du système</H2></CENTER>' |
foreach {
    if($_ -match '<td>Running</td>')
    {
        $_ -replace '<tr>', '<tr bgcolor=#DDDDDD>'
    }
    elseif($_ -match '<td>Stopped</td>')
    {
        $_ -replace '<tr>', '<tr bgcolor= #6699FF>'
    }
    else
    {
        $_
    }
} | Out-File Services.htm
```

Dans cet exemple, lorsqu'un service est en marche, on remplace la balise **TR** (indiquant une ligne) par la même balise **TR** avec en plus l'instruction permettant d'afficher un fond de couleur (**bgcolor=#codeCouleur**).

Services du système - Windows Internet Explorer

C:\Temp\Services.htm

Live Search

Services du système

Etat des services du système

name	displayname	status
AeLookupSvc	Expérience d'application	Running
ALG	Service de la passerelle de la couche Application	Stopped
Appinfo	Informations d'application	Running
AppMgmt	Gestion d'applications	Stopped
aswUpdSv	avast! iAVS4 Control Service	Running
AudioEndpointBuilder	Générateur de points de terminaison du service Audio Windows	Running
Audiosrv	Audio Windows	Running
avast! Antivirus	avast! Antivirus	Running
avast! Mail Scanner	avast! Mail Scanner	Running
avast! Web Scanner	avast! Web Scanner	Running
BFE	Moteur de filtrage de base	Running
BITS	Service de transfert intelligent en arrière-plan	Running
Browser	Explorateur d'ordinateurs	Stopped
CertPropSvc	Propagation du certificat	Stopped
clr_optimization_v2.0.50727_32	Microsoft .NET Framework NGEN v2.0.50727_X86	Stopped
COMSysApp	Application système COM+	Stopped
CryptSvc	Services de chiffrement	Running
CscService	Fichiers hors connexion	Running
DcomLaunch	Lanceur de processus serveur DCOM	Running

Ordinateur | Mode protégé : désactivé 100%

Création d'une page HTML - Affichage d'un tableau (bis)

7. Export de données avec Out-GridView

Avec PowerShell v2, une nouvelle commandelette graphique a fait son apparition : il s'agit de **Out-GridView**. Grâce à elle nous allons pouvoir afficher des données de façon graphique (à l'image de ce que nous venons de faire précédemment avec une page HTML, mais sans effort) et de manière interactive. L'interactivité de la table se trouve dans la possibilité de cliquer sur le titre des colonnes afin d'effectuer un tri des données par ordre alphabétique. Une autre forme d'interactivité est une fonction de recherche intégrée à la table. Celle-ci est pratique lorsque les données sont nombreuses et que l'on en recherche une en particulier.

Exemple :

Liste des services en cours d'exécution.

```
PS > Get-Service | Out-GridView
```

Status	Name	DisplayName
Running	Appinfo	Informations d'application
Running	aswUpdSv	avast! iAVS4 Control Service
Running	AudioEndpointBuilder	Générateur de points de terminaison du service Audio Windows
Running	Audiosrv	Audio Windows
Running	avast! Antivirus	avast! Antivirus
Running	avast! Mail Scanner	avast! Mail Scanner
Running	avast! Web Scanner	avast! Web Scanner
Running	BFE	Moteur de filtrage de base
Running	bthserv	Service de prise en charge Bluetooth
Running	CryptSvc	Services de chiffrement
Running	CscService	Fichiers hors connexion
Running	DcomLaunch	Lanceur de processus serveur DCOM
Running	Dhcp	Client DHCP
Running	Dnscache	Client DNS
Running	DPS	Service de stratégie de diagnostic
Running	EapHost	Protocole EAP (Extensible Authentication Protocol)
Running	eventlog	Journal d'événements Windows
Running	EventSystem	Système d'événement COM+
Running	fdPHost	Hôte du fournisseur de découverte de fonctions
Running	FDResPub	Publication des ressources de découverte de fonctions
Running	FontCache	Service de cache de police Windows
Running	FontCache3.0.0.0	Cache de police de Windows Presentation Foundation 3.0.0.0
Running	gensev	Client de stratégie de groupe

Utilisation de Out-GridView pour afficher la liste des services

Nous avons ici cliqué sur la colonne Status afin de trier les résultats selon l'état des services (Running, Stopped, etc.). Veuillez noter qu'au-dessus des colonnes se trouve la zone de recherche dans laquelle le texte par défaut est « filter ».

Exemple 2 :

Affichage graphique du contenu d'un fichier CSV

```
PS > Import-Csv utilisateurs.csv | Out-GridView
```

import-csv utilisateurs.csv | Out-GridView

Filtrer

+ Ajouter des critères ▼

Nom	Prenom	Domaine	DerniereConnection
Lemesle	Robin	powershell-scripting.com	20/09/2009
Petitjean	Arnaud	powershell-scripting.com	21/09/2009
Teixeira	Jessica	PowerShell-scripting.com	20/09/2009

Utilisation de Out-GridView avec un fichier CSV

Les dates

Avec PowerShell, l'obtention de la date et de l'heure se fait avec la commandelette **Get-Date**. Alors certes, un simple **Get-Date** dans la console vous affiche l'heure et la date actuelles, mais cette date peut se décliner en un bon nombre de formats (cf. Les dates - Les formats, de ce chapitre).

Une variable contenant une date est de type **DateTime**. Pour le vérifier par vous-même, tapez la commande suivante :

```
PS > (Get-Date).GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     DateTime          System.ValueType
```

Les objets de type **DateTime** cachent de nombreuses méthodes intéressantes comme la méthode **IsDaylightSavingTime**, qui vous indique si l'heure actuelle est ajustée pour l'heure d'été ou l'heure d'hiver.

Pour se rendre compte des possibilités d'actions sur les dates, le mieux est encore de faire un **Get-Member** sur l'objet retourné par **Get-Date** :

```
PS > Get-Date | Get-Member
```

Lorsque l'on parle de date ou de temps, il est nécessaire de définir ce que l'on appelle une unité de temps. Et l'unité la plus basse qui soit dans le système est appelée un « tick ». Un tick est une unité de mesure du temps. Elle correspond à un battement de cœur de l'ordinateur, c'est-à-dire à une période du Timer (le Timer est un composant électronique qui gère le temps). Cette valeur vaut actuellement dix millièmes de secondes.



Pour connaître le nombre de ticks présents en une seconde, tapez la commande suivante : PS > \$((Get-Date).ticks) - \$((Get-Date).AddSeconds(-1).ticks) 9990000

Soit à peu près 10 millions moyennant un temps de traitement de la commande.

1. Méthodes de manipulation des objets DateTime

Grâce à la commandelette **Get-Member** appliquée à un objet de type **DateTime** vous avez pu apercevoir une liste considérable de méthodes. Le tableau suivant reprend les méthodes les plus courantes et vous en donne une brève description.

Méthode	Description
Add et toute la famille des Add-* AddDays AddHours AddMilliseconds AddMinutes AddMonths AddSeconds AddTicks AddYears	Ajoute ou retranche une ou plusieurs unités de temps à l'objet date. Ajoute si la valeur passée en argument est positive, retranche si la valeur passée est négative (cf. Les dates - Manipulation des dates, de ce chapitre).
CompareTo	Compare une date à une autre. Les valeurs retournées sont : -1 : si la date est antérieure à celle à laquelle on la compare. 1 : si elle est postérieure.

	0 : si elles sont égales.
Equals	Retourne un indicateur booléen de comparaison. True : si les deux dates sont identiques. False : dans le cas contraire.
GetDateTimeFormats	Retourne tous les formats disponibles pour l'objet DateTime.
GetHashCode	Retourne le code de hachage de la variable.
GetType	Retourne le type de la variable. Dans le cas d'une date, il s'agit du type DateTime.
GetTypeCode	Retourne le code associé au type.
La famille des Get-* Get_Date Get_Day Get_DayOfWeek Get_DayOfYear Get_HourGet_Millisecond Get_Minute Get_Month Get_Second Get_Ticks Get_TimeOfDay Get_Year	Les méthodes Get-* retournent le paramètre de la date en question. Exemple : la méthode Get_DayOfWeek retourne une variable contenant le jour de la semaine correspondant.
Get_Kind	Retourne le type de variable.
IsDayLightSavingTime	Retourne une valeur booléenne qui indique si l'heure actuelle est ajustée pour l'heure d'été ou l'heure d'hiver.
Subtract	Soustrait une date de l'objet.
ToBinary	Retourne la valeur de la date en binaire.
ToFileTime	Retourne la valeur de l'objet DateTime en cours, en heure de fichier Windows.
ToFileTimeUtc	Retourne la valeur de l'objet DateTime en cours, en heure de fichier Windows (Heure Universelle).
ToLocalTime	Retourne la valeur de l'objet DateTime en cours, en heure locale.
ToLongDateString	Retourne une chaîne de caractères contenant la date au format long.
ToLongTimeString	Retourne une chaîne de caractères contenant l'heure au format long.
ToOADate	Retourne la date au format OLE (<i>Object Linking and Embedding</i>) automation (nombre flottant). Le format OLE automation correspond au nombre de jours depuis le 30 décembre 1899 à minuit.
ToShortDateString	Retourne une chaîne de caractères contenant la date au format

	court.
ToShortTimeString	Retourne une chaîne de caractères contenant l'heure au format court.
ToString	Retourne une chaîne de caractères contenant la date et l'heure au format standard.
ToUniversalTime	Retourne la date et l'heure au format standard.

Exemple :

Récupération des minutes dans l'heure actuelle.

```
PS > Get-Date
jeudi 8 octobre 2009 22:36:16

PS > (Get-Date).Get_minute()
36
```

2. Les formats

Choisir un format n'est pas forcément chose aisée, surtout quand il existe une soixantaine de formats dits « standards ».

Et oui, avec une seule date il existe de très nombreuses façons de l'écrire différemment. Pour vous en rendre compte, essayez la commande suivante :

```
PS > (Get-Date).GetDateTimeFormats() | Sort-Object -Unique
```

```
08.10.09                |22:40
08.10.09 22 h 40        |22:40:29
08.10.09 22.40          |8 oct. 09
08.10.09 22:40          |8 oct. 09 20 h 40
08.10.09 22:40:29       |8 oct. 09 20.40
08/10/09                |8 oct. 09 20:40:29
08/10/09 22 h 40        |8 oct. 09 22 h 40
08/10/09 22.40          |8 oct. 09 22.40
08/10/09 22:40          |8 oct. 09 22:40
08/10/09 22:40:29       |8 oct. 09 22:40:29
08/10/2009              |8 octobre
08/10/2009 22 h 40      |8 octobre 2009
08/10/2009 22.40        |8 octobre 2009 20 h 40
08/10/2009 22:40        |8 octobre 2009 20.40
08/10/2009 22:40:29     |8 octobre 2009 20:40:29
08-10-09                |8 octobre 2009 22 h 40
08-10-09 22 h 40        |8 octobre 2009 22.40
08-10-09 22.40          |8 octobre 2009 22:40
08-10-09 22:40          |8 octobre 2009 22:40:29
08-10-09 22:40:29       |jeudi 8 octobre 2009
2009-10-08              |jeudi 8 octobre 2009 20 h 40
2009-10-08 22 h 40      |jeudi 8 octobre 2009 20.40
2009-10-08 22.40        |jeudi 8 octobre 2009 20:40:29
2009-10-08 22:40        |jeudi 8 octobre 2009 22 h 40
2009-10-08 22:40:29     |jeudi 8 octobre 2009 22.40
2009-10-08 22:40:29Z    |jeudi 8 octobre 2009 22:40
2009-10-08T22:40:29     |jeudi 8 octobre 2009 22:40:29
2009-10-08T22:40:29.6675819+02:00 |octobre 2009
22 h 40                 |Thu, 08 Oct 2009 22:40:29 GMT
```

3. Les formats standard

Pour vous aider dans le choix du format, le tableau suivant liste les formats standard applicables aux valeurs **DateTime**.

Format	Description
d	Format date courte.
D	Format date longue.
f	Format date longue et heure abrégée.
F	Format date longue et heure complète.
g	Format date courte et heure abrégée.
G	Format date courte et heure complète.
m,M	Format mois et jour : " dd MMMM ".
r,R	Format date et heure basé sur la spécification de la RFC 1123.
s	Format date et heure triée.
t	Format heure abrégée.
T	Format heure complète.
u	Format date et heure universelle (indicateur de temps universel : "Z").
U	Format date longue et heure complète avec temps universel.
y,Y	Format année et mois.

Voici quelques exemples d'applications des différents formats.

Exemples :

Si vous souhaitez retourner une date au format standard tel que défini dans la RFC 1123, la commande sera la suivante :

```
PS > Get-Date -Format r
Sun, 20 Sep 2009 12:48:53 GMT
```

Si vous souhaitez retourner une date au format date courte et heure complète tel que défini dans la RFC 1123, la commande sera la suivante :

```
PS > Get-Date -Format G
20/09/2009 12:49:04
```

Si vous souhaitez retourner une date au format date longue et heure complète tel que défini dans la RFC 1123, la commande sera la suivante :

```
PS > Get-Date -Format F
dimanche 20 septembre 2009 12:49:30
```

4. Les formats personnalisés

Bien entendu l'affichage d'une date ne se limite pas aux formats standard. Des affichages personnalisés sont également possibles.

Et pour ce faire, vous devez utiliser le paramètre `-Format` associé à des spécificateurs de format. La différence avec les

formats standard énoncés précédemment, réside dans le fait que les formats personnalisés sont des éléments combinables dans une chaîne de caractères par exemple. Alors que les formats standard n'ont de sens que s'ils ne sont pas combinés.

Voici la liste non exhaustive des formats personnalisés :

Format	Description
d	Représentation du jour par un nombre compris entre : 1-31.
dd	Représentation du jour par un nombre compris entre : 01-31. La différence avec le format « d » est l'insertion d'un zéro non significatif pour les nombres allant de 1 à 9.
ddd	Représentation du jour sous la forme de son nom abrégé. Exemple : Lun., Mar., Mer., etc.
dddd	Représentation du jour sous la forme de son nom complet.
f	Représentation du chiffre le plus significatif de la fraction de seconde.
ff	Représentation des deux chiffres les plus significatifs de la fraction de seconde.
fff	Représentation des trois chiffres les plus significatifs de la fraction de seconde.
ffff	Représentation des quatre chiffres les plus significatifs de la fraction de seconde.
h	Représentation de l'heure par un nombre. Nombres compris entre : 1-12.
hh	Représentation de l'heure par un nombre avec insertion d'un zéro non significatif pour les nombres allant de 1 à 9. Nombres compris entre : 01-12.
H	Représentation de l'heure par un nombre. Nombres compris entre : 0-23.
HH	Représentation de l'heure par un nombre avec insertion d'un zéro non significatif pour les nombres allant de 0 à 9. Nombres compris entre : 00-23.
m	Représentation des minutes par un nombre. Nombres compris entre : 0-59.
mm	Représentation des minutes par un nombre avec insertion d'un zéro non significatif pour les nombres allant de 0 à 9. Nombres compris entre : 00-59.
M	Représentation du mois par un nombre. Nombres compris entre : 1-12.
MM	Représentation du mois par un nombre avec insertion d'un zéro non significatif pour les nombres allant de 1 à 9. Nombres compris entre : 01-12.
MMM	Représentation du mois sous la forme de son nom abrégé.
MMMM	Représentation du mois sous la forme de son nom complet.
y	Représentation de l'année sous la forme d'un nombre à deux chiffres, au plus. Si l'année comporte plus de deux chiffres, seuls les deux chiffres de poids faible apparaissent dans le résultat et si elle en comporte moins, seul le ou les chiffres (sans zéro significatif) apparaissent.
yy	Idem que ci-dessus à la différence près que si l'année comporte moins de deux chiffres, le nombre est rempli à l'aide de zéros non significatifs pour atteindre deux chiffres.
yyy	Représentation de l'année sous la forme d'un nombre à trois chiffres. Si l'année comporte plus de trois chiffres, seuls les trois chiffres de poids faible apparaissent dans le résultat. Si l'année comporte moins de trois chiffres, le nombre est rempli à l'aide de zéros non significatifs pour atteindre trois chiffres.

yyyy	Représentation de l'année sous la forme d'un nombre à quatre chiffres. Si l'année comporte plus de quatre chiffres, seuls les quatre chiffres de poids faible apparaissent dans le résultat. Si l'année comporte moins de quatre chiffres, le nombre est rempli à l'aide de zéros non significatifs pour atteindre quatre chiffres.
------	---



Pour obtenir la liste complète de ces spécificateurs de format, rendez-vous sur le site MSDN de Microsoft : [http://msdn2.microsoft.com/fr-fr/library/8kb3ddd4\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/8kb3ddd4(VS.80).aspx)

Exemple :

Dans ce premier exemple, nous souhaitons simplement afficher la date sous le format suivant :

```
<Nom du Jour><Numero du jour> <Mois> <Année> ---- <Heure>:<Minute>
:<Seconde>
```

```
PS > Get-Date -Format 'dddd dd MMMM yyyy ---- HH:mm:ss '
dimanche 20 septembre 2009 ---- 12:50:16
```

Exemple :

Imaginons que vous soyez amenés à générer des rapports dont le nom du fichier doit correspondre à la date à laquelle il a été généré.

Pour cela rien de plus facile...

```
PS > New-Item -Type file -Name "Rapport_$((Get-Date) -Format 'dd-MM-yyyy')).txt"
```

Résultat :

```
PS > New-Item -Type File -Name "Rapport_$((Get-Date) -Format 'dd-MM-yyyy')).txt"
Répertoire : C:\Temp

Mode                LastWriteTime         Length Name
----                -
-a- - -           20/09/2009      12:51             0 Rapport_20-09-2009.txt
```

Il existe un dernier mode d'affichage. Ce dernier s'appelle l'affichage en mode « Unix ».

Comme vous pouvez l'imaginer, ce mode récupère au format Unix les propriétés de l'objet **DateTime** que vous spécifiez. Voici l'essentiel des spécificateurs :

Format	Description
%m	Mois de l'année (01-12).
%d	Jour du mois (01-31).
%y	Année, uniquement les deux derniers chiffres (00-99).
%Y	Année sur quatre chiffres.
%D	Affichage au format mm/dd/yy.
%H	Heures (00-23).
%M	Minutes (00-59).
%S	Secondes (00-59).

%T	Heure au format HH :MM :SS.
%J	Jour de l'année (1-366).
%w	Jour de la semaine (0-6) avec Samedi = 0.
%a	Abréviation du jour (lun. , mar. , etc.).
%h	Abréviation du mois (Fev., Juil. , etc.).
%r	Heure au format HH :MM :SS avec HH (0-12).
%n	Nouvelle ligne.
%t	Tabulation.

Exemple :

Affichage au format Unix de la date actuelle.

```
PS > Get-Date -Uformat 'Nous sommes le %a %d %Y, et il est %T'
Nous sommes le dim. 20 2009, et il est 12:52:19
```

5. Manipulation des dates

a. Créer une date

Il existe plusieurs manières de créer une date en PowerShell. La plus courante consiste à utiliser la commande **Get-Date**. Utilisée sans paramètre, cette commande retourne la date et l'heure. Si nous désirons créer une variable **DateTime** contenant une date de notre choix, il nous faut la spécifier grâce aux paramètres : -Year, -Month, -Day, -Hour, -Minute, -Second.

Exemple :

Si nous souhaitons définir une variable qui contient la date du 10 février 2008, la commande sera la suivante :

```
PS > $Date = Get-Date -Year 2008 -Month 2 -Day 10
```

Notez que tout paramètre qui n'est pas précisé prend la valeur correspondante de la date du jour.

b. Modifier une date

Lors de l'exécution d'un script ou pour une application tierce nous pouvons être amenés à modifier une date donnée. Pour répondre à cela, il faut utiliser la famille des méthodes **Add***.

Les méthodes **Add** permettent d'ajouter un entier relatif de jours avec **AddDays**, d'heures avec **AddHours**, de millisecondes avec **AddMilliseconds**, de mois avec **AddMonth**, de secondes avec **AddSeconds**, d'années avec **AddYears** et de ticks avec **AddTicks**.



Les entiers relatifs sont l'ensemble des entiers (0,1,2,3,...) positifs et négatifs (0,-1,-2,-3,...).

Par exemple, pour savoir quel jour de la semaine sera le même jour qu'aujourd'hui mais dans un an, il suffit d'ajouter un an à la date du moment et de récupérer le jour de la semaine correspondant.

```
PS > $date = Get-Date
PS > $date.AddYears(1).DayOfWeek
Friday
```

De la même façon, il est facile de retrouver le jour de sa naissance.

Exemple :

```
PS > $date = [DateTime]'10/03/1974'  
PS > $date.DayOfWeek  
  
Saturday
```



Lorsque l'on spécifie une date comme dans l'exemple ci-dessus, le format attendu est le format anglo-saxon, à savoir : mois/jour/année.

c. Comparer des dates

Il existe plusieurs types de comparaison de dates, la comparaison la plus simple s'effectue avec la méthode **CompareTo**. Appliquée à la variable de type **DateTime**, cette méthode permet une comparaison rapide et renvoie les valeurs suivantes :

Valeur de retour	Description
-1	Si la date est antérieure à celle à laquelle on la compare.
1	Si elle est postérieure.
0	Si elles sont égales.

Exemple :

Comparaison de la date de deux fichiers.

*Pour cela, il suffit de récupérer une à une les dates de création et de les comparer avec la méthode **CompareTo**.*

```
PS > $Date_fichier_1 = (Get-item Fichier_1.txt).Get_CreationTime()  
PS > $Date_fichier_2 = (Get-item Fichier_2.txt).Get_CreationTime()  
PS > $Date_fichier_1.CompareTo($date_fichier_2)  
-1
```

La deuxième méthode consiste à calculer le temps écoulé entre deux dates de façon à pouvoir les comparer par la suite. Cette opération est rendue possible grâce à la commande **New-TimeSpan**. Pour plus d'informations sur celle-ci tapez : **help New-TimeSpan**.

Exemple :

Calcul du temps écoulé depuis votre naissance.

Pour déterminer le nombre de secondes qui se sont écoulées depuis votre naissance.

Il faut, dans un premier temps, calculer le temps écoulé grâce à la commande **New-TimeSpan**. Puis dans un second temps, on va transformer la valeur reçue par la commande **New-TimeSpan** en secondes.

```
PS > New-TimeSpan $(Get-Date -Year 1985 -Month 10 -Day 6  
-Hour 8 -Minute 30) $(Get-Date)  
  
Days           : 8750  
Hours          : 4  
Minutes        : 24  
Seconds        : 0  
Milliseconds   : 0  
Ticks          : 7560158400000000  
TotalDays      : 8750,183333333333  
TotalHours     : 210004,4
```

```
TotalMinutes      : 12600264
TotalSeconds      : 756015840
TotalMilliseconds : 756015840000
```

Résultat en secondes :

```
PS > (New-TimeSpan $(Get-Date -Year 1985 -Month 10 -Day 6 `
-Hour 8 -Minute 30) $(Get-Date)).TotalSeconds

756015900
```



La commande **New-TimeSpan** retourne une valeur de type **TimeSpan**. Pour observer toutes les méthodes applicables au type **TimeSpan**, tapez la commande suivante : **New-Timespan | Get-Member**

6. Applications en tout genre

a. Manipulations autour des dates

Dans cette partie, nous vous donnons quelques exemples de scripts pour vous montrer l'étendue des applications possibles autour des dates.

Exemple :

Affichage du mois en cours en mode graphique.

Bien que nous n'ayons pas encore abordé les classes graphiques du Framework .NET avec PowerShell (cf. chapitre .NET - Windows Forms), cette fonction vous donne un rapide aperçu des possibilités graphiques offertes.

Notez que pour instancier des objets de la classe **Windows.Forms.Form** il est nécessaire au préalable de charger l'assembly correspondante. Nous ne vous en dirons pas plus pour le moment car nous verrons tout cela dans le chapitre .NET.

Voici le script :

```
#Calendrier.ps1

# Chargement de l'assembly Graphique
[System.Reflection.Assembly]::LoadWithPartialName('System.windows.forms')

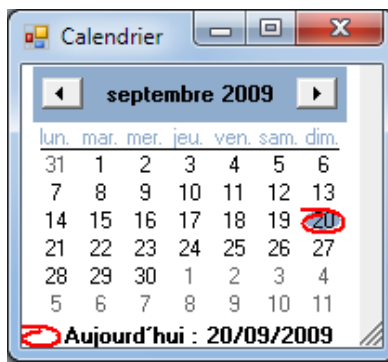
# Création de l'objet Form
$form = new-object Windows.Forms.Form
$form.Text = 'Calendrier'
$form.Size = new-object Drawing.Size(210,190)

# Création de l'objet calendrier
$calendrier = new-object System.Windows.Forms.MonthCalendar

# Ajout du calendrier à la forme
$form.Controls.Add($calendrier)

# Affichage de la forme
$form.Add_Shown({$form.Activate()})
[void]$form.showdialog()
```

Résultat :



Calendrier en mode graphique

b. Active Directory

Si vous fouillez un peu dans Active Directory et que vous cherchez la date du dernier « login » d'un utilisateur, ne faites pas un bond en arrière quand vous verrez un chiffre hallucinant sur 64 bits !!!

En réalité, ce chiffre correspond au nombre d'intervalles de dix millionièmes écoulés entre le 1^{er} janvier 1601 à 0h00 et la date en question.



Un peu d'histoire : le calendrier grégorien, qui a été mis en place en 1582 par le pape Grégoire XIII stipule qu'une année est composée de 365 jours, sauf quand elle est bissextile, c'est-à-dire, divisible par 4 et sauf les années séculaires (divisibles par 100), qui ne sont bissextiles que si elles sont divisibles par 400. Or, en ce qui nous concerne, le dernier multiple de 400 avant l'ère informatique est l'an 1600. C'est donc cette date qui va servir de point de départ pour simplifier l'algorithme de détermination de la date.

Évidemment, il est souhaitable de convertir ce nombre en date « humainement » compréhensible. C'est là qu'intervient la méthode **AddTicks** de l'objet **DateTime**. En effet, un tick correspondant à un intervalle de dix millionièmes de seconde, nous n'avons qu'à ajouter autant de ticks qu'indique la valeur du lastlogin à la date du 1^{er} janvier 1601 à 0h00, pour obtenir une date représentative.

Exemple :

La première étape consiste évidemment à récupérer les utilisateurs présents dans Active Directory :

```
PS > $ldapQuery = '(&(objectCategory=user))'
PS > $de = New-Object System.DirectoryServices.DirectoryEntry
PS > $ads = New-Object System.DirectoryServices.DirectorySearcher `
    -argumentlist $de,$ldapQuery
PS > $complist = $ads.FindAll()
```

Nous voici donc avec la variable **\$complist** qui contient tous les utilisateurs. Reste maintenant à afficher le nom de l'utilisateur ainsi que sa date de dernière connexion.

```
PS > ForEach ($i in $complist) {
    $LastLogon = $i.Properties['lastlogon']
    $LastLogon = [int64]::parse($LastLogon)

    # convertit la représentation de LastLogon sous forme d'un entier de 64 bits

    #Création d'une date : 1/1/1601
    $date = (Get-Date -Year 1601 -Month 1 -Day 1 -Hour 0 `
        -Minute 0 -Second 0)

    #Ajout des ticks à la date d'origine
    $date_derniere_connexion = $date.AddTicks($LastLogon)
    Write-Host si.properties['name'] ": $date-derniere-connexion"
}
```

c. Les fichiers

Voici quelque chose de plus spectaculaire et jusqu'à présent impossible à réaliser avec l'explorateur.

Grâce à PowerShell, vous pouvez désormais changer la date de dernier accès aux fichiers, la date de dernière modification et même la date de création, ce qui peut provoquer des situations assez inattendues. En voici le cheminement.

Étape 1 - Création d'un fichier

```
PS > New-Item -Name essai.txt -Type File

Répertoire : C:\Temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a--	20/09/2009 12:59	0	essai.txt

Étape 2 - Vérification de la date de création de ce fichier

```
PS > (Get-Item essai.txt).Get_CreationTime()
dimanche 20 septembre 2009 13:00:58
```

Étape 3 - Attribution des nouvelles dates de création et de dernier accès

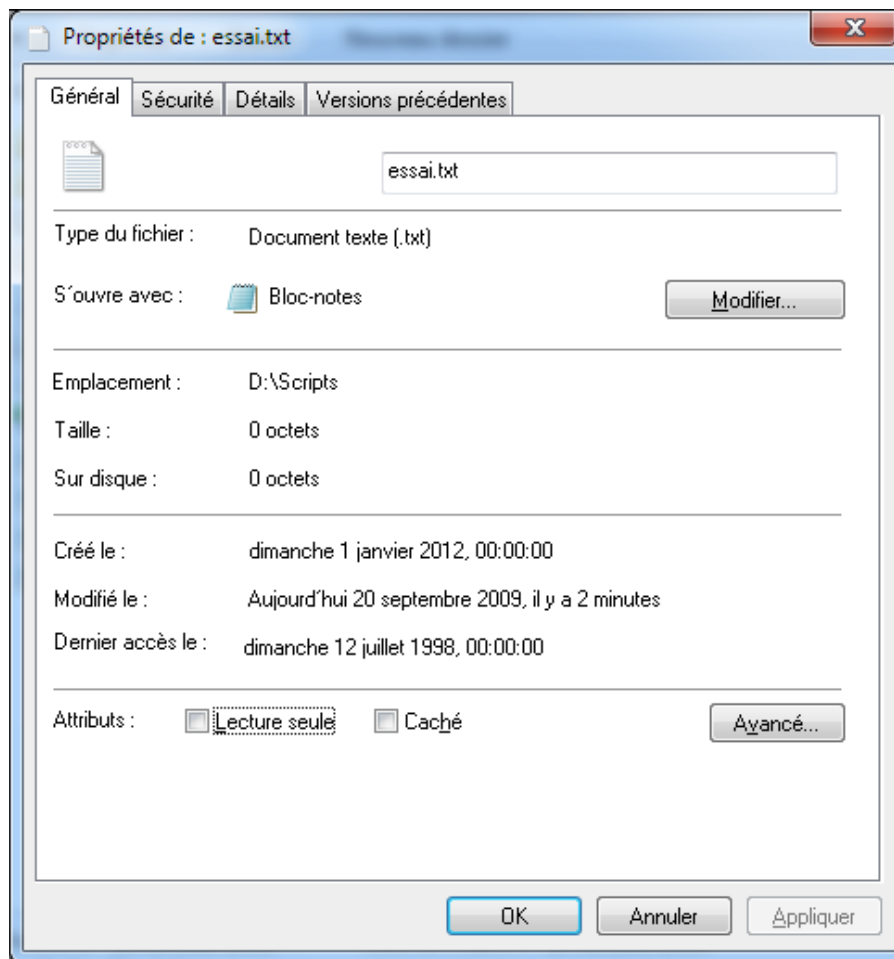
Pour cela créons deux variables : **\$date_dernier_acces** qui équivaut à la date du 13 juillet 1998 et **\$date_creation** qui est portée au 10 juillet 2020.

```
PS > $Date_Dernier_Acces = (Get-Date -Year 1998 -Month 7 `
                           -Day 12 -Hour 0 -Minute 0 -Second 0)
PS > $Date_Creation = (Get-Date -Year 2012 -Month 1 `
                        -Day 1 -Hour 0 -Minute 0 -Second 0)
PS > $Fichier = Get-Item essai.txt
PS > $Fichier.Set_CreationTime($Date_Creation)
PS > $Fichier.Set_LastAccessTime($Date_Dernier_Acces)
```



La création d'un objet correspondant au fichier est une étape intermédiaire qui peut être remplacée par la notation suivante : **(Get-Item essai.txt).Set_CreationTime(\$Date_Creation)**

Nous vous laissons maintenant le soin de découvrir les propriétés de votre fichier en faisant soit : clic droit - **propriétés**.



Propriétés du fichier

Soit en tapant la commande suivante :

```
PS > Get-Item essai.txt | Format-Table CreationTime,LastWriteTime,
LastAccessTime
```

CreationTime	LastWriteTime	LastAccessTime
-----	-----	-----
01/01/2012 00:00:00	20/09/2009 13:03:29	12/07/1998 00:00:00

Internationalisation des scripts

Dans la version 1 de PowerShell, l'édition de scripts destinés à un public constitué de personnes de nationalités différentes n'est pas chose évidente, l'éditeur se doit de traduire manuellement tout le contenu textuel de ses scripts. Dans PowerShell v2, apparaît ce qu'on appelle « l'internationalisation de script ». Le principe est de permettre l'écriture de scripts en différents langages sans pour autant modifier le code contenu dans le script. Par exemple, si vous créez un script et que celui-ci doit être exécuté par plusieurs personnes, chacune utilisant une version de PowerShell différente en termes de langage (variable **\$PSUICulture** différente). Et bien le fait d'utiliser l'internationalisation, permettra aux utilisateurs d'obtenir toute la partie textuelle du script dans leur langue, et ce, sans aucune manipulation de leur part. Regardons à présent comment cela fonctionne.

La première précaution à prendre est, bien évidemment, de séparer les données (data) du code contenu dans le script. Pour ce faire, nous utilisons une nouveauté de PowerShell qu'est la section « Data ». Et c'est à l'intérieur de cette section que nous utilisons une nouvelle commande de PowerShell v2 du nom de **ConvertFrom-StringData**, qui va créer une table de hachage des chaînes de caractère.

Exemple :

```
$TexteScript = Data {
    ConvertFrom-StringData @'
    Message_1 = Bonjour
    Message_2 = Entrez une valeur
    Message_3 = Entrez une autre valeur
    Message_4 = Le résultat de l'addition de ces deux valeurs est :
'@
}
```

Notez que nous utilisons ici une **Here-String**. Une Here-String (cf chapitre À la découverte de PowerShell) commence avec un séparateur @' et se termine par '@ (le dernier séparateur doit absolument être précédé d'un retour chariot). Tous les caractères entre les délimiteurs @' et '@ sont considérés comme du texte pur.

De façon à permettre la traduction de ces « data », une traduction en différentes langues doit être sauvegardée dans des fichiers .psd1 et sous une arborescence particulière.

Les fichiers .psd1 se doivent d'être enregistrés dans un sous-répertoire au format <langage>-<Pays >, comme l'indique la variable **\$PSUICulture**. Ainsi, si le script principal nommé MonScript.ps1 se trouve dans le répertoire C:\Temp, les fichiers MonScript.psd1 se trouveront sous l'arborescence suivante :

```
C:\Temp\MonScript.ps1      # Script Principal
C:\Temp\en-US\MonScript.psd1 # Fichier des données traduites en anglais
C:\Temp\es-ES\MonScript.psd1 # Fichier des données traduites en espagnol
...
```

Exemple de contenu des fichiers .psd1 :

Fichier C:\Temp\en-US\MonScript.psd1.

```
ConvertFrom-StringData @'
    Message_1 = Hello
    Message_2 = Enter a value
    Message_3 = Enter a second value
    Message_4 = The result of the addition of these two values is:
'@
```

Fichier C:\Temp\es-ES\MonScript.psd1

```
ConvertFrom-StringData @'
    Message_1 = Hola
    Message_2 = Introducid un valor
    Message_3 = Introducid un segundo valor
    Message_4 = El resultado de la adición de estos dos valores
es la siguiente:
'@
```

Enfin, dernière étape, permettre l'importation des chaînes de caractères dans la langue de l'interface utilisateur via la

commandelette **Import-LocalizedData**. L'utilisation de cette commandelette importe le fichier .psd1 correspondant à la langue utilisée, et rend transparentes les actions de traduction des éléments textuels du script. Le script complet devient le suivant :

```
$TexteScript = Data {  
    #Culture fr-FR  
    ConvertFrom-StringData @'  
        Message_1 = Bonjour  
        Message_2 = Entrez une valeur  
        Message_3 = Entrez une autre valeur  
        Message_4 = Le résultat de l'addition de ces deux valeurs est :  
'@  
}  
  
Import-LocalizedData TexteScript  
  
# Début du script  
  
Write-host $TexteScript.Message_1  
Write-host $TexteScript.Message_2  
[int]$var1 = Read-host  
Write-host $TexteScript.Message_3  
[int]$var2 = Read-host  
$resultat = $var1 + $var2  
Write-host "$($TexteScript.Message_4) $resultat"  
  
# Fin du script
```

En exécutant le script précédant sur un poste Windows version US, nous obtenons le résultat suivant :

```
PS > C:\temp\MonScript.ps1  
Hello  
Enter a value  
5  
Enter a second value  
6  
The result of the addition of these two values is: 11
```

Objets PSBase et PSObject

Parlons à présent d'une information très peu documentée, mais très utile, que sont les objets PSBase et PSObject. Comme nous vous l'avions déjà dit, PowerShell est basé sur le Framework .NET. Ainsi, les objets que nous manipulons, sont en grande majorité des objets .NET.

Mais PowerShell est également amené à fonctionner avec d'autres objets tels que les objets COM et WMI, qui ne partagent pas la même technologie. En fait, lorsque nous utilisons ces différents objets, PowerShell nous donne une représentation commune, avec des propriétés et des méthodes. C'est en quelque sorte une couche d'abstraction, permettant d'harmoniser l'interface, et ce, quel que soit la technologie de l'objet. Pour cela, PowerShell utilise ce qu'on appelle une adaptation de type (« Type Adaptation ») réalisée par PSObject. Cet objet va faire du « wrapping » (qui vient de wrap qui signifie envelopper) de l'objet de base. Cette adaptation de type met à la fois l'objet en forme, et l'habille en lui ajoutant quelques méthodes natives à PSObject et que par conséquent nous retrouvons partout, comme *ToString*, *CompareTo*, *Equals*, etc.

Prenons par exemple le cas d'un objet de type *DateTime* :

```
PS > $Date = Get-Date
```

Regardons à présent toutes les informations à propos de cet objet tel que PowerShell nous le présente avec une adaptation de type PSObject. Pour cela, tapons simplement la ligne suivante :

```
PS > $date.PsObject

Members      : {DisplayHint, DateTime, Date, Day...}
Properties    : {DisplayHint, DateTime, Date, Day...}
Methods      : {Add, AddDays, AddHours, AddMilliseconds...}
ImmediateBaseObject : 10/12/2008 08:00:00
BaseObject    : 10/12/2008 08:00:00
TypeNames     : {System.DateTime, System.ValueType, System.Object}
```

On s'aperçoit qu'il existe de nombreuses propriétés décrivant chacune des informations sur l'objet. Le détail de ces propriétés est donné dans le tableau suivant :

Propriété	Description
Member	Liste tous les membres de l'objet. Cela comprend les membres de l'objet de base, les membres étendus, et les membres natifs d'un objet PSObject.
Properties	Liste toutes les propriétés de l'objet. Cela comprend les propriétés de l'objet de base ainsi que les propriétés étendues.
Methods	Liste toutes les méthodes de l'objet. Cela comprend les méthodes de l'objet de base ainsi que les méthodes étendues.
ImmediateBaseObject	Retourne l'objet de base encapsulé par PSObject.
BaseObject	Retourne l'objet de base.
TypeName	Liste le nom des types de l'objet.

Seulement, en utilisant cette vue que nous donne PowerShell, il arrive que l'on se prive de quelques-unes des fonctionnalités de l'objet de technologie sous-jacente. Et cela peut parfois poser certains problèmes. Prenons l'exemple présenté dans le chapitre Manipulation d'objets annuaire avec ADSI, qui consiste à lister les groupes d'une base de comptes locale.

Commençons donc par créer une connexion à la base SAM (*Security Account Manager*) grâce à la commande suivante :

```
PS > $connexion = [ADSI]'WinNT://.'
```

Puis regardons quelles méthodes allons nous pouvoir appliquer sur l'objet retourné par la propriété *Children* :

```
PS > $child = $connexion.Children

PS > $child | Get-Member
```

TypeName: System.Management.Automation.PSMethod

Name	MemberType	Definition
----	-----	-----
Copy	Method	System.Management.Automation
Equals	Method	System.Boolean Equals(Object obj)
GetHashCode	Method	System.Int32 GetHashCode()
GetType	Method	System.Type GetType()
get_IsInstance	Method	System.Boolean get_IsInstance()
get_MemberType	Method	System.Management.Automation
get_Name	Method	System.String get_Name()
get_OverloadDefinitions	Method	System.Collections.ObjectModel....
get_TypeNameOfValue	Method	System.String get_TypeNameOfValue()
get_Value	Method	System.Object get_Value()

Et là surprise, les membres listés ne sont pas ceux attendus. La question est donc comment pouvons-nous accéder à ces fonctionnalités de l'objet qui ont l'air masquées ? Et bien, c'est là qu'intervient PSBase. Ce dernier vous procure une vue sur l'objet de base (d'origine), et non sur l'interface PowerShell de l'objet.

En recommençant la même opération, mais cette fois en utilisant la propriété *Children* de l'objet de base nous obtenons ceci :

```
PS > $child = $connexion.PSBase.Children
PS > $child | Get-Member
```

TypeName: System.DirectoryServices.DirectoryEntry

Name	MemberType	Definition
----	-----	-----
AutoUnlockInterval	Property	System.Directory
BadPasswordAttempts	Property	System.Directory
Description	Property	System.Directory
FullName	Property	System.Directory
HomeDirDrive	Property	System.Directory
HomeDirectory	Property	System.Directory
LastLogin	Property	System.Directory
LockoutObservationInterval	Property	System.Directory
LoginHours	Property	System.Directory
LoginScript	Property	System.Directory
MaxBadPasswordsAllowed	Property	System.Directory
MaxPasswordAge	Property	System.Directory
MaxStorage	Property	System.Directory
MinPasswordAge	Property	System.Directory
MinPasswordLength	Property	System.Directory
Name	Property	System.Directory
objectSid	Property	System.Directory
Parameters	Property	System.Directory
PasswordAge	Property	System.Directory
PasswordExpired	Property	System.Directory
PasswordHistoryLength	Property	System.Directory
PrimaryGroupID	Property	System.Directory
Profile	Property	System.Directory
UserFlags	Property	System.Directory

Les membres sont totalement différents de ceux présentés nativement par PowerShell. Ainsi en utilisant ces propriétés, nous avons réellement accès à celles de l'objet de base, et nous pouvons continuer notre script.

```
# Get-LocalGroups.ps1

param ([String]$machine='.')
$connexion = [ADSI]'WinNT://$machine'
$connexion.PSBase.children |
Where {$_.PSBase.SchemaClassName -eq 'group'} | Foreach{$_.Name}
```

Les job en arrière-plan : Start-Job, Receive-Job, Remove-Job

Uniquement disponible avec PowerShell v2, il est désormais possible d'exécuter des commandes et des scripts en arrière plan (ou de façon asynchrone) sans interaction avec la console. Cette fonctionnalité est particulièrement intéressante lorsque l'on a un script assez long à s'exécuter, car l'exécution en arrière plan rend la main immédiatement à la console sans la bloquer. Les tâches exécutées en arrière plan sont ce que l'on appelle des « Jobs » avec PowerShell.

Pour connaître l'ensemble des commandes liées à l'utilisation des Jobs, tapez la commande suivante :

```
PS > Get-Command *Job* -Type cmdlet
```

CommandType	Name	Definition
Cmdlet	Get-Job	Get-Job [-Id] <Int32[]> [-Verbose] [-Debug] [-Erro...
Cmdlet	Receive-Job	Receive-Job [-Job] <Job[]> [[-Location] <String[]>] ...
Cmdlet	Remove-Job	Remove-Job [-Id] <Int32[]> [-Force] [-Verbose] [-Deb...
Cmdlet	Start-Job	Start-Job [-ScriptBlock] <ScriptBlock> [[-Initializa...
Cmdlet	Stop-Job	Stop-Job [-Id] <Int32[]> [-PassThru] [-Verbose] [-De...
Cmdlet	Wait-Job	Wait-Job [-Id] <Int32[]> [-Any] [-Timeout <Int32>] [...]

Commandelette	Description
Get-Job	Commande permettant de lister toutes les tâches s'exécutant en arrière plan.
Receive-Job	Commande permettant d'obtenir le ou les résultats des tâches qui se sont exécutées en arrière plan.
Remove-Job	Commande permettant de supprimer les tâches s'exécutant en arrière plan.
Start-Job	Commande permettant de démarrer une tâche en arrière plan.
Wait-Job	Commande permettant de d'attendre qu'une ou plusieurs tâches se termine pour rendre la main.

Lorsqu'un Job termine son exécution, il ne retourne rien à la console qui l'a lancé, mais à la place le résultat d'exécution est stocké dans un objet de type « job ». Il suffit alors de manipuler l'objet pour en récupérer le contenu ; contenu qui peut n'être que partiel si le job n'a pas terminé complètement son exécution.

```
PS > Start-Job -scriptblock {get-service}
```

Id	Name	State	HasMoreData	Location	Command
--	----	-----	-----	-----	-----
1	Job1	Running	True	localhost	get-service

Comme vous le remarquez, l'utilisation de la commande Start-Job dans sa version basique est relativement simple. Il suffit de la faire suivre de l'argument scriptblock et d'insérer un bloc d'exécution. Dès la commande saisie, nous récupérons l'accès à la console sans attendre la fin de l'exécution de notre commande. Nous voyons que l'état de celle-ci est actuellement en cours d'exécution (state : Running). Nous remarquons également que les jobs sont identifiés selon un numéro d'identification (Id) mais également par un nom.

Exemple :

Prenons par exemple une petite boucle variant de 1 à 10 où nous affichons bonjour 1, bonjour 2, ..., bonjour 10. Nous allons exécuter cette boucle en arrière plan avec la commande suivante :

```
PS > Start-Job -Name Job_Tableau -ScriptBlock {1..10 | Foreach { Write-Host "bonjour $_" }}
```

Id	Name	State	HasMoreData	Location	Command
--	----	-----	-----	-----	-----
5	Job_Tableau	Running	True	localhost	1..10 forea...

Maintenant nous pouvons observer grâce à la commande **Get-Job** la liste des jobs en arrière-plan en cours d'exécution ou terminés :

```
PS > Get-Job
```

Id	Name	State	HasMoreData	Location	Command
--	----	-----	-----	-----	-----
1	Job1	Completed	True	localhost	get-service
5	Job_Tableau	Completed	True	localhost	1..10

```
forea...
```

Nous voyons que l'état a changé et que notre job est à présent terminé (*Completed*). Pour obtenir le résultat de celui-ci ou plutôt l'affichage du résultat nous pouvons utiliser la commande : **Receive-Job**

```
PS > Get-Job -Id 1 | Receive-Job
```

Status	Name	DisplayName
-----	----	-----
Stopped	AeLookupSvc	Expérience d'application...
Stopped	ALG	Service de la passerelle d...
Stopped	AppIDSvc	Identité de l'application...
...		

Ou encore, en filtrant sur le nom, sur la tâche Job_Tableau par exemple :

```
PS > Get-Job -Name Job_Tableau | Receive-Job
```

```
bonjour 1
bonjour 2
bonjour 3
bonjour 4
bonjour 5
bonjour 6
bonjour 7
bonjour 8
bonjour 9
bonjour 10
```

À présent, afin de libérer de la mémoire nous devons supprimer le job avec la commande **Delete-PsJob**. Si nous ne le faisons pas, le job reste dans le cache de la console courante ; néanmoins il sera tout de même détruit à la fermeture de la console

```
PS > Remove-Job -Id 5
```

Ou encore, d'une autre manière :

```
PS > Remove-Job -Name Job_Tableau
```

Enfin, si vous souhaitez supprimer tous les jobs :

```
PS > Remove-Job *
```



Il est également possible d'exécuter des commandes ou des scripts PowerShell sur des machines distantes. Il n'est pas ici question d'ouvrir une console interactive à distance ; mais il s'agit bien uniquement de pouvoir exécuter des commandes ou des scripts à distance de façon non interactive. Ce point sera abordé dans le chapitre Exécution à distance.

Snap-Ins et modules

Avec PowerShell 1.0, l'ajout de fonctionnalités et de commandelettes se réalise par le biais des Snap-Ins. Pour des raisons de compatibilité, les Snap-Ins sont toujours supportés dans PowerShell v2. Cependant les modules sont amenés à remplacer progressivement les Snap-Ins. À présent, la création de « compléments » s'en trouve être beaucoup plus souple et facile, et n'est plus réservée aux développeurs comme pouvaient l'être les Snap-Ins

1. Les Snap-Ins : Add-PSSnapin, Remove-PSSnapin

Les Snap-Ins sont des fichiers compilés (DLL) qui permettent de partager un ensemble de commandelettes considérées comme des extensions de fonctionnalité de PowerShell. En réalité, les Snap-Ins fournissent le même service que les modules, à la différence près que les modules ne sont pas obligatoirement des fichiers compilés.

a. Lister les Snap-Ins installés

Pour connaître la liste des Snap-Ins présents sur votre machine et importés dans la session courante, tapez la commande **Get-PSSnapin**.

```
PS > Get-PSSnapin

Name          : Microsoft.PowerShell.Diagnostics
PSVersion     : 2.0
Description   : Le composant logiciel enfichable Windows PowerShell
contient les applets de commande Windows Eventing et Performance Counter.
Name          : Microsoft.WSMan.Management
PSVersion     : 2.0
Description   : Ce composant logiciel enfichable Windows PowerShell
contient des applets de commande (tels que Get-WSManInstance et
Set-WSManInstance) qui sont utilisées par l'hôte Windows PowerShell
pour gérer les opérations WSMan.
Name          : Microsoft.PowerShell.Core
PSVersion     : 2.0
Description   : Ce composant logiciel enfichable Windows PowerShell
contient des applets de commande utilisées pour gérer les composants
de Windows PowerShell.
Name          : Microsoft.PowerShell.Utility
PSVersion     : 2.0
Description   : Ce composant logiciel enfichable Windows PowerShell
contient des applets de commande utilitaires qui permettent de manipuler
des données.
Name          : Microsoft.PowerShell.Host
PSVersion     : 2.0
Description   : Ce composant logiciel enfichable Windows PowerShell
contient des applets de commande (telles que Start-Transcript et
Stop-Transcript) fournies pour être utilisées avec l'hôte de
la console Windows PowerShell.
Name          : Microsoft.PowerShell.Management
PSVersion     : 2.0
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     : 2.0
Description   : Ce composant logiciel enfichable Windows PowerShell
contient des applets de commande qui permettent de gérer la sécurité
de Windows PowerShell.
```

Get-PSSnapin possède également le switch **-Registered**. Celui-ci lorsque spécifié permet de lister les Snap-Ins disponibles du système qui n'ont pas été importés dans la session courante. Le résultat de la commande ne contient pas les Snap-Ins nécessaires au fonctionnement de PowerShell.

Exemple :

```
PS > Get-PSSnapin -Registered
```

```
Name       : VMware.VimAutomation.Core
PSVersion  : 2.0
Description : This Windows PowerShell snap-in contains Windows PowerShell
cmdlets used to manage vSphere.
```

Ce qui signifie que ce Snap-In est installé mais non importé dans la session courante.

b. Importer un Snap-In

L'import se réalise quand à lui avec la commande **Add-PSSnapin**. Prenons l'exemple du Snap-In fourni par l'éditeur de logiciels de virtualisation VMware. VMware fournit un ensemble de commandelettes PowerShell capable d'administrer des serveurs VMware et leurs machines virtuelles associées. Disponible sur le site de VMware sous le nom vSphere PowerCLI, cette boîte à outils est un ensemble de fichiers DLL, qui une fois installés sont importables en tant que Snap-In via la commande **Add-PSSnapin** :

```
PS > Add-PSSnapin -Name VMware.VimAutomation.Core
```

Une fois le Snap-In chargé, si l'on sait que le nouveau jeu de commandes contient les lettres « VM », nous pouvons lister les commandes ainsi :

```
PS > Get-Command -Type cmdlet -Name *VM*
CommandType      Name                Definition
-----
Cmdlet           Add-VMHost          Add-VMHost [-Name] <String> [[...
Cmdlet           Add-VMHostNtpServer Add-VMHostNtpServer [-NtpServe...
Cmdlet           Get-VM              Get-VM [[-Name] <String[]>] [-...
Cmdlet           Get-VMGuest         Get-VMGuest [-VM] <VirtualMach...
Cmdlet           Get-VMHost          Get-VMHost [[-Name] <String[]>...
Cmdlet           Get-VMHostAccount   Get-VMHostAccount [[-Id] <Stri...
```

Mais nous pouvons encore faire mieux car si une commande ne contient pas les caractères « VM » nous ne la listerons pas en faisant ainsi.

c. Lister les commandes d'un Snap-In

Une commandelette PowerShell est caractérisée par un certain nombre de propriétés. Parmi celles-ci, il en existe une en particulier qui indique le Snap-In d'appartenance de chaque commandelette. Il s'agit de la propriété PSSnapin.

Ainsi, grâce à cette propriété qui va nous servir de filtre, nous allons pouvoir lister le contenu des commandes fournies par un Snap-In donné.

Exemple :

Liste des commandes contenues dans les Snap-Ins qui contiennent le mot « *Diagnostics* »

```
PS > Get-Command | Where {$_.PSSnapin.name -Match 'Diagnostics'}
```

Ou si l'on connaît le nom exact du Snap-In :

```
PS > Get-Command | Where {$_.PSSnapin.name -eq 'VMware.VimAutomation.Core'}
```

d. Décharger un Snap-In

Lorsque le Snap-In n'a plus raison d'être, vous pouvez avoir envie de le supprimer de la session courante (pas du système) par la commande **Remove-PSSnapin**.

Exemple :

```
PS > Remove-PSSnapin -Name 'VMware.VimAutomation.Core'
```

2. Les modules

Un module est une sorte de conteneur (package) qui regroupe des scripts, des commandes, des variables, des alias et des fonctions. L'avantage est que les modules sont facilement partageables afin d'en faire profiter d'autres

utilisateurs. L'utilisation de modules permet de créer des scripts qui s'appuient sur d'autres scripts présents dans vos modules ; ce qui évite d'avoir à inclure le contenu d'un script dans le script en cours de développement. Facilitant ainsi leur maintenance.

L'idée de la Team PowerShell est de bâtir une grande communauté d'utilisateurs de PowerShell, et de faire en sorte que celle-ci puisse aisément s'échanger ou partager des modules à l'image de la communauté CPAN (*Comprehensive Perl Archive Network*) que connaissent bien les utilisateurs du langage PERL.

Les modules se présentent sous la forme de dossiers contenant un ou plusieurs fichiers. Ces répertoires modules sont disposés à l'emplacement suivant : %UserProfile%\Documents\WindowsPowerShell\Modules. Sous Windows 7, ce répertoire n'existe pas par défaut, il est possible de le créer à l'aide de l'instruction suivante :

```
PS > New-Item -Type directory -Path $home\Documents\WindowsPowerShell\Modules
```

 L'emplacement \$env:UserProfile\Documents\WindowsPowerShell\Modules constitue l'emplacement pour les modules applicables aux utilisateurs. Pour une application système et donc accessible à l'ensemble des utilisateurs, l'emplacement est le suivant \$env:windir\System32\WindowsPowerShell\v1.0\Modules. À noter, l'existence de la variable d'environnement \$PSModulePath qui regroupe ces deux emplacements.

Avec Windows Server 2008 R2, Windows PowerShell est fourni avec plusieurs modules préinstallés. Il suffit d'utiliser l'assistant « Ajout de fonctionnalités » du gestionnaire de serveur pour installer automatiquement les modules de fonctionnalités que vous sélectionnez. Mais si vous recevez un module sous forme de dossier contenant des fichiers, il suffit simplement de le placer dans le répertoire Modules pour pouvoir l'importer dans Windows PowerShell.

Il existe plusieurs types de module, ces derniers sont décrits ci-dessous.

Type de module	Description
Script	Un module de type Script est un module composé d'un fichier (.psm1) qui contient du code PowerShell. Il s'agit du type le plus courant.
Binary	Un module de type Binary est un module qui contient du code compilé (fichier .dll).
Manifest	Un module de type Manifest est composé d'un fichier (.psd1) contenant plusieurs informations relatives à un module tel que son mode d'exécution, l'auteur, le numéro de version, etc.
Dynamic	Un module de type Dynamic est un module qui n'est pas stocké sur disque. Il s'agit d'un module de courte durée et par conséquent non visible en utilisant la commande Get-Module (voir ci-après).

a. Lister les modules

Afin de connaître les modules déjà importés, PowerShell v2 est doté de la commande Get-Module.

Get-Module possède les paramètres suivants :

Paramètre	Description
All <Switch>	Obtient tous les modules exportés pour tous les modules disponibles
ListAvailable <Switch>	Obtient tous les modules importables dans la session.
Name <String[]>	Obtient uniquement le ou les modules spécifié(s)

Utilisée seule, Get-Module retourne la liste des modules importés dans la session courante :

```
PS > Get-Module
```

Si vous souhaitez lister les modules installés (dans le répertoire Modules) mais non importés, il faut alors utiliser le paramètre -listAvailable.

Par exemple sous Windows 7 :

```
PS > Get-Module -listAvailable
ModuleType Name                               ExportedCommands
-----
Manifest AppLocker                           {}
Manifest BitsTransfer                        {}
Manifest PSDiagnostics                       {}
Manifest TroubleshootingPack                 {}
```

On s'aperçoit que quatre modules de type « manifest » sont installés par défaut avec Windows 7 (il s'agit des quatre modules placés dans l'arborescence \$env:windir\System32\WindowsPowerShell\v1.0\Modules).

Voici un tableau récapitulatif des modules installés par défaut dans les systèmes d'exploitation Windows 7 et Windows Server R2 :

Windows 7	Windows Server 2008 R2	Description des modules	Commandes disponibles par module
AppLocker	AppLocker	Empêcher l'exécution de logiciels non autorisés	Get-AppLockerPolicy Get-AppLockerFileInformation Test-AppLockerPolicy New-AppLockerPolicy Set-AppLockerPolicy
BitsTransfer	BitsTransfer	Transfert intelligent de fichiers en arrière plan	Start-BitsTransfer Remove-BitsTransfer Resume-BitsTransfer Get-BitsTransfer Add-BitsFile Set-BitsTransfer Complete-BitsTransfer Suspend-BitsTransfer
PSDiagnostics	PSDiagnostics	Aide au diagnostic de Windows	Enable-PSTrace Enable-WSManTrace Start-Trace Disable-PSWSManCombinedTrace Disable-PSTrace Disable-WSManTrace Get-LogProperties Stop-Trace Enable-PSWSManCombinedTrace Set-LogProperties
TroubleShootingPack	TroubleShootingPack	Aide à la résolution de problèmes	Get-TroubleshootingPack Invoke-TroubleshootingPack
	ADRMS	Microsoft Windows Active Directory Rights Management Services Module	Uninstall-ADRMS Update-ADRMS Install-ADRMS
	BestPractices	Best Practices Module	Get-BpaModel Set-BpaResult

			Invoke-BpaModel Get-BpaResult
	ServerManager	Server Manager Module	Remove-WindowsFeature Get-WindowsFeature Add-WindowsFeature

b. Importer un module

Lorsqu'un module est correctement installé dans le répertoire module, il faut ensuite l'importer. Cette opération s'effectue avec la commande `import-module <nom module>`.

Exemple :

```
PS > Import-Module BitsTransfer
```

 La commande `Import-Module` importe les modules dans votre session utilisateur de PowerShell. Pour que l'importation des modules soit effective pour l'ensemble des utilisateurs, il est recommandé d'ajouter la commande `Import-Module` au profil machine de PowerShell (cf chapitre Maîtrise du Shell - Personnaliser PowerShell en modifiant son profil).

Une fois le module installé, la commande `Get-Module` abordée précédemment vous confirmera que le module est correctement importé.

```
PS > Get-Module

ModuleType Name          ExportedCommands
-----
Manifest   BitsTransfer {Start-BitsTransfer, Remove-BitsTransfer, Resume-Bit...
```

c. Lister les commandes d'un module

Pour connaître les commandes apportées par le module importé, demandez la propriété `ExportedCommands` comme ceci :

```
PS > (Get-Module BitsTransfer).ExportedCommands

Name          Value
----
Start-BitsTransfer Start-BitsTransfer
Remove-BitsTransfer Remove-BitsTransfer
Resume-BitsTransfer Resume-BitsTransfer
Get-BitsTransfer  Get-BitsTransfer
Add-BitsFile      Add-BitsFile
Set-BitsTransfer  Set-BitsTransfer
Complete-BitsTransfer Complete-BitsTransfer
Suspend-BitsTransfer Suspend-BitsTransfer
```

 Pour lister les commandes apportées par un module, le plus simple est de le charger au préalable avec la commande `Import-Module`. Ceci étant, vous pouvez également explorer l'arborescence des fichiers et répertoires qui composent les modules si vous ne souhaitez pas les charger.

 Pour importer en une opération tous les modules disponibles de votre système vous pouvez utiliser la commande suivante : `Get-Module -ListAvailable | Import-Module`

d. Télécharger un module

La commandelette Remove-Module permet quant à elle de supprimer le module. Il n'est donc plus utilisable par l'utilisateur. Il n'est cependant pas supprimé du système.

```
PS > Remove-Module BitsTransfer
```



Pour supprimer en une opération tous les modules importés de votre session vous pouvez utiliser la commande suivante : `Get-Module | Remove-Module`

Introduction à la gestion des erreurs et au débogage

Dans votre vie de scripteur vous serez tôt ou tard confronté aux erreurs ou plus précisément à la gestion des erreurs à l'intérieur de vos scripts. Quoi de plus rageant qu'un script qui plante en cours d'exécution ? Lorsque cela arrive, il est préférable et plus élégant d'intercepter les erreurs afin d'afficher un joli message personnalisé plutôt que de laisser PowerShell afficher ses propres messages.

D'autre part, il peut être intéressant d'essayer d'anticiper les erreurs afin d'agir en conséquence. Par exemple, si vous essayez de supprimer une arborescence complète de fichiers et que pour une raison quelconque elle contient un fichier sur lequel vous n'avez pas les permissions adéquates, une erreur sera générée.

Grâce à ce que vous allez apprendre dans cette partie, vous allez pouvoir faire en sorte de décider comment l'interpréteur PowerShell devra se comporter face aux erreurs. Devra-t-il interrompre l'exécution du script ou bien continuer ? Et s'il continue doit-il ou non afficher un message d'erreur ? Si oui, quel type de message ? Celui par défaut ou un message que vous aurez défini vous-même ?

L'autre volet de la gestion des erreurs concerne le débogage. Lorsqu'un script compte plusieurs centaines de lignes de code, le débogage peut se révéler être une tâche complexe très consommatrice de temps. Heureusement, vous découvrirez que PowerShell possède quelques mécanismes bien utiles qui pourront vous sauver la mise et vous faire gagner un temps précieux.

La gestion des erreurs

Pour bien aborder ce sujet, il nous faut tout d'abord distinguer deux types d'erreurs : les **erreurs critiques** (« Terminating » est le terme anglais correspondant) et les **erreurs non-critiques** (« Non-Terminating » en anglais).

Les premières sont considérées comme graves, et lorsqu'elles surviennent l'exécution de la commande, ou du script dans certains cas, est interrompu. Les erreurs critiques se rencontrent généralement avec une erreur de syntaxe, une division par zéro, ou autres.

Les secondes, les erreurs non-critiques, sont plutôt considérées comme des avertissements ; la plupart des erreurs sont, par défaut, de ce type. Dans ce cas, l'exécution du script continue mais les erreurs sont consignées par PowerShell (dans **\$error**) et - sauf indication contraire - affichées à l'écran. On peut rencontrer ce type d'erreur, par exemple, lors de la suppression d'un fichier si les droits d'accès sont insuffisants ou si l'on cherche à déplacer un fichier qui n'existe pas.

Nous verrons que le comportement par défaut, qui consiste à continuer l'exécution d'un script lorsqu'une erreur non-critique est rencontrée, peut être modifié. Car dans certains cas, il peut être préférable d'arrêter le déroulement d'un script plutôt que de le laisser continuer au risque de provoquer d'autres erreurs en cascade qui pourraient mettre en péril au pire notre système, au mieux quelques fichiers.

Commençons par nous intéresser aux erreurs non-critiques, car généralement ce sont celles-ci que l'on rencontre le plus souvent.

Les erreurs non-critiques

Il faut savoir que PowerShell permet de définir son comportement face aux erreurs de plusieurs façons :

- Globalement : c'est-à-dire pour tout le script ou pour l'étendue en cours (cf. chapitre Fondamentaux) grâce à la variable de préférence **\$ErrorActionPreference**.
- Sélectivement : c'est-à-dire que pour chaque commandelette le comportement peut différer. Ceci est rendu possible grâce à l'utilisation d'un paramètre qui est commun à toutes les commandelettes. Ce paramètre se nomme **ErrorAction**.

1. Variable de préférence : \$ErrorActionPreference

Intéressons-nous pour l'instant à la « variable de préférence » (c'est ainsi qu'on appelle les variables intrinsèques qui stockent les préférences des utilisateurs) **\$ErrorActionPreference**.

Elle peut prendre les valeurs suivantes :

Valeur	Description
SilentlyContinue	Le script s'exécute sans afficher d'erreur même s'il en rencontre.
Continue	Le script continue s'il rencontre une erreur et l'affiche (valeur par défaut).
Stop	Le script s'interrompt s'il rencontre une erreur. Dans ce cas toutes les erreurs deviennent des erreurs critiques.
Inquire	Lorsqu'une erreur survient un prompt demande à l'utilisateur ce qu'il doit faire (continuer, continuer en mode silencieux, arrêter ou suspendre).

Comme par défaut **\$ErrorActionPreference** contient la valeur **Continue**, les erreurs non-critiques ne sont pas bloquantes ; elles sont seulement consignées par PowerShell et affichées à l'écran.



Lorsque **\$ErrorActionPreference** prend la valeur **Stop**, toutes les erreurs rencontrées deviennent des erreurs critiques. Ainsi une erreur non-critique apparaissant durant l'exécution d'un script interrompra ce dernier, exactement à la manière d'une erreur critique. Pour modifier la valeur de **\$ErrorActionPreference**, vous pouvez faire ceci **\$ErrorActionPreference = 'Stop'** (n'oubliez pas les guillemets !) ou **Set-Variable ErrorActionPreference Stop**

Exemple :

Lorsque vous êtes sous Windows Vista ou Windows 7 et que vous lancez normalement PowerShell, celui-ci s'exécute en mode utilisateur quand bien même vous vous seriez connecté avec votre compte administrateur. De ce fait, il est tout à fait possible que vous n'ayez pas les droits d'accès suffisants pour afficher le contenu d'un répertoire en particulier.

```
PS > $ErrorActionPreference
Continue

PS > Get-ChildItem 'C:\document privé'
Get-ChildItem : L'accès au chemin d'accès 'C:\document privé'
est refusé.
Au niveau de ligne : 1 Caractère : 14
+ Get-ChildItem << 'C:\document privé'
```

Maintenant modifions la variable **\$ErrorActionPreference** avec la valeur **SilentlyContinue** et recommençons :

```
PS > $ErrorActionPreference = 'SilentlyContinue'
PS > Get-ChildItem 'C:\document privé'
```

Cette fois, bien que l'erreur soit toujours présente, elle ne s'affiche pas. Les erreurs d'ailleurs ne s'afficheront plus, et ce quel que soit la commande que nous pourrions taper tant que nous ne serons pas revenus en mode **Continue**.

```
PS > $ErrorActionPreference = 'SilentlyContinue'
PS > Get-ChildItem 'C:\document privé'
PS > Get-CommandQuiNExistePas
```



Plutôt que de jongler sans cesse avec l'étendue courante (celle du script), et au risque de ne plus savoir dans quel mode on se trouve, nous pourrions utiliser un bloc de script pour faire nos tests. Car vous l'aurez compris, la portée de **\$ErrorActionPreference** comme pour toutes les autres variables, se limite au bloc. Pour reprendre notre dernier exemple, nous pourrions écrire ceci :

```
PS > &{
>> $ErrorActionPreference = 'SilentlyContinue'
>> Get-ChildItem 'C:\document privé'
>> Get-CommandQuiNExistePas
>> }
>>
```

Ainsi, quel que soit le mode d'exécution courant de notre shell, le bloc s'exécutera toujours de la même façon.

2. Le paramètre -ErrorAction et les paramètres communs

Une autre technique consiste à utiliser les paramètres « communs » des commandelettes. On trouve très souvent le terme anglais « common parameters » pour les désigner.

Ceux-ci constituent l'une des grandes forces de PowerShell : l'homogénéité. En effet, ces paramètres sont présents pour tout le jeu de commandes de PowerShell.

Ces paramètres sont les suivants :

Paramètre	Description
ErrorAction (SilentlyContinue Continue Inquire Stop)	Détermine le comportement de la commandelette en cas d'erreur.
ErrorVariable <nom de variable>	L'erreur résultant sera stockée dans la variable passée en paramètre, en plus de \$Error .
Debug {\$true \$false}	Indique à la commandelette de passer en mode débogage. À noter que ce mode n'existe pas forcément pour toutes les commandelettes.
Verbose {\$true \$false}	Indique à la commandelette de passer en mode verbeux. À noter que ce mode n'existe pas forcément pour toutes les commandelettes.
OutVariable <nom de variable>	La sortie résultante de la commande au cours du traitement sera stockée dans la variable passée en paramètre.
OutBuffer <Int32>	Détermine le nombre d'objets à mettre en mémoire tampon avant d'appeler la commandelette suivante du pipeline.

Pour donner un peu plus de souplesse à nos scripts, et pour éviter de jouer sans cesse avec la variable **\$ErrorActionPreference**, nous pouvons utiliser le paramètre **-ErrorAction**. Ce paramètre va nous permettre d'agir sur le comportement de chaque commandelette exécutée et non plus au niveau global comme précédemment.

Exemple :

```
PS > $ErrorActionPreference = 'Continue'
PS > Get-ChildItem 'C:\document privé'
Get-ChildItem : L'accès au chemin d'accès 'C:\document privé'
est refusé.
Au niveau de ligne : 1 Caractère : 14
+ Get-ChildItem <<<< 'C:\document privé'
```

```
PS > gci 'C:\document privé' -ErrorAction SilentlyContinue
```



Nous avons utilisé l'alias « gci » de Get-ChildItem afin de faire en sorte que la commande tienne sur une seule ligne, et ce pour une meilleure compréhension. Nous aurions pu également utiliser le paramètre court « -EA » au lieu de ErrorAction. Grâce à l'emploi de **-ErrorAction SilentlyContinue** nous avons empêché l'affichage d'un message d'erreur alors que **\$ErrorActionPreference** était « en mode **Continue** ».

Nous venons d'illustrer les valeurs **Continue** et **SilentlyContinue** de **\$ErrorActionPreference**, mais nous n'avons pas encore parlé de **Stop** et de **Inquire**.

Stop permet d'interrompre l'exécution d'un script lorsqu'une erreur est rencontrée même s'il s'agit d'une erreur non-critique. C'est un moyen de s'assurer qu'un script ne pourra se dérouler si une erreur quelconque survient.

Quant à **Inquire**, cela indique à PowerShell de demander à l'utilisateur ce qu'il faut faire, comme dans l'exemple suivant :

```
PS > gci 'C:\document privé' -ErrorAction Inquire

Confirmer
L'accès au chemin d'accès 'C:\document privé' est refusé.
[0] Oui [T] Oui pour tout [I] Interrompre la commande
[S] Suspendre [?]
Aide (la valeur par défaut est « 0 ») : i
Get-ChildItem : L'exécution de la commande s'est arrêtée parce
que l'utilisateur a sélectionné l'option Halt.
Au niveau de ligne : 1 Caractère : 4
+ gci << 'C:\document privé' -ErrorAction Inquire
```



On peut au choix pour passer une valeur à un paramètre faire comme ceci : `gci 'C:\document privé' -ErrorAction Inquire` ; `gci 'C:\document privé' -ErrorAction:Inquire`. Ces deux formes de syntaxe sont possibles.

3. Consignation des erreurs

Quel que soit le mode dans lequel vous vous trouvez (**Continue**, **SilentlyContinue**, **Stop** ou **Inquire**), il existe une variable « automatique » nommée **\$Error** qui contient, sous forme d'un tableau (ou plus précisément un *ArrayList*, il s'agit ici d'un tableau d'objets de type *ErrorRecord*), les 256 derniers messages d'erreurs rencontrés. 256 correspondant à la variable **\$MaximumErrorCount**. Si toutefois vous aviez besoin de stocker davantage d'erreurs, vous pouvez modifier ce nombre.

La dernière erreur se retrouve toujours dans **\$Error[0]**, l'avant-dernière dans **\$Error[1]** et ainsi de suite...

Par exemple :

```
PS > $ErrorActionPreference = 'Continue'
PS > gci 'C:\document privé' -ErrorAction SilentlyContinue
```

Grâce au paramètre **ErrorAction** nous avons pu empêcher l'affichage d'un message d'erreur alors que nous étions au niveau du script en mode **Continue**.

Regardons maintenant le contenu de la variable **\$Error[0]** :

```
PS > $ErrorActionPreference = 'Continue'
PS > gci 'C:\document privé' -ErrorAction SilentlyContinue
PS > $Error[0]
Get-ChildItem : L'accès au chemin d'accès 'C:\document privé'
est refusé.
Au niveau de ligne : 1 Caractère : 4
+ gci << 'C:\document privé' -ErrorAction SilentlyContinue
```

Il existe un autre moyen de récupérer un message d'erreur qu'en utilisant **\$Error[0]**. Bien que **\$Error[0]** soit très pratique, à l'usage vous vous rendrez compte que dans certains cas nous n'obtenons pas toujours l'erreur escomptée. Imaginons que nous ayons quelques lignes de codes qui se succèdent les unes aux autres et qu'ensuite arrive notre test d'erreur avec **\$Error[0]**.

L'enregistrement **\$Error[0]** contient suffisamment d'informations pour qu'on puisse très facilement identifier la ligne qui a provoqué une erreur. Le problème, dans ce contexte, est surtout qu'il est possible de rater une erreur et d'enregistrer l'erreur générée par une autre commande, plus loin dans le script. Il faut alors remonter « à la main » dans le tableau **\$Error** pour retrouver la ligne qui nous intéresse, ce qui peut être fastidieux et rendre compliqué l'automatisation d'un traitement de l'erreur.

D'où l'avantage de « fixer » le stockage de l'erreur au niveau de la commande elle-même.

Grâce au paramètre **-ErrorVariable** nous allons pouvoir stocker le message d'erreur dans une variable choisie par nos soins, et ce pour chaque commandelette ou juste pour celles qui nous intéressent ; exactement à la manière du paramètre **-ErrorAction**.

Dans l'exemple suivant, nous allons envoyer l'erreur dans la variable **\$MaVariable**. Notez qu'il ne faut pas mettre le signe dollar devant le nom de la variable avec **-ErrorVariable**.

```
PS > $ErrorActionPreference = 'SilentlyContinue'
PS > gci 'C:\document privé' -ErrorVariable MaVariable
PS >
PS > $MaVariable
Get-ChildItem : L'accès au chemin d'accès 'C:\document privé'
est refusé.
Au niveau de ligne : 1 Caractère : 4
+ gci << 'C:\document privé' -ErrorVariable MaVariable
```



Si vous êtes un économe du clavier, vous pouvez vous contenter d'utiliser les paramètres abrégés **-ea** pour **-ErrorAction** et **-ev** pour **-ErrorVariable**.

4. Le type **ErrorRecord**

Examinons de plus près notre variable **\$MaVariable** car celle-ci s'avère être particulièrement intéressante. Tapez :

```
PS > $MaVariable | Get-Member -Force
      TypeName: System.Management.Automation.ErrorRecord

Name      MemberType Definition
-----
pstypenames CodeProperty System.Collections.ObjectModel...
psadapted MemberSet    psadapted {Exception, TargetObj...
PSBase MemberSet    PSBase {Exception, TargetObject...
psextended MemberSet    psextended {PSMessageDetails}...
psobject MemberSet    psobject {Members, Properties, ...
Equals Method      bool Equals(System.Object obj) ...
GetHashCode Method      int GetHashCode()...
GetObjectData Method      System.Void GetObjectData(Syste...
GetType Method      type GetType()
get_CategoryInfo Method      System.Management.Automation.Er...
get_ErrorDetails Method      System.Management.Automation.Er...
get_Exception Method      System.Exception get_Exception(...
get_FullyQualifiedErrorId Method      string get_FullyQualifiedErrorI...
get_InvocationInfo Method      System.Management.Automation.In...
get_PipelineIterationInfo Method      System.Collections.ObjectModel. ...
get_TargetObject Method      System.Object get_TargetObject(...
set_ErrorDetails Method      System.Void set_ErrorDetails(Sy...
ToString Method      string ToString()
CategoryInfo Property      System.Management.Automation.Er...
ErrorDetails Property      System.Management.Automation.Er...
Exception Property      System.Exception Exception {get...
FullyQualifiedErrorId Property      System.String FullyQualifiedErr...
InvocationInfo Property      System.Management.Automation.In...
PipelineIterationInfo Property      System.Collections.ObjectModel. ...
TargetObject Property      System.Object TargetObject {get...
PSMessageDetails ScriptProperty System.Object PSMessageDetails
```

Nous nous apercevons que le type est **ErrorRecord**, le même que celui des enregistrements du tableau **\$Error** ; et ce type possède les propriétés suivantes (les propriétés signalées d'une étoile ne sont disponibles qu'avec PowerShell v2) :

Propriété	Description
Exception	Il s'agit du message d'erreur tel qu'il s'affiche à l'écran. C'est en réalité un peu plus complexe que cela car cette propriété retourne en fait un objet dont le type varie en fonction de l'erreur. Pour le vérifier, il suffit d'observer le type et les membres de \$Error[0] , puis de \$Error[1] , vous risqueriez d'être surpris.
ErrorDetails	Contient des informations complémentaires sur l'erreur rencontrée. Cette propriété peut être nulle. Si non nulle, il est préférable d'afficher ErrorDetails.message au lieu de Exception.message car le message est beaucoup plus précis.
FullyQualifiedErrorId	Cette propriété identifie l'erreur de la façon la plus précise qui soit. À utiliser pour faire un filtre sur une erreur précise.
CategoryInfo	Retourne la catégorie d'erreur.
TargetObject	Objet ayant provoqué l'erreur. Cette propriété peut être nulle.
PipelineIterationInfo (*)	Retourne le statut du pipeline lorsqu'une erreur est créée.
InvocationInfo	Retourne le contexte dans lequel l'erreur s'est produite (cf. figure suivante), comme la position et le numéro de ligne.

Observons à présent les propriétés de notre erreur :

```
PS > $MaVariable | Format-List -Force
```

```
Exception           : System.UnauthorizedAccessException: L'accès au
chemin d'accès 'C:\document privé' est refusé.
                    à System.IO.__Error.WinIOError(Int32 errorCode,
String maybeFullPath)
                    à System.IO.Directory.InternalGetFileDirectoryNames
(String path, String userPathOriginal, String searchPattern, Boolean
includeFiles, Boolean includeDirs, SearchOption searchOption)
                    à System.IO.DirectoryInfo.GetDirectories(String
searchPattern, SearchOption searchOption)
                    à System.IO.DirectoryInfo.GetDirectories()
                    à Microsoft.PowerShell.Commands.FileSystemProvider.
Dir(DirectoryInfo directory, Boolean recurse, Boolean nameOnly,
ReturnContainers returnContainers)
TargetObject        : C:\document privé
CategoryInfo        : PermissionDenied: (C:\document privé:String)
[Get-ChildItem], UnauthorizedAccessException
FullyQualifiedErrorId : DirUnauthorizedAccessError,Microsoft.PowerShell.
Commands.GetChildItemCommand
ErrorDetails        :
InvocationInfo       : System.Management.Automation.InvocationInfo
PipelineIterationInfo : {0, 1}
PSMessageDetails    :
```

➤ Afin d'afficher toutes les propriétés nous sommes contraints d'utiliser le paramètre **-Force** de la commande **Format-List**. Cela est dû à un affichage personnalisé défini par les créateurs de PowerShell. Ils ont fait en sorte de n'afficher que le strict nécessaire, afin de ne pas submerger l'utilisateur de tout un tas de messages superflus.

5. Redirection de l'affichage des messages d'erreur

Pour en finir avec l'affichage des messages d'erreur, nous venons de voir la première méthode. Il s'agit de changer le mode d'exécution en **SilentlyContinue**, soit de façon globale (avec **\$ErrorActionPreference**), soit de façon sélective

commandelette par commandelette (avec le paramètre **-ErrorAction**).

L'autre méthode consiste à rediriger le flux d'écriture des messages d'erreurs, non plus vers le flux d'erreur, mais vers le flux d'affichage standard. Il devient par conséquent possible d'envoyer les erreurs soit dans un fichier texte, soit dans une variable.

a. Redirection des erreurs dans un fichier texte

Cela se fait grâce à l'emploi de l'opérateur « **2>** ».

Exemple :

Redirection des erreurs dans un fichier de log.

```
PS > Get-ChildItem 'C:\document privé' 2> Erreur.log
```

En faisant cela nous avons créé un fichier. Cependant, faites attention car s'il existe un fichier portant le même nom, il sera écrasé.



Si nous voulons ajouter du contenu à un fichier, il faut cette fois utiliser l'opérateur « **2>>** ».

b. Redirection des erreurs dans une variable

Nous avons vu précédemment que cela était possible avec l'emploi du paramètre **-ErrorVariable**. Ceci étant, il existe une seconde possibilité grâce à l'opérateur « **2>&1** ».

Cet opérateur indique à l'interpréteur d'envoyer les erreurs vers le flux standard, donc à l'écran, et non plus vers le flux d'erreurs. Ainsi il nous faut utiliser une variable pour stocker notre erreur.



Nous employons volontairement les termes « stocker notre erreur » au lieu de « stocker notre message d'erreur » car la variable d'erreur est de type **ErrorRecord** et non pas de type **String**.

Exemple :

Redirection des erreurs dans une variable.

```
PS > $Erreur = Get-ChildItem 'C:\document privé' 2>&1
```



Vous remarquerez que lorsqu'un message d'erreur s'affiche il est de couleur rouge. Lorsque vous utilisez l'opérateur « **2>&1** » sans variable pour récupérer l'erreur, le message affiché à l'écran est de couleur standard. Ceci est donc la preuve que le message d'erreur a été redirigé vers le flux standard.



Si la couleur rouge des messages d'erreur ne vous plaît pas, vous pouvez la changer, par exemple en vert : **\$host.PrivateData.set_ErrorForegroundColor('green')**. Pour la liste des couleurs, veuillez vous référer au chapitre Maîtrise du Shell - Personnaliser PowerShell en modifiant son profil.

c. Redirection des erreurs vers \$null

La redirection des messages peut se faire également vers **\$null** en utilisant la syntaxe « **2>\$null** ». Bien que l'erreur soit toujours consignée dans **\$Error**, le flux d'erreur est redirigé et ne s'affiche pas à l'écran.

Exemple :

Redirection vers \$null.

```
PS > Get-ChildItem 'C:\document privé' 2>$null
```

6. Interception des erreurs non-critiques

Il y a plusieurs façons d'intercepter les erreurs dans un script. La plus simple consiste à tester le résultat booléen contenu dans la variable `$?`. Cette variable contient le résultat d'exécution de la dernière opération.

Pour bien comprendre son fonctionnement prenons l'exemple suivant :

```
PS > &{
>> $ErrorActionPreference = 'SilentlyContinue'
>> [int]$i='ABC' # Source de l'erreur
>>
>> if ($?)
>> {
>>     Write-Host 'Tout va bien'
>> }
>> Else
>> {
>>     Write-Host "Une erreur est survenue : $($Error[0].exception.message)"
>> }
>> Write-Host 'Suite du script.'
>> }
>>
Une erreur est survenue : Impossible de convertir la valeur « ABC » en type «
System.Int32 ». Erreur : « Le format de la chaîne d'entrée est incorrect. »
Suite du script.
```

Dans l'exemple ci-dessus, si `$?` contient la valeur false c'est qu'une erreur s'est produite. Et c'est bien normal car nous essayons de mettre une chaîne dans une variable de type entier (Int) ce qui ne va pas sans poser problème. Ensuite nous affichons un message d'erreur personnalisé suivi de l'erreur en question.

Pour intercepter les erreurs de cette façon il faut que nous soyons en mode **Continue** ou **SilentlyContinue** car si nous nous trouvons en mode **Stop** le script s'interrompt juste au niveau de l'erreur et ne passe donc pas dans le test. Nous pouvons donc en conclure que cette façon de faire ne nous permettra pas de gérer les erreurs critiques.



Il vaut mieux se mettre en mode **SilentlyContinue** sinon PowerShell affichera l'erreur standard en plus du message personnalisé, ce qui peut faire désordre...

Il existe un autre moyen de savoir si un programme s'est bien déroulé avec `$LastExitCode`. Cette variable contient le code d'erreur de la dernière commande exécutée mais elle ne s'applique qu'aux fichiers exécutables externes à PowerShell. En d'autres termes, elle n'est pas utilisable avec les commandelettes. Sous Windows, lorsqu'un processus Win32 se termine, il retourne toujours un code de sortie sous forme d'entier. Par convention, ce code vaut **zéro** si le processus s'est déroulé sans erreur, une **autre valeur** dans le cas contraire.

Exemple :

```
PS > ping maMachine.domaine
La requête Ping n'a pas pu trouver l'hôte maMachine.domaine.
Vérifiez le nom et essayez à nouveau.
PS > $?
False
PS > $LastExitCode
1
PS > ping 127.0.0.1 -n 1

Envoi d'une requête 'Ping' 127.0.0.1 avec 32 octets de données :

Réponse de 127.0.0.1 : octets=32 temps<1ms TTL=128

Statistiques Ping pour 127.0.0.1:
    Paquets : envoyés = 1, reçus = 1, perdus = 0 (perte 0%),
Durée approximative des boucles en millisecondes :
    Minimum = 0ms, Maximum = 0ms, Moyenne = 0ms
PS > $LastExitCode
0
```

```
PS > $?  
True
```

La commande « ping » étant un exécutable externe (ping.exe), la variable **\$LastExitCode** contient une valeur à son exécution. Remarquez que même dans cette situation, **\$?** peut nous être utile. En fait, le principe de **\$?** consiste à vérifier si le dernier code de sortie d'un exécutable (**\$LastExitCode**) était à 0 ou pas.

Les erreurs critiques

Partons à présent à la chasse aux erreurs critiques, que l'on appelle couramment « exceptions ». C'est ainsi que nous allons tenter de les appeler afin de les distinguer des erreurs non-critiques.

Grâce à ce que nous allons découvrir dans cette partie, nous allons avoir encore plus de maîtrise sur nos scripts. Ainsi plutôt que de voir un script s'arrêter brutalement à cause d'une erreur critique nous allons pouvoir agir en conséquence et prendre les mesures (correctives ou alternatives) qui s'imposent. En effet, il est parfois utile pour un script de savoir si tout s'exécute normalement. Mais pour cela il va nous falloir essayer de prévoir les exceptions avant qu'elles ne se produisent...

1. Interception des erreurs critiques

La chasse aux exceptions est un jeu particulièrement intéressant qui se pratique en créant des « gestionnaires d'interception » (le terme américain correspondant est « trap handler ») en utilisant le mot clé « trap ».

La syntaxe est la suivante :

```
trap [Type d'erreur à intercepter]
{ ... bloc de code à exécuter en cas d'erreur ...;
[break | continue] }
```

Vous n'êtes pas obligés de spécifier le type d'erreur mais si vous l'omettez, toutes les erreurs sans distinction seront capturées. L'indication du type d'erreur permet d'avoir un contrôle plus précis sur le déroulement du script.



Pour une fois les créateurs de PowerShell n'ont pas opté pour le modèle verbe-nom dans la définition de cette commande. En effet, « trap » n'est pas une commandelette mais une instruction.

Avant de prendre un exemple concret, vous devez vous rappeler que PowerShell travaille avec des étendues (« scopes »). Celles-ci sont très importantes dans le cadre de l'interception des erreurs. Pour mémoire, l'interpréteur PowerShell représente l'étendue globale. Lorsque vous lancez un script, celui-ci crée une nouvelle étendue dans laquelle il va s'exécuter. De même, chaque fonction du script s'exécute dans une étendue qui lui est propre. Il en est de même pour chaque bloc d'instructions, généralement encadré par des accolades : { <bloc d'instructions> }. Il faut bien comprendre que PowerShell gère plusieurs niveaux d'étendues.

Lorsque l'on crée un gestionnaire d'interception plusieurs possibilités s'offrent à nous pour son positionnement :

- On le place dans une étendue spécifique, auquel cas son action sera limitée à cette étendue.
- On le place dans l'étendue globale, ainsi il pourra intercepter toutes les exceptions, y compris celles générées dans les sous-étendues (ou étendues enfant).

Il faut savoir que lorsqu'une exception est générée, PowerShell cherche à la passer au gestionnaire de l'étendue courante ; et plus précisément au corps du gestionnaire (la partie entre accolades). S'il ne trouve pas de gestionnaire dans son étendue courante, PowerShell quittera alors son étendue, et tentera de passer l'exception au gestionnaire de l'étendue parente, et ainsi de suite, jusqu'à remonter au gestionnaire de l'étendue globale.

Lorsqu'une exception est interceptée par un gestionnaire, les instructions contenues dans son corps sont exécutées. Lorsque nous en sommes là, nous pouvons dire au script, soit de continuer normalement son déroulement avec l'instruction Continue, soit de s'arrêter avec l'instruction Break. Si vous ne spécifiez rien, le comportement du gestionnaire dépendra de l'état de la variable **\$ErrorActionPreference**.

Valeur de \$ErrorActionPreference	Comportement de l'instructionTrap
Stop	Break : messages d'erreurs affichés.
Continue	Continue : messages d'erreurs affichés.
SilentlyContinue	Continue : messages d'erreurs non affichés.
Inquire	Demande confirmation à chaque erreur critique interceptée.



Afin de lever toute ambiguïté possible sur les différents modes d'exécution des gestionnaires d'interception mais aussi pour plus de souplesse, nous vous recommandons de systématiquement spécifier **Continue** ou **Break**. L'interception des erreurs n'étant pas la chose la plus évidente qui soit, il vaut mieux être le plus explicite possible pour ne pas avoir de surprises. D'autre part, lorsque vous vous pencherez à nouveau sur des scripts que vous aurez écrits plusieurs mois auparavant, vous comprendrez tout le sens de ces propos... En forçant explicitement « le mode » **Continue** dans une instruction **Trap**, les messages d'erreurs ne sont plus affichés. Ceci est une petite subtilité qu'il peut être bon de connaître.

Veillez noter qu'en cas d'arrêt de l'exécution d'un bloc de script avec **Break**, l'exception, en plus d'être passée au gestionnaire de l'étendue courante, est aussi transmise aux gestionnaires parents (de niveau supérieur). Tandis qu'avec **Continue**, l'exception n'est pas transmise au gestionnaire de niveau supérieur et le script continue son cours.

Voici une petite série d'exemples pour illustrer tous ces propos.

Exemple 1 :

Pour bien comprendre « Continue »...

Dans cet exemple, nous allons faire une boucle sur un indice qui varie de -2 à 2, puis nous diviserons le nombre 100 par cet indice. Le but étant de provoquer une erreur (critique) de division par zéro. Nous pouvons remarquer que nous avons deux étendues distinctes : celle du bloc et celle de la boucle **for**. Nous continuons volontairement d'itérer après la valeur d'indice zéro dans le but de voir si la boucle continue où non selon la façon dont nous gérons les exceptions.

```
PS > &{
>> $ErrorActionPreference = 'continue' #pas obligatoire car mode par défaut
>> for ($i=-2; $i -le 2; $i++)
>> {
>>     trap { 'Erreur critique détectée !' }
>>     100/$i
>> }
>> Write-host 'suite...'
>>}
>>
-50
-100
Erreur critique détectée !
Tentative de division par zéro.
Au niveau de ligne : 6 Caractère : 11
+      100/$ <<<< i
100
50
suite...
```

Nous pouvons remarquer ceci :

- Le gestionnaire a détecté l'exception et a bien affiché notre message d'erreur personnalisé.
- Le message d'erreur standard s'est affiché.
- La boucle ne s'est pas arrêtée après l'exception et les autres itérations ont eu lieu.



Comme rien ne lui a été spécifié, notre gestionnaire s'est basé sur la valeur de **\$ErrorActionPreference** pour déterminer son comportement. Comme cette variable était en mode **Continue**, l'exécution du script a continué normalement mais le message d'erreur s'est affiché. Si nous avons indiqué l'instruction **Continue** à notre gestionnaire, nous aurions eu le même résultat mais cette fois sans message d'erreur.

Essayons cette fois de forcer l'instruction **Continue** dans notre gestionnaire d'interception et de donner à notre variable **\$ErrorActionPreference** la valeur **stop**.

```
PS > &{
>> $ErrorActionPreference = 'stop'
>> for ($i=-2; $i -le 2; $i++)
>> {
>>     trap { 'Erreur critique détectée !' ; continue }
>> }
```

```
>>      100/$i
>>    }
>>    Write-host 'suite...'
>> }
>>
-50
-100
Erreur critique détectée !
100
50
suite...
```

Nous remarquons ceci :

- Le gestionnaire a détecté l'exception et a bien affiché notre message d'erreur personnalisé.
- Il n'y a pas eu d'affichage du message d'erreur standard.
- La boucle ne s'est pas arrêtée après l'exception et les itérations suivantes ont eu lieu.

Nous pouvons constater que quelle que soit la valeur de la variable **\$ErrorActionPreference**, un gestionnaire d'interception n'en tient pas compte lorsque nous lui disons quoi faire.

Dernière déclinaison de notre exemple 1, nous allons remettre **\$ErrorActionPreference** à **Continue** et spécifier l'instruction **Break** à notre gestionnaire. Ainsi nous aurons balayé presque tous les cas de figure.

```
PS > &{
>>   $ErrorActionPreference = 'continue'
>>   for ($i=-2; $i -le 2; $i++)
>>   {
>>       trap { 'Erreur critique détectée !' ; break }
>>       100/$i
>>   }
>>   Write-host 'suite...'
>> }
>
-50
-100
Erreur critique détectée !
Tentative de division par zéro.
Au niveau de ligne : 6 Caractère : 12
+      100/$ << i
```

Cette fois nous remarquons que :

- Le gestionnaire a détecté l'exception et a bien affiché notre message d'erreur personnalisé.
- Le message d'erreur standard s'est affiché.
- Les itérations suivant l'exception ne se sont pas déroulées.
- Le script s'est arrêté (le texte « suite... » ne s'étant pas affiché).

Exemple 2 :

Jouons avec plusieurs gestionnaires d'interception !

Dans cet exemple, nous allons créer deux gestionnaires, chacun dans sa propre étendue. L'idée est de montrer comment se comportent les gestionnaires d'interception lorsqu'il y en a un à différents niveaux.

Nous partirons de l'exemple précédent auquel nous ajouterons un gestionnaire d'interception dans l'étendue parente.



Pour l'exemple, nous nous contenterons d'afficher un simple message d'erreur à l'écran. Cependant, dans nos scripts en exploitation nous redirigeons la dernière exception dans un fichier. Ainsi, un script déclenché

par une tâche planifiée venant à « planter » durant la nuit génère un fichier log indiquant précisément ce qu'il s'est passé. Pour récupérer la dernière exception à l'intérieur d'un gestionnaire d'interception, il existe la variable `$_`. Celle-ci retourne non seulement l'exception au format texte, mais surtout l'objet erreur lui-même (de type **ErrorRecord**). Ainsi nous disposons de toutes ses propriétés et méthodes associées.

```
PS > &{
>> $ErrorActionPreference = 'continue'
>> trap { "Exception détectée dans l'étendue parent !" ; continue }
>> for ($i=-2; $i -le 2; $i++)
>> {
>>     trap { "Exception détectée dans l'étendue enfant !" ; break }
>>     100/$i
>> }
>> Write-host 'suite...'
>> }
>>
-50
-100
Exception détectée dans l'étendue enfant !
Exception détectée dans l'étendue parent !
suite...
```

Nous remarquons ceci :

- Le gestionnaire de la **boucle for** a détecté l'exception et a affiché notre message.
- Parce que **Break** a été spécifié dans le gestionnaire de la boucle, le gestionnaire parent (du bloc) a aussi détecté l'exception et a affiché notre message.
- Il n'y a pas eu d'affichage du message d'erreur standard.
- La boucle s'est interrompue après l'exception et les autres itérations n'ont pas eu lieu.
- Le script s'est terminé normalement par l'affichage de « suite... ».

Comme une erreur s'est produite dans l'étendue enfant et que son gestionnaire était en mode **Break**, l'exception s'est propagée à l'étendue parent et l'interpréteur a quitté l'étendue courante. Le gestionnaire d'interception de niveau supérieur étant en mode **Continue**, le script a donc continué son exécution sans donner lieu à un quelconque affichage d'erreur.

Mais que se serait-il passé si nous avions été en mode **Break** dans le gestionnaire parent ?

```
PS > &{
>> $ErrorActionPreference = 'continue'
>> trap { "Exception détectée dans l'étendue parent !" ; break }
>> for ($i=-2; $i -le 2; $i++)
>> {
>>     trap { "Exception détectée dans l'étendue enfant !" ; break }
>>     100/$i
>> }
>> Write-host 'suite...'
>> }
>>
-50
-100
Exception détectée dans l'étendue enfant !
Exception détectée dans l'étendue parent !
Tentative de division par zéro.
Au niveau de ligne : 7 Caractère : 12
+      100/$ <<<< i
```

Et bien comme on pouvait le prévoir, le script s'est arrêté au niveau de l'exception et le message d'erreur standard a eu lieu.

Ça y est, vous commencez à comprendre le fonctionnement des gestionnaires d'interception ? Très bien, alors dans ce cas passons à la vitesse supérieure.

Exemple 3 :

Jouons avec plusieurs gestionnaires d'interception de types différents !

Quand nous écrivons « de types différents » il faut comprendre que les gestionnaires vont cette fois-ci non plus intercepter la première exception qui vient, mais plutôt intercepter les exceptions d'un type donné.

Continuons sur l'exemple précédent, sauf que cette fois nous allons tenter de choisir les exceptions qui nous intéressent. Pour ce faire, nous avons typé nos gestionnaires d'interceptions dans l'étendue enfant (celle de la **bouclefor**). Nous n'avons pas typé le gestionnaire de l'étendue parente car comme cela il peut intercepter n'importe quelle exception venue de l'étendue enfant.

D'autre part, nous avons introduit dans la boucle une nouvelle instruction qui va tenter de lister le contenu d'un répertoire qui n'existe pas. Et comme par défaut une telle commande ne génère que des erreurs non-critiques, nous l'avons forcé à générer des erreurs critiques grâce à **\$ErrorActionPreference = 'stop'**.

Donc en résumé, dès la première itération une exception aura lieu à cause du **Get-ChildItem** mais comme celle-ci sera interceptée par le gestionnaire associé et que celui-ci fonctionne en « **mode Continue** », le script continuera à se dérouler normalement. Puis l'exception fatidique de la division par zéro arrivera à la troisième itération, entraînant l'arrêt du script à cause des deux gestionnaires en « **mode break** ».

```
PS > &{
>> $ErrorActionPreference = 'stop'
>> trap { "Exception détectée dans l'étendue parent !" ; break}
>> for ($i=-2 ; $i -le ; $i++)
>> {
>> trap [System.DivideByZeroException] {
>>   'Attention, division par zero !' ; break}
>> trap [System.Management.Automation.ActionPreferenceStopException]
>> {
>>   "Le repertoire indiqué n'existe pas !"; continue }
>>   Write-Host "--> Itération No $i <--"
>>   100/$i
>>   Get-ChildItem "D:\Scripts\Test"
>> }
>> Write-Host 'Suite...'
>> }
>>
--> Itération No -2 <--
-50
Le repertoire indiqué n'existe pas !
--> Itération No -1 <--
-11
Le repertoire indiqué n'existe pas !
--> Itération No 0 <--
Attention, division par zero !
Exception détectée dans l'étendue parent !
Tentative de division par zéro.
Au niveau de ligne : 11 Caractère : 12
+ 100/$ <<<< i
```

Vous vous demandez certainement comment connaître à l'avance le type de l'exception que nous voulons intercepter ? Et bien c'est ce que nous allons voir dans la partie qui suit.



Bien qu'il soit possible de transformer les erreurs non-critiques en erreurs critiques, en donnant la valeur **stop** à **\$ErrorActionPreference**, et de les intercepter en tant que tel, nous ne vous conseillons pas de le faire. La raison est simple : l'exception qui est levée est de type **System.Management.Automation.ActionPreferenceStopException**. Ce type d'exception nous indique simplement qu'à cause de la valeur **\$ErrorActionPreference='stop'** une exception a eu lieu, sans nous indiquer quelle en est l'origine. Ainsi nous ne pourrions pas savoir dans un bloc de script contenant plusieurs commandes, qui potentiellement peuvent générer cette erreur, laquelle d'entre elles a rencontré un problème. C'est la raison pour laquelle, nous vous préconisons de préférer les mécanismes de gestion des erreurs adaptés à chaque type.

2. Déterminer le type des erreurs critiques

Pour connaître le type d'une exception, et bien le plus simple est de la provoquer puis d'aller voir son type dans les propriétés de la variable `$Error[0]` (ou `$Error[1]` dans certains cas).

Prenons l'exemple de la division par zéro :

```
PS > &{
>> $zero = 0
>> 1/$zero
>> $Error[0] | Format-List * -Force
>> }
>>
Tentative de division par zéro.
Au niveau de ligne : 3 Caractère : 3
+ 1/$ <<< zero

PSMessageDetails      :
Exception              : System.Management.Automation.RuntimeException:
                        Tentative de division par zéro.
                        ---> System.DivideByZeroException: Tentative
                        de division par zéro.
                        à System.Management.Automation.ParserOps.
                        polyDiv(ExecutionContext context, Token opToken,
                        Object lval, Object rval)
                        --- Fin de la trace de la pile
                        d'exception interne ---
                        à System.Management.Automation.Parser.
                        ExpressionNode.Execute(Array input, Pipe
                        outputPipe)
                        à System.Management.Automation.ParseTree
                        Node.Execute(Array input, Pipe
                        outputPipe, ArrayList& resultList)
                        à System.Management.Automation.Parser.
                        StatementListNode.Execute(Array input,
                        Pipe outputPipe, ArrayList& resultList)
TargetObject           :
CategoryInfo           : NonSpécifié : (:) [], RuntimeException
FullyQualifiedErrorId  : RuntimeException
ErrorDetails           :
InvocationInfo          : System.Management.Automation.InvocationInfo
PipelineIterationInfo  : {}
```

Nous venons provoquer l'exception qui nous intéresse, et maintenant grâce au contenu de `$Error[0]` nous avons déterminé que son type est **System.DivideByZeroException**.

Désormais, nous pouvons par exemple intercepter l'erreur de division par zéro de la façon suivante :

```
Trap [System.DivideByZeroException] { 'Une tentative de
division par zéro a été détectée !' ; break }
```

3. Génération d'exceptions personnalisées

Grâce à l'instruction **throw**, vous allez pouvoir vous amuser à créer vos propres exceptions.

La syntaxe est la suivante :

```
throw ["Texte à afficher lors de la levée de l'exception."]
```

Sachez que vous n'êtes pas obligés de spécifier une chaîne de caractères après l'instruction **throw**. Celle-ci peut s'utiliser seule dans sa plus simple expression.

Exemple :

```
PS > &{
>> $errorActionPreference = 'continue'
>> trap { 'Exception détectée : ' + $_ }
```

```
>>
>>     throw 'Ma première exception personnalisée !'
>> }
>>
Exception détectée : Ma première exception personnalisée !
Ma première exception personnalisée !
Au niveau de ligne : 4 Caractère : 9
+     throw <<<< "Ma première exception personnalisée !"
```

L'instruction **throw** est souvent utilisée de concert avec des fonctions ou des scripts qui nécessitent des paramètres obligatoires pour fonctionner.

Exemple :

Cas d'une fonction.

```
Function Bonjour ($prenom = $(throw 'Il manque le paramètre -Prenom'))
{
    Write-Host "Bonjour $prenom" -ForegroundColor green
}
```

Si vous omettez de spécifier le paramètre au lancement de la fonction, vous obtiendrez ceci :

```
PS > bonjour
Il manque le paramètre -Prenom
Au niveau de ligne : 1 Caractère : 36
+ Function Bonjour ($prenom = $(throw <<<< 'Il manque le paramètre
-prenom'))
```

Exemple :

Cas d'un script.

Créez un script nommé par exemple **Bonjour.ps1** et insérez ces quelques lignes à l'intérieur :

```
param([string]$prenom = $(throw 'Il manque le paramètre -Prenom'))

Write-Host "Bonjour $prenom" -ForegroundColor green
```

Puis exécutez votre script ainsi : `./bonjour.ps1 -Prenom Arnaud`



Il est possible, mais pas nécessairement recommandé, de remplacer le texte entre guillemets après l'instruction **throw** par un bloc de script.

Exemple :

```
Function Bonjour ($prenom = $(throw `
    &{ write-host 'Manque paramètre -Prenom' -ForegroundColor green
        Get-Childitem c:\
    }))
{
    Write-Host "Bonjour $prenom" -ForegroundColor red
}
```

Les créateurs de PowerShell ont simplifié au maximum le moyen de créer des exceptions, sachez cependant qu'il est possible de passer directement à la place du texte, des objets de type **ErrorRecord** ou des exceptions .NET.

4. Gérer les erreurs critiques avec Try-Catch-Finally

Comme nous le disions précédemment, lorsqu'une erreur critique survient, l'exécution de la commande, ou du script dans certains cas, est interrompue. Pour éviter d'avoir à gérer les erreurs d'une façon manuelle, PowerShell v2 ajoute

la possibilité d'utiliser des blocs d'exécution Try, Catch et Finally pour définir des sections dans lesquelles PowerShell va surveiller les erreurs. Try, Catch et Finally abordent un mode de fonctionnement déjà bien rôdé dans de nombreux langages de développement.

Le bloc **try** contient le code protégé risquant de provoquer l'exception. Le bloc est exécuté jusqu'à la levée d'une exception (exemple : division par zéro) ou jusqu'à sa réussite totale. Si une erreur avec fin d'exécution se produit pendant l'exécution des instructions, PowerShell passe l'objet erreur du bloc Try à un bloc Catch approprié.

La syntaxe du bloc Try est la suivante :

```
try {<bloc d'instructions>}
```

La clause **catch** contient le bloc d'exécution associé à un type d'erreur interceptée. Catch peut être utilisée sans argument. Dans ce cas, elle intercepte tout type d'exception et est appelée la clause catch générale.

La syntaxe du bloc Catch est la suivante :

```
catch [<type d'erreur>] {<bloc d'instructions>}
```



Voir section Déterminer le type des erreurs critiques dans ce chapitre pour savoir comment connaître les types des erreurs critiques.

Optionnel, le **Finally** s'utilise suivi d'un bloc de script qui s'exécute chaque fois que le script est exécuté. Et ce, qu'une erreur ait été interceptée ou non.

La syntaxe du bloc Finally est la suivante :

```
finally {<bloc d'instructions>}
```

Illustrons tous cela avec une tentative de division par Zero :

```
Function Boucle
{
  For ($i=-2 ;$i -le 2;$i++)
  {
    Try { 100/$i }
    Catch { Write-Host 'Erreur dans le script !' }
  }
}
```

Résultat :

```
PS > boucle
-50
-100
Erreur dans le script !
100
50
```

L'opération 100/\$i est testée à chaque passage dans la boucle de façon à déterminer si une remontée d'erreur se produit. Si une erreur se produit, alors la commande Write-Host sera exécutée. Cependant, dans l'exemple ci-dessus, le bloc d'instructions ne tient pas compte du type d'erreur rencontrée. Pour filtrer sur un ou plusieurs types d'erreurs, il suffit de le préciser avec le bloc d'instructions, comme ci-dessous :

```
Function Boucle
{
  For ($i=-2 ;$i -le 2;$i++)
  {
    Try {100/$i}
    Catch [System.DivideByZeroException] {
      Write-Host 'Erreur dans le script !'
    }
  }
}
```

Résultat :

```
PS > boucle
-50
-100
```


Erreur dans le script !

100

50

Ainsi, l'exécution du bloc Catch est liée à une erreur commise par une division par zéro et uniquement cela.

Comme vous l'avez sans aucun doute remarqué, la finalité de l'interception d'erreur critique avec Trap et l'utilisation de Try-Catch est particulièrement proche. Pourtant, il existe quelques différences expliquées ci-dessous :

Trap	Try/Catch
Disponible dans PowerShell v1 et v2.	Uniquement disponible dans PowerShell v2.
Conçu pour une utilisation simple par les administrateurs système.	Conçu pour une utilisation orientée développement.
Peut intercepter une erreur générée dans la portée globale du script/fonction.	Peut intercepter uniquement une erreur générée dans la portée du bloc d'instruction Try.

Le débogage

PowerShell, côté débogage, est doté de fonctionnalités assez riches par rapport à son cousin VBScript. Et cela est d'autant plus vrai avec PowerShell v2 qui intègre la notion de point d'arrêt et d'exécution pas à pas de façon graphique.

Il y a maintes et maintes façons de déboguer un script. La plus basique étant d'intercaler dans un script des affichages de variables ou de messages ici et là pour essayer de trouver la ou les erreurs. Ceci étant dit, avec PowerShell nous verrons que nous pouvons faire mieux que placer des **Write-Host** de débogage un peu partout dans notre script et ainsi « polluer » notre code.

Une autre méthode de base de débogage pourrait aussi consister à forcer la déclaration des variables ; ce que nous verrons également.

1. Affichage de messages en mode verbose

Il existe dans PowerShell un mode verbeux. Celui-ci permet aux scripts ou aux commandelettes (et elles sont nombreuses) qui possèdent des informations complémentaires de s'afficher.

Pour écrire une information visible uniquement en « mode verbose », il faut utiliser la commandelette **Write-Verbose**. Ceci est la première chose à faire mais ce n'est pas suffisant, car pour permettre à vos informations de s'afficher sur la console, il faut ajuster la valeur de la variable **\$VerbosePreference** qui par défaut est positionnée à **SilentlyContinue**. Cela signifie que, par défaut, les informations complémentaires ne sont pas affichées. Les autres valeurs possibles sont les mêmes que pour la variable **\$ErrorActionPreference**, à savoir **Continue**, **Stop**, et **Inquire**.

Exemple :

Essayons d'écrire une information en mode par défaut.

```
PS > $VerbosePreference
SilentlyContinue
PS > Write-Verbose 'Ceci est un test !'
PS >
```

Comme prévu, il ne se passe rien. Voyons maintenant ce qu'il se passe en mettant la valeur **Continue** à la variable **\$VerbosePreference**.

```
PS > $VerbosePreference = 'continue'
PS > Write-Verbose 'Ceci est un test !'
COMMENTAIRES : Ceci est un test !
PS >
```

Nous sommes obligés de remarquer que notre chaîne de caractères commence par « COMMENTAIRES : » et surtout qu'elle s'affiche en jaune, ce qui est parfaitement visible au milieu d'un affichage standard.



Le moins que l'on puisse dire c'est que le jaune ne ressort pas très bien sur un tirage imprimé en noir et blanc. C'est la raison pour laquelle nous avons fait apparaître en gras ce que vous verriez normalement en jaune sur un écran.

En mode **Stop**, le message s'affiche mais l'exécution s'arrête car une exception est levée ; tandis qu'en mode **Inquire** le menu de choix habituels nous est proposé.

2. Affichage de messages en mode debug

Le mode debug pour l'affichage des messages fonctionne exactement de la même manière que la commandelette **Write-Verbose**, aux différences près :

- L'écriture d'un message de débogage s'effectue avec **Write-Debug**.
- La variable de préférence à ajuster est **\$DebugPreference**.

- Le message affiché commencera par « DÉBOGUER : ».

Exemple :

```
PS > $DebugPreference = 'continue'
PS > Write-Debug 'Ceci est une information de débogage.'
DÉBOGUER : Ceci est une information de débogage.
PS >
PS > $DebugPreference = 'stop'
PS > Write-Debug 'Ceci est une information de débogage.'
DÉBOGUER : Ceci est une information de débogage.
Write-Debug : L'exécution de la commande s'est arrêtée, car la variable
d'environnement « DebugPreference » a la valeur Stop.
Au niveau de ligne : 1 Caractère : 12
+ Write-Debug <<<< 'Ceci est une information de débogage.'
PS >
```

3. Affichage de messages en mode warning

Le dernier mode d'affichage possible pour des messages d'états est le **mode avertissement**. Il fonctionne de la même façon que les deux précédents modes (**verbose** et **debug**), aux différences près :

- L'écriture d'un message d'avertissement s'effectue avec **Write-Warning**.
- La variable de préférence à ajuster est **\$WarningPreference**.
- Le message affiché commencera par « AVERTISSEMENT : ».



Au lieu de manipuler les variables de préférence que nous venons de voir, il est possible, pour chaque commande, d'utiliser les paramètres communs suivants :

- **Verbose** : pour afficher les informations complémentaires s'il y en a.
- **Debug** : pour afficher un menu de choix de type « Inquire » lorsqu'il se produit l'une des choses suivantes : affichage d'un message de débogage, d'informations ou d'avertissement, erreur non-critique.
- **Confirm** : demande à l'utilisateur une confirmation avant d'exécuter une commande qui modifie l'état du système.

4. Forcer la définition des variables

Avec PowerShell vous n'êtes pas obligés de définir vos variables. Une simple affectation de valeur suffit à déclarer une variable car PowerShell se charge du reste.

Peut-être avez-vous déjà remarqué qu'une variable non définie est une **chaîne vide** pour le cas d'une chaîne de caractère, et **zéro** pour un entier ? En principe PowerShell affecte la valeur **\$null** aux variables qui ne sont pas définies.

Voici un cas d'erreur classique où nous faisons une erreur dans le nom d'une variable :

```
PS > $maVariable = 25
PS > $Total = $maVaraible * 12
```

Cela va produire, on s'en doute, un résultat imprévisible mais surtout imprévu !

Pour éviter cela, et forcer la déclaration de toutes les variables, PowerShell met à notre disposition la commande **Set-PSDebug** suivie du paramètre **-Strict**.



Set-PSDebug-Strict correspond dans l'esprit à « **Option Explicit** » en VBScript mais en moins restrictif.

Reprenons notre exemple pour voir le changement :

```
PS > Set-PSDebug -Strict
PS > $maVariable = 25
PS > $Total = $maVariable * 12
```

La variable \$maVariable ne peut pas être récupérée, car elle n'a pas encore été définie.

Au niveau de ligne : 1 Caractère : 21

```
+ $Total = $maVariable <<<< * 12
```

Le message est très clair : nous n'avons pas défini **\$maVariable** ; ce qui est normal vu que nous avons dérapé sur le clavier. Avec une telle indication, difficile de ne pas trouver son erreur.

Avec la version 2, PowerShell va encore plus loin dans la définition de variables grâce à la commandelette **Set-StrictMode**. Très proche du fonctionnement de **Set-PSDebug** suivie du paramètre **-Strict**, **Set-StrictMode** ne permet pas seulement de repérer les erreurs concernant les variables non initialisées, mais également celles provoquées par des propriétés qui n'existent pas. En fait, **Set-StrictMode** peut exploiter les différents niveaux de version définis ci-dessous :

Version	Définitions concernées
Version 1.0	• Interdit les références aux variables non initialisées, à l'exception de celles présentes dans les chaînes.
Version 2.0	• Interdit les références aux variables non initialisées (notamment les variables non initialisées présentes dans des chaînes). • Interdit les références à des propriétés inexistantes d'un objet. • Interdit les appels de fonctions qui utilisent la syntaxe pour l'appel de méthodes. • Interdit une variable sans nom (\${}).
Version Latest	• Sélectionne la version la plus récente (la plus stricte) disponible. Utilisable dans les versions futures de PowerShell.

Par exemple, prenons le cas typique d'une propriété qui n'existe pas. Même en activant le mode **Set-PSDebug -Strict**, aucune erreur n'est levée.

```
PS > Set-PSDebug -Strict
PS > $maVariable = 25
PS > $mavARIABLE.jutiliseuneproprietequinexistepas
```

À présent, si nous utilisons la commandelette **Set-StrictMode** dans sa version 2.0 (qui interdit les références à des propriétés qui n'existent pas), une erreur sera levée cette fois-ci.

```
PS > Set-StrictMode -Version 2.0
PS > $maVariable = 25
PS > $mavARIABLE.jutiliseuneproprietequinexistepas
La propriété « jutiliseuneproprietequinexistepas » est introuvable sur
cet objet. Vérifiez qu'elle existe.
Au niveau de ligne : 1 Caractère : 13
+ $maVariable. <<<< jutiliseuneproprietequinexistepas
+ CategoryInfo          : InvalidOperation: (.:OperatorToken) [],
RuntimeException
+ FullyQualifiedErrorId : PropertyNotFoundStrict
```

Le message est une nouvelle fois très clair : nous n'avons pas la propriété citée, par conséquent le script s'arrête.

Voilà donc une bonne habitude à prendre pour vous faire gagner du temps dans le développement de vos scripts !

5. Exécuter un script pas à pas

Exécuter un script pas à pas, mettre des points d'arrêts, inspecter des variables durant l'exécution d'un script ; toutes ces choses font partie du rêve de tout scripteur ayant déjà goûté à un langage de développement de haut niveau tels que le Visual Basic ou le C++. Et bien sachez que tout cela n'est plus un rêve, mais belle et bien une réalité avec PowerShell.

Pour entrer dans le mode d'exécution pas à pas, il vous faut utiliser la commande suivante :

```
Set-PSDebug -Step
```

L'exécution pas à pas va permettre l'exécution d'un script ligne après ligne, et pour chaque ligne vous allez devoir indiquer à l'interpréteur de commandes ce qu'il doit faire. Vous aurez les possibilités suivantes :

- **Oui** (touche « O » ou « Entrée ») : exécute la commande.
- **Oui pour tout** (touche « T ») : quitte le mode pas à pas et exécute le script jusqu'à la fin.
- **Non** (touche « N ») : refuse l'exécution de la commande courante.
- **Non pour tout** (touche « U ») : refuse l'exécution de toutes les commandes jusqu'à la fin du script.
- **Suspendre** (touche « S ») : suspend l'exécution du script en cours et entre dans un interpréteur de commandes imbriqué.

Prenons l'exemple suivant :

```
PS > Set-PSDebug -Step
PS > For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
```

Résultat :

```
Voulez-vous continuer cette opération ?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S] Suspendre
[?] Aide
```

Nous allons répondre « **Oui** », trois fois de suite et voir ce qu'il se passe.

```
PS > Set-PSDebug -Step
PS > For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}

Voulez-vous continuer cette opération ?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout
[S] Suspendre [?] Aide
(la valeur par défaut est « 0 ») :
DÉBOGUER : 1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}

Voulez-vous continuer cette opération ?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout
[S] Suspendre [?] Aide
(la valeur par défaut est « 0 ») :
DÉBOGUER : 1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
Bonjour 1

Voulez-vous continuer cette opération ?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout
[S] Suspendre [?] Aide
(la valeur par défaut est « 0 ») :
DÉBOGUER : 1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
Bonjour 2
```

```
Voulez-vous continuer cette opération ?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout
[S] Suspendre [?] Aide
(la valeur par défaut est « 0 ») :
```

On valide une première fois pour confirmer l'exécution de la commande, puis les fois suivantes on valide pour exécuter chaque itération. Vous remarquerez qu'à chaque itération nous avons droit à l'affichage du résultat, exactement comme si nous exécutions notre script normalement.

Nous allons à présent entrer en mode débogage en choisissant de suspendre l'exécution du script en pressant la touche « S ». En faisant cela nous allons entrer dans un sous-shell ou shell imbriqué. À partir de ce moment là, nous serons dans une nouvelle instance PowerShell et nous pourrions examiner le contenu des variables en cours d'exécution, et même les modifier.

```
Voulez-vous continuer cette opération ?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S]
Suspendre [?] Aide
(la valeur par défaut est « 0 ») :S
PS >>>
PS >>> $i
3
PS >>> $i=-2
PS >>> exit

Voulez-vous continuer cette opération ?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout
[S] Suspendre [?] Aide
(la valeur par défaut est « 0 ») :
DÉBOGUEUR : 1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
Bonjour -2

Voulez-vous continuer cette opération ?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Bonjour $i"}
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout
[S] Suspendre [?] Aide
(la valeur par défaut est « 0 ») :
```

En entrant dans un shell imbriqué, vous constaterez qu'un prompt légèrement différent de celui que nous avons d'habitude s'offre à nous (car nous avons un double chevron en plus « >> »).

Nous avons demandé la valeur de **\$i** (qui vaut 3), puis nous l'avons modifié à la valeur -2. Nous aurions également pu faire tout un tas d'autres choses, comme lancer des commandelettes ou des scripts. Enfin nous avons quitté le sous-shell grâce à la commande **exit**, et le mode pas à pas a repris son cours, comme si rien ne s'était passé alors même que nous avons modifié **\$i**.

Voilà toute la puissance de PowerShell ! Pouvoir déboguer un script PowerShell avec lui-même, c'est épatant vous ne trouvez pas ?



Il faut savoir que l'entrée dans un shell imbriqué peut se faire à tout moment, dès lors que vous utilisez la commande suivante : `$host.EnterNestedPrompt()` Pour savoir si vous êtes dans le shell principal ou dans un shell imbriqué, allez consulter la valeur de la variable **\$NestedPromptLevel**. Si celle-ci est différente de **zéro**, c'est que vous êtes dans un shell imbriqué.



Le fait que l'invite PowerShell se transforme en ajoutant deux chevrons « >> » supplémentaires est dû à la définition de la fonction **Prompt**. Celle-ci est définie ainsi à l'installation de PowerShell (voir chapitre Maîtrise du shell). Si vous modifiez la fonction **Prompt**, ayez conscience qu'il se peut que vous ayez un affichage différent.



Pour revenir dans un mode d'exécution normal et désactiver le mode pas à pas, la commande à saisir est `Set-PSDebug -Off`.

6. Les points d'arrêts (break points) avec PowerShell v1

Avant même de commencer à présenter l'utilisation des points d'arrêts, il est indispensable de dissocier l'utilisation de PowerShell v1 et de PowerShell v2. L'utilisation des points d'arrêts avec PowerShell v1, décrite ci-dessous, n'est sans commune mesure comparable à celle de PowerShell v2. C'est pour ces raisons que, nous vous encourageons à utiliser la v2 en cas d'utilisation intensive de points d'arrêts. Cependant, si pour des raisons particulières, vous souhaitez placer des points d'arrêts, voici comment procéder.

Comme nous venons de l'apprendre précédemment, nous pouvons utiliser la méthode **EnterNestedPrompt()** de l'objet **\$host** afin de suspendre l'exécution d'un script et entrer dans un shell imbriqué. Cela revient en fait à créer ce que l'on appelle couramment un « **point d'arrêt** ». Nous pouvons donc à tout moment, dans un script, utiliser la commande **\$host.EnterNestedPrompt()** si cela nous fait plaisir.

Ceci étant, sur le Blog (<http://blogs.msdn.com/powershell>) des tous puissants créateurs de PowerShell, on peut trouver une petite fonction intéressante pour créer des points d'arrêts ; et ce de façon, dirons-nous, plus élégante que de disséminer des **\$host.EnterNestedPrompt()**.

La voici :

```
function Start-Debug
{
    $scriptName = $MyInvocation.ScriptName
    function Prompt
    {
        "Debugging [{0}]> " -F $(if ([String]::IsNullOrEmpty
            $scriptName)) { 'globalscope' } else { $scriptName } )
    }
    $host.EnterNestedPrompt()
}

Set-Alias bp Start-Debug
```

Cette fonction va modifier le prompt afin de faire un peu mieux que le standard « >> » en vous indiquant dans quelle étendue vous vous trouvez (globalscope ou celle du script). Cette information sera obtenue par **\$MyInvocation.ScriptName**. Puis l'alias « **bp** », pour « **break point** », sera créé afin de faciliter l'usage de la fonction.

Exemple :

Regardons le résultat si vous tapez simplement « **bp** » dans l'interpréteur de commandes.

```
PS > bp
Debugging [globalscope]> $NestedPromptLevel
1
Debugging [globalscope]> exit
PS >
```

Pratique et élégant, n'est-ce pas ?

Cette fonction trouverait parfaitement sa place dans votre profil pour être pleinement utile et éviter de la rechercher lorsqu'on en a besoin.

7. Les points d'arrêts (break points) avec PowerShell v2

La gestion des points d'arrêts est grandement améliorée et enrichie dans la version 2 de PowerShell. Alors que dans la version 1.0 de PowerShell nous étions obligés de créer nos propres fonctions pour déboguer nos scripts (voir précédemment), la v2 apporte son lot de nouvelles commandes décrites ci-dessous :

Commande	Description
Set-PsBreakpoint	Permet de définir un point d'arrêts.
Get-PsBreakpoint	Permet de lister les points d'arrêts.
Disable-PsBreakpoint	Permet de désactiver les points d'arrêts.

Enable-PsBreakpoint	Permet d'activer les points d'arrêts.
Remove-PsBreakpoint	Permet de supprimer les points d'arrêts.
Get-PsCallStack	Permet d'afficher la pile des appels.

Exemple d'utilisation :

Prenons par exemple la fonction suivante qui nous retournera la taille libre en giga-octets (Go) du disque C :, ainsi que l'espace disque total de tous nos lecteurs.

```
Function Get-FreeSpace {
# Création de l'instance de l'objet WMI
$elements = Get-WmiObject Win32_LogicalDisk

$taille_totale = 0 # initialisation de la variable

# Boucle pour parcourir tous les disques
foreach ( $disque in $elements ) {
    if ($disque.Name -Like 'C:') {
        # Calcul de la taille en Giga octet
        $taille = $disque.freespace / 1GB
        $taille = [math]::round($taille, 1) #Arrondi la taille à 1 décimale
        Write-Host "Le disque $($disque.Name) a $taille Go de disponibles"
    }
    $taille_totale = $taille_totale + $taille
}
Write-Host "Taille disponible cumulée = $taille_totale Go"
}
```

Plaçons à présent un point d'arrêt sur l'entrée de fonction :

```
PS > Set-PsBreakpoint -Command Get-FreeSpace
ID Script      Line Command      Variable      Action
--
0             1 Get-FreeSpace
```

À l'exécution de la fonction, le mode débogage s'active :

```
PS > Get-FreeSpace
Passage en mode débogage. Utilisez h ou ? pour obtenir de l'aide.
Appuyez sur Point d'arrêt de commande sur « Get-FreeSpace »
Get-FreeSpace
[DBG]: PS >>>
```

Lorsque le prompt PowerShell affiche [DBG], cela signifie que vous vous situez dans l'environnement de débogage de PowerShell. Pour naviguer dans le débogueur PowerShell, voici les commandes :

Commande débogueur	Description
S « Step-Into »	Exécute l'instruction suivante, puis s'arrête.
V « Step-Over »	Exécute l'instruction suivante, mais ignore les fonctions et les appels.
O « Step-Out »	Effectue un pas à pas hors de la fonction actuelle, en remontant d'un niveau si elle est imbriquée. Si elle se trouve dans le corps principal, l'exécution se poursuit jusqu'à la fin ou jusqu'au point d'arrêt suivant.
C « Continue »	Continue à s'exécuter jusqu'à ce que le script soit terminé ou que le point d'arrêt suivant soit atteint.
L « List »	Affiche la partie du script qui s'exécute. Par défaut, la commande affiche la ligne en cours, les cinq lignes précédentes et les 10 lignes suivantes. Pour continuer à lister le script, appuyez sur [Entrée].

L <x> « List »	Affiche 16 lignes du début de script avec le numéro de ligne spécifié par la valeur <x>.
L <x> <n> « List »	Affiche <n> lignes du script commençant par le numéro de ligne spécifié par <x>.
G « Stop »	Arrête l'exécution du script et quitte le débogueur.
K « Get-PsCallStack »	Affiche la pile des appels.
<Entrée>	Répète la dernière commande s'il s'agit de Step (s), Step-over (v) ou List (l). Dans les autres cas, représente une action d'envoi.
?, h	Affiche l'aide sur les commandes du débogueur.

Exemple :

```
PS > Get-FreeSpace
Passage en mode débogage.
[DBG]: PS >>> S
$elements = Get-WmiObject Win32_LogicalDisk
[DBG]: PS >>> S
$taille_totale = 0 # initialisation de la variable
[DBG]: PS >>> S
foreach ( $disque in $elements ) {
[DBG]: PS >>> K

Command           Arguments          Location
-----
Get-FreeSpace      {}                 prompt
prompt             {}                 prompt

[DBG]: PS >>> Q
PS >
```

Pour enlever les points d'arrêts, il suffit d'utiliser la commande Remove-PSbreakpoint avec pour argument le nom ou l'ID du point d'arrêt. Exemple ci-dessous avec le point d'arrêt ayant pour ID 0 :

```
PS > Remove-PSbreakpoint -ID 0
```

C'est ainsi que vous pourrez naviguer avec le débogueur en mode console. Cependant, avec PowerShell ISE, le débogage peut également se réaliser graphiquement.

Dans l'encadré d'édition (en haut), il est possible de sélectionner une ligne souhaitée et d'y placer un point d'arrêt.

```
1 # FreeSpace.ps1
2 # Script calculant la mémoire libre de chaque disque logique
3
4 # Création de l'instance de l'objet WMI
5 $elements = get-WmiObject Win32_LogicalDisk
6
7 $staille_totale = 0 # initialisation de la variable
8
9 # Boucle pour parcourir tous les disques
10 foreach ( $disque in $elements ) {
11     if ($disque.Name -like 'C:') {
12         # Calcul de la taille en Giga octet
13         Staille = $disque.freespace / 1GB
14         Staille = [math]::round(Staille, 1) #Arrondi la taille à 1 décimale
15         write-host "Le disque $($disque.Name) a $staille Go de disponibles"
16     }
17     $staille_totale = $staille_totale + $staille
18 }
19 Write-Host "Taille disponible cumulée = $staille_totale Go"
20
```

PS C:\Users\Robin>

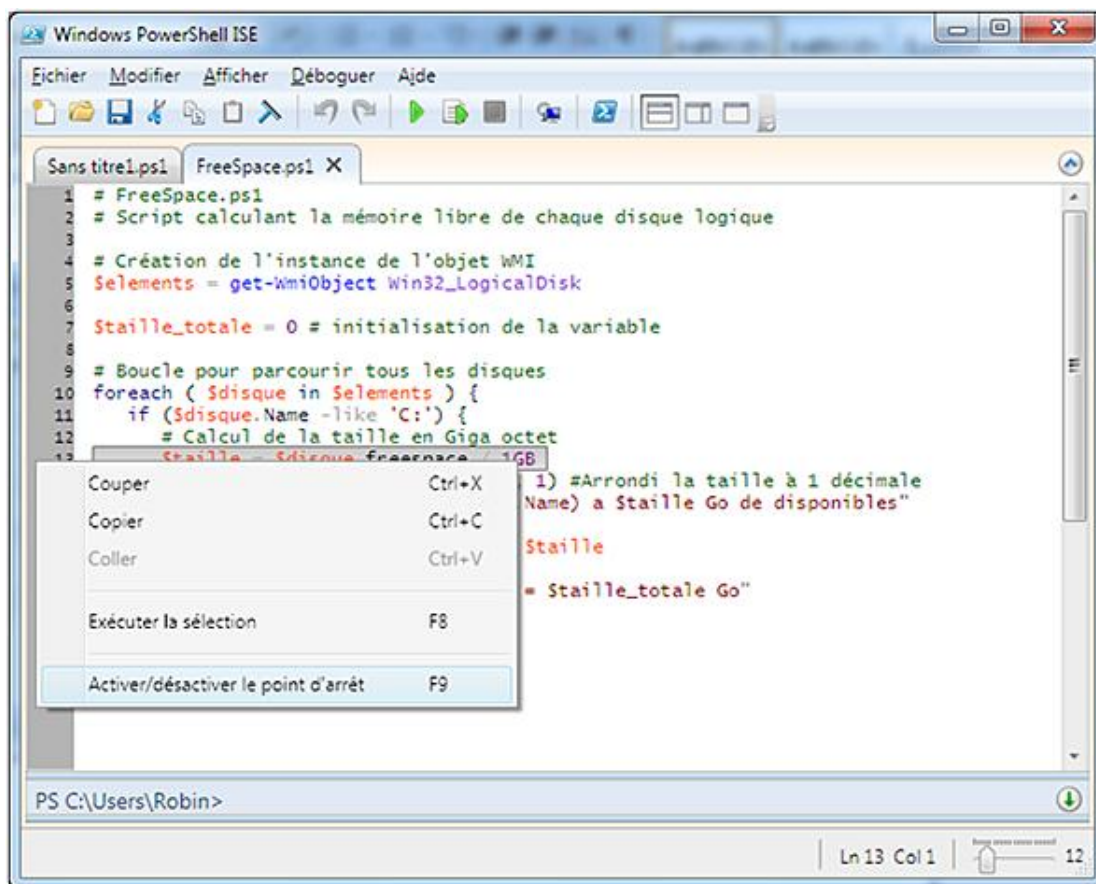
Ln 14 Col 1 12

Points d'arrêts via PowerShell ISE-1



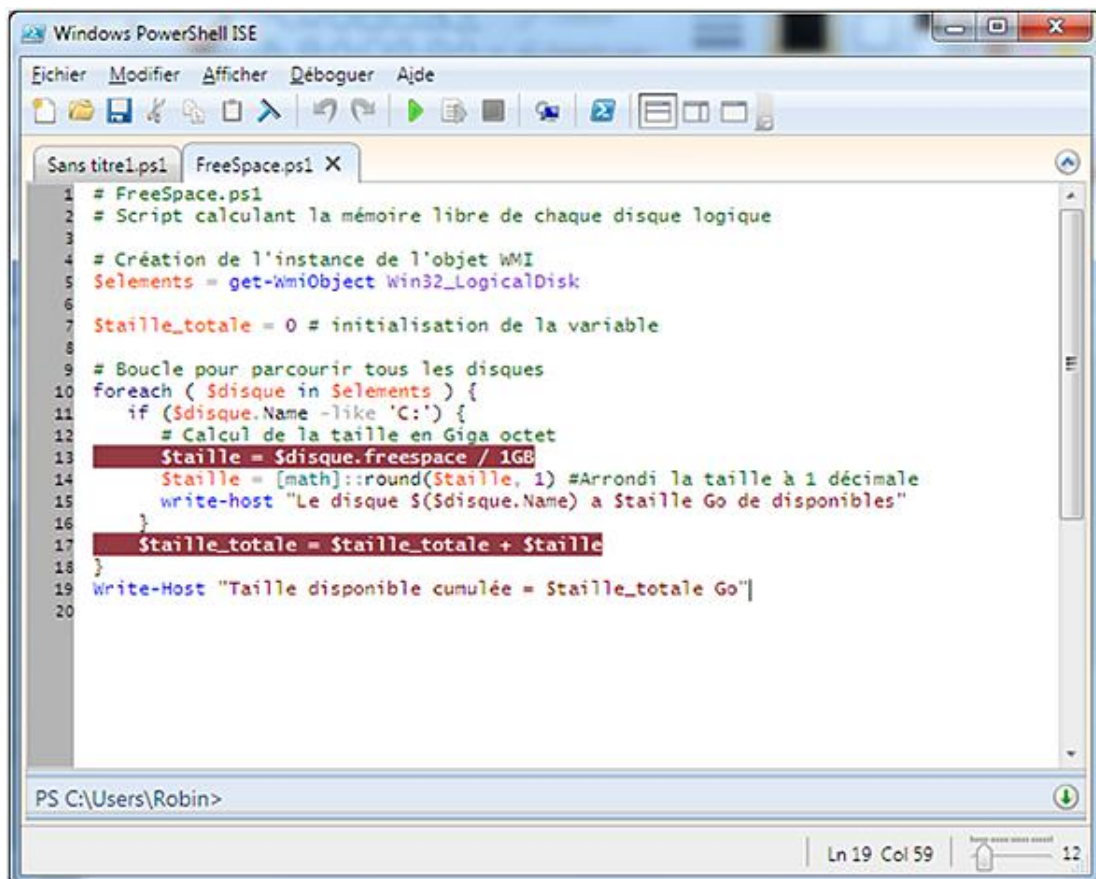
Pour pouvoir placer des points d'arrêts, PowerShell nécessite que le script en cours d'édition soit enregistré.

Le point d'arrêt se choisit en sélectionnant la ligne, puis en choisissant d'un clic droit l'option **Activer/désactiver le point d'arrêt**. Ou bien en pressant la touche [F9].

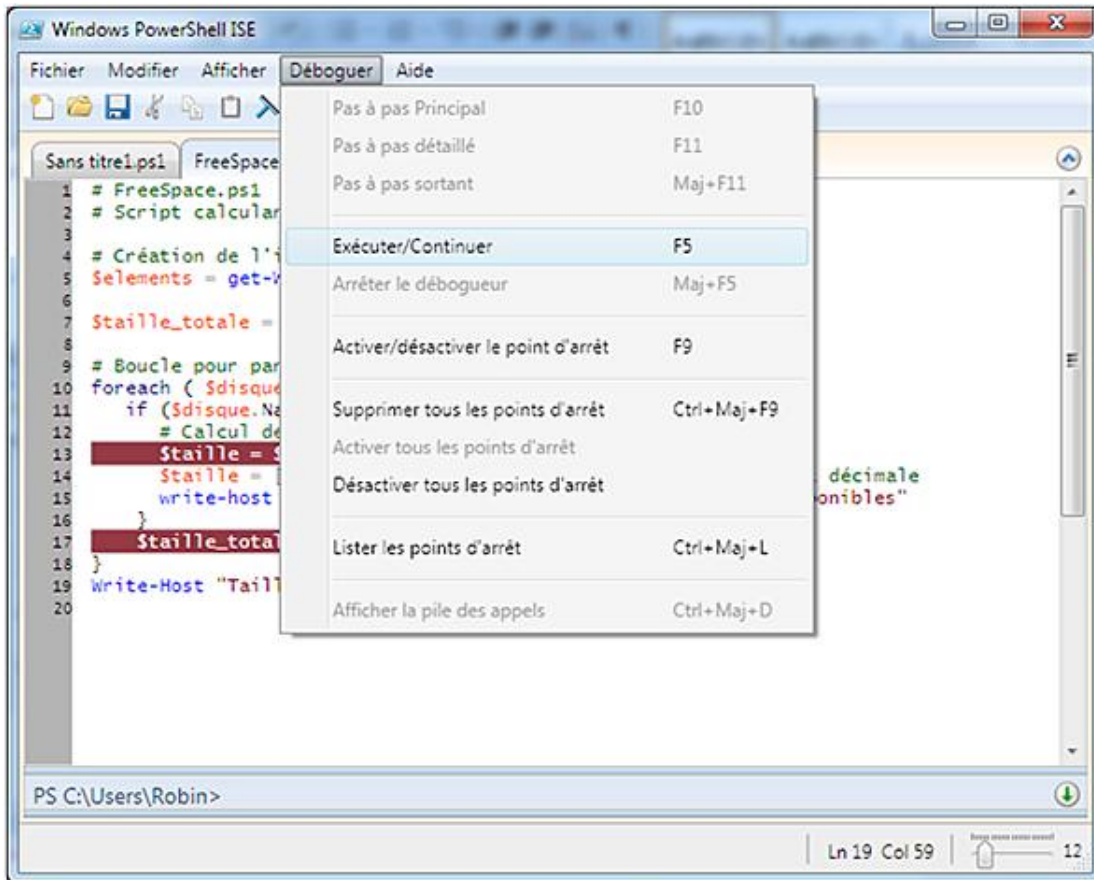


Points d'arrêts via PowerShell ISE-2

Plusieurs points d'arrêts peuvent être placés au sein d'un même script.



Enfin, l'exécution est soit réalisée par pression de la touche [F5], soit en choisissant **Exécuter/continuer** depuis le menu **Débuguer**.



Points d'arrêts via PowerShell ISE-4



Lors de l'exécution, l'état de chaque variable est visible en positionnant le pointeur de souris dessus.

8. Mode trace de Set-PSDebug

Le mode « trace » va nous permettre de comprendre comment un script est interprété par PowerShell ; nous verrons ainsi le résultat d'exécution de chaque traitement. Cela nous permettra, par exemple, de découvrir plus rapidement la source d'un bug.

L'activation du mode « trace » se fait de la façon suivante :

```
Set-PSDebug -Trace [1 | 2]
```

Il existe deux modes de trace, le premier « **-trace 1** », est le mode de base qui n'affiche que les traitements. Le seconde mode « **-trace 2** » est le mode détaillé qui affiche en plus des traitements, tous les appels de scripts ou de fonctions. On rencontre également les termes « **niveaux de suivi** » pour désigner ces modes.

Reprenons par exemple le script suivant qui nous retournera la taille libre en giga-octets du disque C :, ainsi que l'espace disque total de tous nos lecteurs.

```
# FreeSpace.ps1
# Script calculant la mémoire libre de chaque disque logique

# Création de l'instance de l'objet WMI
$selements = Get-WmiObject Win32_LogicalDisk

$staille_totale = 0 # initialisation de la variable
```

```
# Boucle pour parcourir tous les disques
foreach ( $disque in $elements ) {
    if ($disque.Name -like 'C:') {
        # Calcul de la taille en Giga octet
        $taille = $disque.freespace / 1GB
        $taille = [math]::round($taille, 1) #Arrondi la taille à 1 décimale
        write-host "Le disque $($disque.Name) a $taille Go de disponibles"
    }
    $taille_totale = $taille_totale + $taille
}
Write-Host "Taille disponible cumulée = $taille_totale Go"
```

Voyons le résultat dans le premier mode de traces :

```
PS > Set-PSDebug -Trace 1
PS > ./FreeSpace.ps1
DÉBOGUER : 1+ ./FreeSpace.ps1
DÉBOGUER : 5+ $elements = Get-WmiObject Win32_LogicalDisk
DÉBOGUER : 7+ $taille_totale = 0 # initialisation de la variable
DÉBOGUER : 10+ foreach ( $disque in $elements ) {
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+         $taille_totale = $taille_totale + $taille
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 13+         $taille = $disque.freespace / 1GB
DÉBOGUER : 14+         $taille = [math]::round($taille, 1)
                        #Arrondi la taille à 1 décimale
DÉBOGUER : 15+         write-host "Le disque $($disque.Name)
                        a $taille Go de disponibles"
DÉBOGUER : 1+ $disque.Name
Le disque C: a 73.3 Go de disponibles
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER : 19+ write-host "Taille disponible cumulée =$taille_totale Go"
Taille disponible cumulée = 513.1 Go
```

Sur la console, nous constatons que tous les traitements sont affichés en jaune et en tant que message de débogage. De plus, un nombre suivi du signe « + » est affiché devant chaque traitement. Ce nombre correspond au numéro de ligne du script en cours d'exécution. On remarque également que plusieurs numéros de lignes réapparaissent comme le 11 et le 17. Cela est normal dans la mesure où notre script exécute une boucle grâce à l'instruction **foreach**.

Regardons maintenant ce qui se passe en définissant le niveau de suivi à 2 :

```
PS > Set-PSDebug -Trace 2
PS > ./FreeSpace.ps1
DÉBOGUER : 1+ ./FreeSpace.ps1
DÉBOGUER : ! CALL script 'FreeSpace.ps1'
DÉBOGUER : 5+ $elements = get-WmiObject Win32_LogicalDisk
DÉBOGUER : ! SET $elements ='\\PCVISTA\\root\\cimv2:
                Win32_LogicalDisk.DeviceID...'.
DÉBOGUER : 7+ $taille_totale = 0 # initialisation de la variable
DÉBOGUER : ! SET $taille_totale = '0'.
DÉBOGUER : 10+ foreach ( $disque in $elements ) {
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+         $taille_totale = $taille_totale + $taille
```

```

DÉBOGUER :      ! SET $taille_totale = '0'.
DÉBOGUER : 11+   if ($disque.Name -like 'C:') {
DÉBOGUER : 13+       $taille = $disque.freespace / 1GB
DÉBOGUER :      ! SET $taille = '73.3418655395508'.
DÉBOGUER : 14+       $taille = [math]::round($taille, 1)
                    #Arrondi la taille à 1 décimale
DÉBOGUER :      ! CALL method 'static System.Double Round(Double value,
                    Int32 digits)'
DÉBOGUER :      ! SET $taille = '73.3'.
DÉBOGUER : 15+       write-host "Le disque $($disque.Name)
                    a $taille Go de disponibles"
DÉBOGUER : 1+ $disque.Name
Le disque C: a 73.3 Go de disponibles
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER :      ! SET $taille_totale = '73.3'.
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER :      ! SET $taille_totale = '146.6'.
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER :      ! SET $taille_totale = '219.9'.
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER :      ! SET $taille_totale = '293.2'.
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER :      ! SET $taille_totale = '366.5'.
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER :      ! SET $taille_totale = '439.8'.
DÉBOGUER : 11+     if ($disque.Name -like 'C:') {
DÉBOGUER : 17+     $taille_totale = $taille_totale + $taille
DÉBOGUER :      ! SET $taille_totale = '513.1'.
DÉBOGUER : 19+ write-host "Taille disponible cumulée = $taille_totale Go"
Taille disponible cumulée = 513.1 Go

```

Dans ce mode nous voyons, en plus, apparaître l'appel de notre script, les différentes affectations de variables et leurs valeurs associées, ainsi que l'appel des méthodes statiques du Framework .NET.

9. Trace-Command

Cette commandelette permet d'obtenir des traces de très bas niveau. Elle a été initialement conçue pour (et par) les employés de Microsoft en charge du développement de PowerShell mais aussi pour ceux en charge de l'assistance aux utilisateurs. Son usage et son interprétation complexes la réservent davantage aux développeurs expérimentés qu'aux utilisateurs finaux de PowerShell qui, dans la version 1.0 se contenteront pour leur part de **Set-PSDebug**. Il existe très peu de documentation sur **Trace-Command**.

Pour la suite des opérations, il peut être utile de savoir que la mécanique de traces de cette commande est celle du Framework .NET.

Regardons quels sont les paramètres de **Trace-Command** :

Paramètre	Description
Name	Nom de la source de trace.
Expression	Bloc de script à tracer.
Option	Type des événements tracés, All est la valeur par défaut.
FilePath	Envoi de la trace dans un fichier.
Debugger	Envoi de la trace dans un débogueur.

PSHost	Envoi de la trace sur l'écran.
ListenerOption	Niveau de détail de chaque ligne de trace.

Les paramètres les plus courants sont les suivants :

- **-Name** : on indique ici le nom de la source de trace ; c'est-à-dire les informations qui nous intéressent de tracer. Par exemple nous pouvons uniquement nous intéresser aux conversions de type que réalise PowerShell lors d'une affectation de variable, ou bien encore à l'affectation des paramètres lors de l'appel d'un script ou d'une commande. Les sources de trace sont nombreuses : il y en a près de cent quatre-vingt ! Pour toutes les connaître, utilisez la commande : **Get-TraceSource**.
- **-Expression** : on spécifie dans ce paramètre un bloc de scripts entre accolades. Exemple : `{./monScript.ps1}`.
- **-PSHost** : affiche la trace sur l'écran.
- **-FilePath** : lorsque les informations sont nombreuses il est préférable de les rediriger dans un fichier. À noter que cette option peut être utilisée conjointement avec **-PSHost**.

Les sources de trace sont incroyablement nombreuses, pour en avoir la liste utilisez la commande **Get-TraceSource**. Vous trouverez la liste complète dans l'annexe Liste des sources de traces.

Voici une description de quelques sources de trace :

Source	Description
TypeConversion	Trace la mécanique interne de conversion de type. Par exemple, lors de l'affectation de variables.
CommandDiscovery	Permet d'observer comment l'interpréteur de commandes fait pour trouver une commande ou un script.
ParameterBinding	Trace l'association de paramètres entre l'appel d'un script ou d'une fonction et l'interpréteur de commandes.
FormatViewBinding	Permet de savoir si une vue prédéfinie existe ou non.

Exemple :

Source de trace *TypeConversion*.

Prenons un exemple simple où nous définissons une variable en forçant son type :

```
PS > [char]$var=65
```

Nous affectons à une variable de type **char** la valeur « 65 », afin d'obtenir son caractère ASCII correspondant, soit « A ».

Grâce à Trace-Command, nous allons mieux comprendre ce qui se passe dans les entrailles de notre interpréteur de commandes préféré.

Essayons la ligne de commandes suivante :

```
PS > Trace-Command -Name TypeConversion -Expression {[char]$var=65} -Pshost
```

Voici le résultat obtenu :

```
PS > Trace-Command -Name TypeConversion -Expression {[char]$var=65} -Pshost
```

```
DÉBOGUER : TypeConversion Information: 0 : Converting "char" to
           "System.Type".
```

```
DÉBOGUER : TypeConversion Information: 0 : Original type before getting
```



```

DÉBOGUER : TypeConversion Information: 0 : BaseObject: "System.String".
Original type after getting
BaseObject: "System.String".
DÉBOGUER : TypeConversion Information: 0 : Standard type conversion.
DÉBOGUER : TypeConversion Information: 0 : Converting integer
to System.Enum.
DÉBOGUER : TypeConversion Information: 0 : Type conversion from string.
DÉBOGUER : TypeConversion Information: 0 : Conversion
to System.Type
DÉBOGUER : TypeConversion Information: 0 : The conversion is a
standard conversion. No custom type conversion will
be attempted.
DÉBOGUER : TypeConversion Information: 0 : Converting "65" to
"System.Char".
DÉBOGUER : TypeConversion Information: 0 : Original type before
getting BaseObject: "System.Int32".
DÉBOGUER : TypeConversion Information: 0 : Original type after
getting BaseObject: "System.Int32".
DÉBOGUER : TypeConversion Information: 0 : Standard type conversion.
DÉBOGUER : TypeConversion Information: 0 : Converting integer to
System.Enum.
DÉBOGUER : TypeConversion Information: 0 : Type conversion from
string.
DÉBOGUER : TypeConversion Information: 0 : Custom type conversion.
DÉBOGUER : TypeConversion Information: 0 : Parse type conversion.
DÉBOGUER : TypeConversion Information: 0 : Constructor type
conversion.
DÉBOGUER : TypeConversion Information: 0 : Cast operators type
conversion.
DÉBOGUER : TypeConversion Information: 0 : Looking for "op_Implicit"
cast operator.
DÉBOGUER : TypeConversion Information: 0 : Cast operator for
"op_Implicit" not found.
DÉBOGUER : TypeConversion Information: 0 : Looking for
"op_Explicit" cast operator.
DÉBOGUER : TypeConversion Information: 0 : Cast operator
for "op_Explicit" not found.
DÉBOGUER : TypeConversion Information: 0 : Could not find cast operator.
DÉBOGUER : TypeConversion Information: 0 : Cast operators
type conversion.
DÉBOGUER : TypeConversion Information: 0 : Looking
for "op_Implicit" cast operator.
DÉBOGUER : TypeConversion Information: 0 : Cast operator
for "op_Implicit" not found.
DÉBOGUER : TypeConversion Information: 0 : Looking
for "op_Explicit" cast operator.
DÉBOGUER : TypeConversion Information: 0 : Cast operator
for "op_Explicit" not found.
DÉBOGUER : TypeConversion Information: 0 : Could not find cast operator.
DÉBOGUER : TypeConversion Information: 0 : Conversion
using IConvertible succeeded.
DÉBOGUER : TypeConversion Information: 0 : Converting
"A" to "System.Char".
DÉBOGUER : TypeConversion Information: 0 : Result
type is assignable from value to convert's type

```



Le résultat obtenu peut différer selon que vous utilisez PowerShell 1.0 ou 2.0. Ici, peu importe la version, l'essentiel est de vous montrer la fonctionnalité de la commandelette trace-debug.

Exemple :

Source de trace CommandDiscovery.

Dans cet exemple, nous allons tenter d'exécuter un script qui n'existe pas et observer le comportement de l'interpréteur de commandes.

Essayons la ligne de commandes suivante :

```
PS > Trace-Command -Name CommandDiscovery -Expression {c:\monScript.ps1}
-Pshost
```

Voici le résultat obtenu :

```
PS > Trace-Command -Name CommandDiscovery -Expression
{c:\monScript.ps1} -Pshost
DÉBOGUER : CommandDiscovery Information: 0 : Looking up command:
c:\monScript.ps1
DÉBOGUER : CommandDiscovery Information: 0 : Attempting to resolve
function or filter: c:\monScript.ps1
DÉBOGUER : CommandDiscovery Information: 0 : The name appears to be
a qualified path: c:\monScript.ps1
DÉBOGUER : CommandDiscovery Information: 0 : Trying to resolve the path
as an PSPATH
DÉBOGUER : CommandDiscovery Information: 0 : ERROR: The path could
not be found: c:\monScript.ps1
DÉBOGUER : CommandDiscovery Information: 0 : The path is rooted,
so only doing the lookup in the specified directory:
c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for monScript.ps1
in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.ps1 in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.COM in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.EXE in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.BAT in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.CMD in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.VBS in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.VBE in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.JS in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.JSE in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.WSF in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.WSH in c:\
DÉBOGUER : CommandDiscovery Information: 0 : Looking for
monScript.ps1.MSC in c:\
DÉBOGUER : CommandDiscovery Information: 0 : The command
[c:\monScript.ps1] was not found, trying again with get-prepended
DÉBOGUER : CommandDiscovery Information: 0 : Looking up command:
get-c:\monScript.ps1
DÉBOGUER : CommandDiscovery Information: 0 : Attempting to resolve
function or filter: get-c:\monScript.ps1
DÉBOGUER : CommandDiscovery Information: 0 : The name appears
to be a qualified path: get-c:\monScript.ps1
DÉBOGUER : CommandDiscovery Information: 0 : Trying to resolve
the path as an PSPATH
DÉBOGUER : CommandDiscovery Information: 0 : ERROR: A drive could
not be found for the path: get-c:\monScript.ps1
DÉBOGUER : CommandDiscovery Information: 0 : ERROR: The drive does
not exist: get-c
DÉBOGUER : CommandDiscovery Information: 0 : The path is relative,
```

so only doing the lookup in the specified directory:

```
DÉBOGUER : CommandDiscovery Information: 0 : ERROR: 'get-c:\monScript.ps1' is not recognized as a cmdlet, function, operable program or script file.
```

Le terme « c:\monScript.ps1 » n'est pas reconnu en tant qu'applet de commande, fonction, programme exécutable ou fichier de script. Vérifiez le terme et réessayez.

Au niveau de ligne : 1 Caractère : 67

```
+ Trace-Command -name CommandDiscovery -expression {c:\monScript.ps1} <<<< -PShost
```

Nous constatons que PowerShell commence d'abord à rechercher une fonction ou un filtre portant le nom indiqué (« c:\monScript.ps1 »). Puis comme il n'en trouve pas, il détermine qu'il s'agit d'un chemin vers un fichier. Il cherche alors le fichier « monScript.ps1 » dans le répertoire « c:\ ». Ce fichier étant introuvable, il passe alors en revue toutes les extensions contenues dans la variable d'environnement **PATHEXT** afin d'essayer de trouver un fichier à exécuter. Pour finir, comme la recherche a été pour l'instant infructueuse, l'interpréteur recherche une commandelette de type « get » en ajoutant le préfixe « get- » à « c:\monScript.ps1 », soit « **get-c:\monScript.ps1** ». Enfin, lorsque toutes les solutions sont épuisées PowerShell génère une erreur.

Intéressant n'est-ce pas ? Difficile d'imaginer tout ce qui se passe derrière une simple opération d'exécution de script.

Exemple :

Source de trace FormatViewBinding.

Cette source de trace va nous permettre de savoir si le résultat d'une commande affichée à l'écran a bénéficié d'un formatage de la part de PowerShell. En effet, nous avons pu constater dans le chapitre précédent qu'un grand nombre de types d'objets bénéficient d'un formatage par défaut, qui est décrit dans les fichiers **.ps1xml** contenus dans le répertoire d'installation de PowerShell (dans la variable **\$PSHome**, soit généralement **C:\Windows\System32\WindowsPowerShell\v1.0**).

Essayons la ligne de commandes suivante :

```
PS > Trace-Command -Name FormatViewBinding -Expression {Get-Process notepad | Out-Host} -PShost
```

Voici le résultat obtenu :

```
PS > Notepad.exe
PS > Trace-Command -Name FormatViewBinding -Expression {Get-Process notepad | Out-Host} -PShost

DÉBOGUER : FormatViewBindin Information: 0 : FINDING VIEW TYPE:
System.Diagnostics.Process
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
ThumbprintTable TYPE:
System.Security.Cryptography.X509Certificates.X509Certificate2
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH List NAME:
ThumbprintList GROUP: CertificateProviderTypes
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Wide NAME:
ThumbprintWide GROUP: CertificateProviderTypes
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
PSTThumbprintTable TYPE: System.Management.Automation.Signature
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Wide NAME:
PSTThumbprintWide TYPE: System.Management.Automation.Signature
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH List NAME:
PathOnly GROUP: CertificateProviderTypes
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Security.Cryptography.X509Certificates.X509CertificateEx TYPE:
System.Security.Cryptography.X509Certificates.X509CertificateEx
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Reflection.Assembly TYPE: System.Reflection.Assembly
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Reflection.AssemblyName TYPE: System.Reflection.AssemblyName
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Globalization.CultureInfo TYPE: System.Globalization.CultureInfo
```

```

DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME: System.
Diagnostics.FileVersion Info TYPE: System.Diagnostics.FileVersionInfo
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Diagnostics.EventLogEntry TYPE: System.Diagnostics.EventLogEntry
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Diagnostics.EventLog TYPE: System.Diagnostics.EventLog
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Version TYPE: System.Version
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Drawing.Printing.PrintDocument TYPE: System.Drawing.Printing.
PrintDocument
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
Dictionary TYPE: System.Collections.DictionaryEntry
DÉBOGUER : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
ProcessModule TYPE: System.Diagnostics.ProcessModule
DÉBOGUER : FormatViewBindin Information: 0 : MATCH FOUND Table NAME:
process TYPE: System.Diagnostics.Process
DÉBOGUER : FormatViewBindin Information: 0 : MATCH FOUND Table NAME:
process TYPE: Deserialized.System.Diagnostics.Process
DÉBOGUER : FormatViewBindin Information: 0 : An applicable
view has been found

```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
53	3	1444	5564	53	0,41	2888	notepad
53	2	1484	5436	53	1,17	5476	notepad

Nous pouvons voir sur les toutes dernières lignes les informations suivantes « **DÉBOGUER : FormatViewBindin Information: 0 : An applicable view has been found** ». Cela signifie qu'une vue a été trouvée.

Quant à la première ligne « **DÉBOGUER : FormatViewBindin Information: 0 : FINDING VIEW TYPE: System.Diagnostics.Process** », celle-ci est intéressante car nous indique précisément le nom du type de la vue.

Si aucune vue n'avait été trouvée pour ce type, nous aurions eu le message suivant sur la dernière ligne : « **DÉBOGUER : FormatViewBindin Information: 0 : No applicable view has been found** ».

Pré-requis d'exécution de script

Au rayon des nouveautés de PowerShell v2, on peut également compter sur les prérequis d'exécution. En effet, à partir de la version 2 de Powershell il est désormais possible d'empêcher l'exécution d'un script si des conditions ne sont pas remplies. Ces conditions peuvent concerner les domaines suivants :

Type de filtres	Syntaxe
La version de PowerShell utilisée, v2 ou ultérieure.	#requires -Version <numéro de version>
La version d'un snap-in (composant enfichable).	#requires -PsSnapIn <nom du snapin> [-Version <numéro de version>]
L'interpréteur de commande.	#requires -ShellId <Id du shell>

Pour mettre en place ces pré-requis, rien de plus simple, il suffit de placer en tête du script, ou bien d'une ligne, le symbole dièse puis le mot clé « requires » suivi des paramètres souhaités, voir tableau ci-dessus. Prenons le cas très simple d'un script qui doit utiliser le snap-in VMware livré avec vSphere PowerCLI permettant d'administrer une ferme de serveurs VMware ESX à partir de PowerShell.

```
#requires -PsSnapIn vmware.VimAutomation.Core
```

Si le snap-in n'est pas correctement chargé, alors le message suivant apparaît :

```
PS > .\MonScript.ps1
Le script « MonScript.ps1 » ne peut pas être exécuté, car les composants
logiciels enfichables suivants, spécifiés par les instructions
« #requires » du script, sont absents : vmware.VimAutomation.Core.
Au niveau de ligne : 1 Caractère : 16
+ .\MonScript.ps1 <<<<
+ CategoryInfo          : ResourceUnavailable: (MonScript.ps1:String) [],
ScriptRequiresException
+ FullyQualifiedErrorId : ScriptRequiresMissingPSSnapIns
```

A contrario, c'est-à-dire que si le snap-in est chargé avant l'exécution du script. Alors ce dernier ne rencontrera pas un bridage à l'exécution.

```
PS > Add-PSSnapin vmware.VimAutomation.Core
PS > .\MonScript.ps1
Début du script...
```

Deuxième exemple, celui de la version de l'interpréteur de commande. Prenons le cas très simple d'un script qui doit utiliser le shell PowerShell et non pas celui d'échange 2007 (Exchange Shell). Le pré-requis est alors indiqué de la manière suivante :

```
#requires -ShellId Microsoft.PowerShell
```

Si le pré-requis n'est pas respecté à l'exécution du script alors un message d'erreur empêchant l'exécution se produira. Astuce : Le ShellID de la console PowerShell par défaut est « Microsoft.PowerShell ». D'une manière générale, le ShellID peut s'obtenir via l'affichage du contenu de la variable \$ShellID.

Concernant le cas de la version de PowerShell, les plus attentifs d'entre vous auront remarqué que pour le moment cette restriction n'a que peu d'intérêt. En effet, les restrictions commencent par le symbole dièse, ce qui signifie que la version 1.0 qui interprète le dièse comme une ligne de commentaire, n'est pas en mesure de déterminer qu'il s'agit d'un filtre sur la version de PowerShell. Cette fonctionnalité de filtrage sur la version sera par contre très utile pour les futures versions.



L'utilisation de l'instruction Requires peut être cumulée plusieurs fois dans un même script, cependant elle n'est utilisable que dans les scripts.

La sécurité : pour qui ? Pourquoi ?

L'arrivée des réseaux locaux et d'Internet a changé beaucoup de choses dans la manière de protéger son PC. Il ne suffit plus d'attacher son disque dur au radiateur et de fermer la porte du bureau le soir pour ne pas se faire voler ou pirater des données. Maintenant, protéger son poste de travail est devenu essentiel pour ne pas faire les frais d'intrusions ou de malversations.

Mais alors contre qui se prémunir ? Hé bien, contre tout ce qui bouge... et même ce qui ne bouge pas. En effet, que ce soit des programmes malveillants, des utilisateurs mal intentionnés, voire des utilisateurs inexpérimentés, tous peuvent être considérés comme une menace. C'est pour cela que vous devez verrouiller votre système en établissant des règles de sécurité, en les appliquant et vous assurant que les autres en font autant.

Les risques liés au scripting

Vous allez vite deviner que ce qui fait la force du scripting, en fait aussi sa faiblesse. La facilité avec laquelle vous pouvez tout faire, soit en cliquant sur un script, soit en l'exécutant depuis la fenêtre de commande, peut vous mettre dans l'embarras si vous ne faites pas attention.

Imaginez un script de logon qui dès l'ouverture de la session la verrouille aussitôt ! Alors, oui c'est sympa entre copains, mais en entreprise, nous doutons que cela soit de bon ton. Plus grave encore, un script provenant d'une personne mal intentionnée ou vraiment peu expérimentée en PowerShell (dans ce cas, nous vous conseillons de lui acheter un exemplaire de ce livre...) peut parfaitement vous bloquer des comptes utilisateurs dans Active Directory, vous formater un disque, vous faire rebooter sans cesse. Enfin, vous l'avez compris, un script peut tout faire. Car même si aujourd'hui des alertes sont remontées jusqu'à l'utilisateur pour le prévenir de l'exécution d'un script, elles ne sont pas capables de déterminer à l'avance si un script est nuisible au bon fonctionnement du système.

Les risques liés au scripting se résument à une histoire de compromis, soit vous empêchez toute exécution de script, c'est-à-dire encourir le risque de vous pourrir la vie à faire et à refaire des tâches basiques et souvent ingrates. Soit vous choisissez d'ouvrir votre système au scripting, en prenant soin de prendre les précautions qui s'imposent.

Mais ne vous laissez pas démoraliser car même si l'exécution de scripts vous expose à certains problèmes de sécurité, PowerShell se dote de nouveaux concepts qui facilitent grandement cet aspect du scripting.

Optimiser la sécurité PowerShell

1. La sécurité PowerShell par défaut

Vous l'avez compris, la sécurité est une chose très importante, surtout dans le domaine du scripting. C'est pour cela que les créateurs de PowerShell ont inclus deux règles de sécurités par défaut.

Des fichiers ps1 associés au bloc-notes

L'extension « .ps1 » des scripts PowerShell, est par défaut associée à l'éditeur de texte bloc-notes (ou Notepad). Ce procédé permet d'éviter de lancer des scripts potentiellement dangereux sur une mauvaise manipulation. Le bloc-notes est certes un éditeur un peu classique, mais a le double avantage d'être inoffensif et de ne pas bloquer l'exécution d'un script lorsque celui-ci est ouvert avec l'éditeur.

➤ Ce type de sécurité n'était pas mis en place avec les scripts VBS dont l'ouverture était directement associée au Windows Script Host. Que ceux n'ayant jamais double cliqué sur un script VBS en voulant l'éditer nous jettent la pierre !

Une stratégie d'exécution restreinte

La seconde barrière de sécurité est l'application de stratégie d'exécution « restricted » par défaut (cf. stratégies d'exécution).

Cette stratégie est la plus restrictive. C'est-à-dire qu'elle bloque systématiquement l'exécution de tout script. Seules les commandes tapées dans le shell seront exécutées. Pour remédier à l'inexécution de script, PowerShell requiert que l'utilisateur change le mode d'exécution avec la commande **Set-ExecutionPolicy <mode d'exécution>**.

➤ Peut-être comprenez-vous mieux pourquoi le déploiement de PowerShell sur vos machines ne constitue pas un accroissement des risques, dans la mesure où certaines règles sont bien respectées.

2. Les stratégies d'exécution

PowerShell intègre un concept de sécurité que l'on appelle les stratégies d'exécution (execution policies) pour qu'un script non autorisé ne puisse pas s'exécuter à l'insu de l'utilisateur. Il existe quatre configurations possibles : **Restricted**, **RemoteSigned**, **AllSigned** et **unrestricted**. Chacune d'elles correspond à un niveau d'autorisation d'exécution de scripts particulier, et vous pourrez être amenés à en changer en fonction de la stratégie que vous souhaitez appliquer.

a. Les différentes stratégies d'exécution

Restricted : c'est la stratégie la plus restrictive, et c'est aussi la stratégie par défaut. Elle ne permet pas l'exécution de script mais autorise uniquement les instructions en ligne de commande, c'est-à-dire uniquement dans le shell. Cette stratégie peut être considérée comme la plus radicale étant donné qu'elle protège l'exécution involontaire de fichiers « .ps1 ».

Lors d'une tentative d'exécution de script avec cette stratégie, un message de ce type est affiché dans la console :

```
Impossible de charger le fichier C:\script.ps1,  
car l'exécution de scripts est désactivée sur ce système.
```

Comme cette stratégie est celle définie par défaut lors de l'installation de PowerShell, il vous faudra donc la changer pour l'exécution de votre premier script.

AllSigned : c'est la stratégie permettant l'exécution de script la plus « sûre ». Elle autorise uniquement l'exécution des scripts signés. Un script signé est un script comportant une signature numérique comme celle présentée sur la figure suivante.

Exemple de script signé

Avec la stratégie **AllSigned**, l'exécution de scripts signés nécessite que vous soyez en possession des certificats correspondants (cf. partie sur la signature des scripts).

RemoteSigned : cette stratégie se rapporte à **AllSigned** à la différence près que seuls les scripts ayant une origine autre que locale nécessitent une signature. Par conséquent, cela signifie que tous vos scripts créés localement peuvent être exécutés sans être signés.

Si vous essayez d'exécuter un script provenant d'Internet sans que celui-ci soit signé, le message suivant sera affiché dans la console.

```
Impossible de charger le fichier C:\script.ps1.
Le fichier C:\script.ps1 n'est pas signé numériquement.
Le script ne sera pas exécuté sur le système.
```

 Vous vous demandez sûrement comment PowerShell fait pour savoir que notre script provient d'Internet ? Réponse : Grâce aux « Alternate Data Streams » qui sont implémentés sous forme de flux cachés depuis des applications de communication telles que Microsoft Outlook, Internet Explorer, Outlook Express et Windows Messenger (voir partie traitant des Alternate Data Streams).

Unrestricted : c'est la stratégie la moins contraignante, et par conséquent la moins sécurisée. Avec elle, tout script, peu importe son origine, peut être exécuté sans demande de signature. C'est donc la stratégie où le risque d'exécuter des scripts malveillants est le plus élevée.

Cette stratégie affiche tout de même un avertissement lorsqu'un script téléchargé d'Internet tente d'être exécuté.

```
PS > .\script.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:\script.ps1 ?
[N] Ne pas exécuter [O] Exécuter une fois
[S] Suspendre [?] Aide (la valeur par défaut est « N ») :
```


PowerShell v2 apporte deux stratégies d'exécution supplémentaires :

Bypass : rien n'est bloqué et aucun message d'avertissement ne s'affiche.

Undefined : pas de stratégie d'exécution définie dans l'étendue courante. Si toutes les stratégies d'exécution de toutes les étendues sont non définies alors la stratégie effective appliquée sera la stratégie **Restricted**.

b. Les étendues des stratégies d'exécution

Cette partie ne s'applique qu'à PowerShell v2.

PowerShell v2 apporte un niveau de granularité que PowerShell v1 n'avait pas dans la gestion des stratégies d'exécution. PowerShell v1 ne gère la stratégie d'exécution qu'au niveau de l'ordinateur autrement dit, PowerShell v1 n'était doté que de l'étendue LocalMachine.

En plus de l'étendue LocalMachine, PowerShell v2 apporte les deux nouvelles étendues suivantes : Process, et CurrentUser. Il est possible d'affecter une stratégie d'exécution à chaque étendue si on le désire.

La priorité d'application des stratégies est la suivante :

- **Etendue Process** : la stratégie d'exécution n'affecte que la session courante (processus Windows PowerShell). La valeur affectée à l'étendue Process est stockée en mémoire uniquement ; elle n'est donc pas conservée lors de la fermeture de la session PowerShell.
- **Etendue CurrentUser** : la stratégie d'exécution appliquée à l'étendue CurrentUser n'affecte que l'utilisateur courant. Le type de stratégie est stocké de façon permanente dans la partie du registre HKEY_CURRENT_USER.
- **Etendue LocalMachine** : la stratégie d'exécution appliquée à l'étendue LocalMachine affecte tous les utilisateurs de la machine. Le type de stratégie est stocké de façon permanente dans la partie du registre HKEY_LOCAL_MACHINE.

La stratégie ayant une priorité 1 est plus propriétaire que celle ayant une priorité 3. Par conséquent, si l'étendue LocalMachine est plus restrictive que l'étendue Process, la stratégie qui s'appliquera sera quand même la stratégie de l'étendue Process. À moins que cette dernière soit de type Undefined auquel cas PowerShell appliquera la stratégie de l'étendue CurrentUser, puis tentera d'appliquer la stratégie LocalMachine.

À noter que l'étendue LocalMachine est celle par défaut lorsque l'on applique une stratégie d'exécution sans préciser d'étendue particulière.

c. Identifier la stratégie d'exécution courante

La stratégie d'exécution courante s'obtient avec la commande **Get-ExecutionPolicy**.

Exemple :

```
PS > Get-ExecutionPolicy
Restricted
```

Avec PowerShell v1, ne se pose pas la question de savoir quelle est l'étendue concernée par le mode retourné par cette commande. En effet, PowerShell v1 ne gère que l'étendue LocalMachine.

Avec PowerShell v2, nous bénéficions du switch -List. Grâce à lui nous allons savoir quelles stratégies s'appliquent à nos étendues.

Par exemple :

```
PS > Get-ExecutionPolicy -List

Scope ExecutionPolicy
-----
MachinePolicy Undefined
UserPolicy Undefined
Process Undefined
CurrentUser AllSigned
LocalMachine Restricted
```

Dans cet exemple nous voyons qu'à l'étendue CurrentUser nous avons appliqué la stratégie AllSigned, tandis qu'à l'étendue LocalMachine est affectée la stratégie Restricted (valeur par défaut). Si vous avez bien suivi jusque là, quelle est d'après vous la stratégie qui s'applique à notre session PowerShell courante ?

Pour le savoir demandons-le à **Get-ExecutionPolicy** :

```
PS > Get-ExecutionPolicy
AllSigned
```

Et oui, il s'agit de la stratégie AllSigned car l'étendue CurrentUser a la priorité par rapport à l'étendue LocalMachine.

Notez que nous avons dans la liste des étendues retournées les étendues MachinePolicy et UserPolicy. Ces étendues correspondent respectivement aux étendues LocalMachine et CurrentUser lorsque les stratégies de groupes (GPO) sont utilisées pour paramétrer le comportement de PowerShell sur les postes des utilisateurs d'un domaine.

L'ordre d'application des stratégies d'exécution est celui retourné par la commande **Get-ExecutionPolicy -List**. Il faut savoir que les stratégies d'exécution définies par GPO sont prioritaires par rapport aux autres.

d. Appliquer une stratégie d'exécution

La stratégie **Restricted** est la stratégie appliquée par défaut à l'environnement PowerShell. Celle-ci n'est pas adaptée, puisqu'elle ne permet pas l'exécution de scripts. Nous ne pouvons donc que vous conseiller d'en choisir une plus souple, de façon à pouvoir jouir des nombreux avantages qu'offre le scripting PowerShell.

Le changement de stratégie d'exécution se fait par la commande **Set-ExecutionPolicy** suivi du mode choisi.

Par exemple : Set-ExecutionPolicy RemoteSigned aura pour effet d'appliquer la stratégie d'exécution **RemoteSigned** à l'étendue **LocalMachine**.

Avec PowerShell v2, pour appliquer une stratégie à une autre étendue que l'étendue LocalMachine, il faut utiliser le paramètre -Scope suivi du nom de l'étendue.

Exemple : application de la stratégie RemoteSigned à l'étendue Process

```
PS > Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process
```

Modification de la stratégie d'exécution

La stratégie d'exécution permet de vous prémunir contre les scripts que vous jugez non fiables. En modifiant la stratégie d'exécution, vous vous exposez aux risques de sécurité décrits dans la rubrique d'aide about_Execution_Policies. Voulez-vous modifier la stratégie d'exécution ?
[0] Oui [N] Non [S] Suspendre [?] Aide (la valeur par défaut est « 0 ») :

Le changement de stratégie d'exécution s'accompagne systématiquement d'un message d'avertissement vous demandant de confirmer l'action.

Vérifions à présent ce que cela donne :

```
PS > Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
MachinePolicy	Undefined
UserPolicy	Undefined
Process	RemoteSigned
CurrentUser	AllSigned
LocalMachine	Restricted

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



Il n'est possible de modifier la stratégie d'exécution s'appliquant à l'étendue LocalMachine qu'avec les droits administrateur. Par conséquent, choisissez de démarrer PowerShell en tant qu'administrateur (clic droit **exécuter en tant que**).



La clé de registre correspondant à l'étendue LocalMachine est la suivante :
HKEY_Local_Machine\SOFTWARE\Microsoft\PowerShell\ 1\ShellIds\Microsoft.PowerShell\ExecutionPolicy.
Un autre moyen de configurer la stratégie d'exécution des machines, consiste à utiliser des GPO (*Group Policy Objects*). Pour cela, Microsoft livre un fichier .adm (voir plus loin dans ce chapitre).

3. Les scripts provenant d'Internet

Si vous avez lu la partie précédente vous savez donc que les scripts créés localement ne sont pas soumis aux mêmes obligations que ceux provenant d'Internet.

Tout d'abord, avec la stratégie **RemoteSigned**, il ne vous sera pas possible d'exécuter des scripts téléchargés sur Internet s'ils ne sont pas signés ou débloqués. Ceci étant, même signés, pour s'exécuter sans encombre, il faut que les scripts proviennent d'une entité approuvée.

Essayons d'y voir un peu plus clair ! Par exemple, lorsque vous téléchargez un script depuis Internet en passant par les outils de communication Microsoft (Outlook, Internet Explorer, Outlook Express et Windows Live Messenger), ces derniers associent à votre script un flux de données additionnelles appelé Alternate Data Stream (voir partie traitant des Alternate Data Streams) permettant à PowerShell d'identifier la provenance du script. Et si le script n'a pas une provenance locale, alors deux cas peuvent se présenter :

Cas d'un script signé

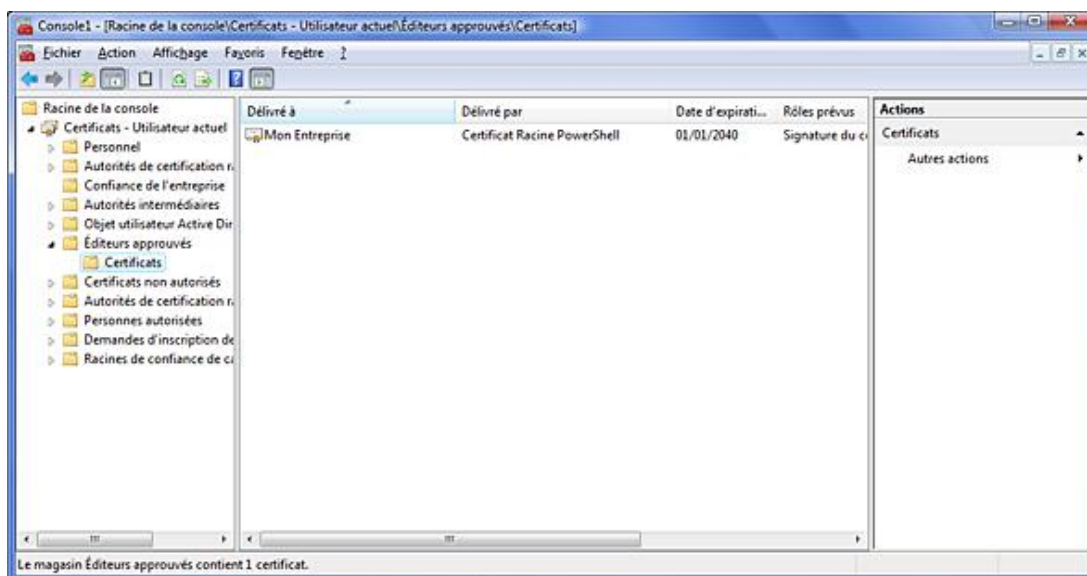
Si le script est signé numériquement, c'est-à-dire s'il contient une signature permettant à la fois d'identifier l'éditeur et de garantir l'intégrité du script, alors PowerShell vous demandera si vous souhaitez approuver cet éditeur.

Le choix d'approuver ou non un éditeur peut se faire lors de l'exécution du script. À ce moment précis, PowerShell vous demande si vous souhaitez exécuter le logiciel d'un éditeur non approuvé, et vous laisse le choix de répondre par :

```
[m] Ne jamais exécuter
[N] Ne pas exécuter
[o] Exécuter une fois
[T] Toujours exécuter
[?] Aide
```

Notez que si vous choisissez l'option **Ne jamais exécuter [m]** ou **Toujours exécuter [T]**, cette remarque concernant l'éditeur ne vous sera plus jamais posée.

En choisissant **Toujours exécuter**, vous faites confiance à l'éditeur, et de ce fait, comme vous pouvez le voir dans la console de gestion (cf. figure ci-après), le certificat correspondant à l'éditeur est importé dans le magasin de certificats « éditeurs approuvés ».



Éditeurs approuvés affichés dans la console de management

Notez que pour exécuter un script signé, il faut impérativement que vous soyez en possession d'un certificat d'une autorité racine de confiance correspondante. C'est ce que nous allons voir ultérieurement dans ce chapitre (cf. partie sur les certificats).



Le premier réflexe à avoir lorsque l'on récupère un script depuis Internet ou d'ailleurs, est de vérifier son contenu. Un œil critique décèlera vite les risques potentiels, si le code n'est pas trop long. Et si finalement après avoir inspecté le code vous avez déterminé qu'il ne présente aucun danger pour votre système, alors vous pouvez l'exécuter.

Cas d'un script non signé

Si vous essayez d'exécuter un script non signé provenant d'un ordinateur distant et que vous êtes en mode **RemoteSigned**, voici ce que PowerShell va vous répondre.

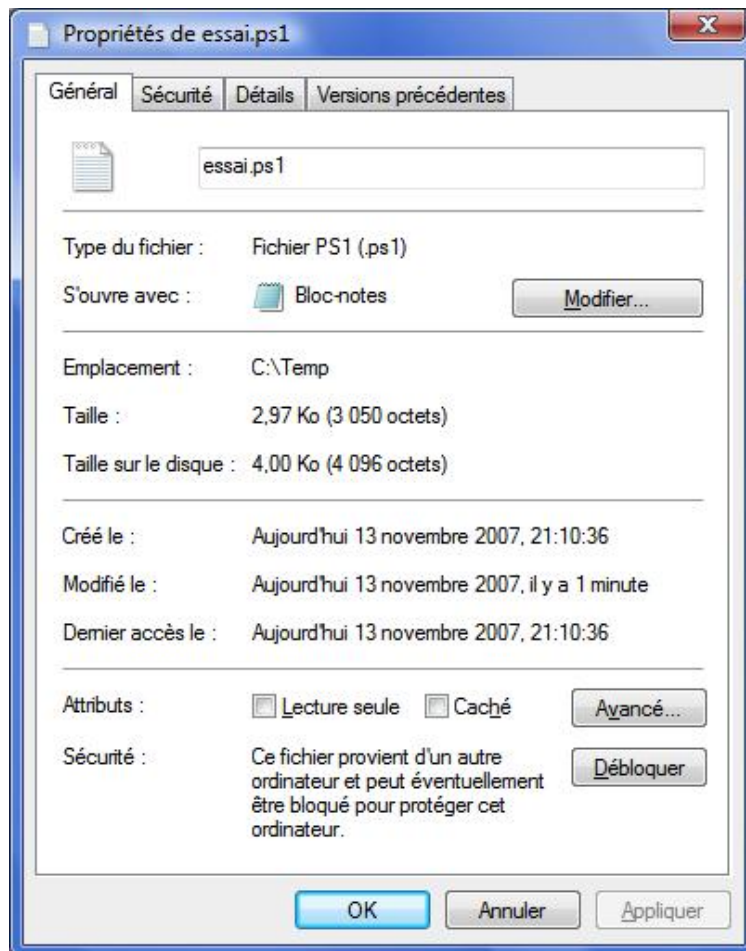
Impossible de charger le fichier C:\Temp\essai.ps1. Le fichier C:\Temp\essai.ps1 n'est pas signé numériquement. Le script ne sera pas exécuté sur le système.

Cependant, il est tout de même possible d'exécuter des scripts non signés. Pour cela il faut faire ce que l'on appelle « débloquer » le script. Débloquer un script correspond à supprimer l'Alternate Data Stream contenant l'information sur sa provenance. Une fois le script débloqué, vous pouvez l'exécuter comme s'il avait été fait localement.

De toute évidence, il vous faut plus que jamais inspecter le script. N'oubliez pas qu'un script téléchargé d'Internet sur un site dans lequel vous n'avez pas confiance est potentiellement dangereux.

Voici les étapes à suivre pour débloquer un script non signé téléchargé d'Internet.

- Faire un clic droit sur le script en question et choisir **Propriétés**. Puis, sur l'onglet **Général**, dans la partie inférieure de la fenêtre, choisissez **Débloquer**.



Fenêtre Propriétés permettant de débloquer un script

Le script est maintenant débloqué, il n'est désormais plus possible d'en connaître sa provenance.

4. Les Alternate Data Streams (ADS)

a. Les origines

Méconnus de nombreux informaticiens, les Alternate Data Streams (en français : flux de données additionnelles) ne datent pas d'hier. En effet, les ADS ont vu le jour avec le système de fichiers NTFS (*New Technology File System*) utilisé par la famille Windows dès le début des années 90 avec l'apparition de Windows NT 3.1.

Le principe des ADS qui n'a pas évolué depuis sa création, est d'insérer des flux de données supplémentaires dans un fichier. Jusqu'à là rien de surprenant me direz-vous ! Oui, mais chose plus surprenante, c'est que le contenu ainsi que la taille des flux sont invisibles. C'est-à-dire que vous pouvez « cacher » un exécutable de plusieurs mégaoctets dans un fichier texte de quelques octets sans que la taille du fichier, visible par l'utilisateur depuis l'onglet **Propriétés**, n'indique la présence des octets occupés par l'exécutable. Peu documentés et pas clairement expliqués, vous comprenez sûrement pourquoi les ADS sont encore aujourd'hui l'instrument de nombreux virus. La question est donc de savoir pourquoi ces flux de données sont rendus invisibles !

Ce qu'il faut savoir, c'est que l'utilisation des ADS aujourd'hui a bien dérivé de sa fonction principale. À l'origine les ADS ont été intégrés dans les systèmes d'exploitation Windows pour permettre une compatibilité avec le système de fichiers Macintosh : le Hierarchical File System (HFS). Car peut-être n'êtes-vous pas sans savoir que les fichiers Macintosh (sur les OS antérieurs à 'OS X') sont le résultat de l'association de deux composants : le *Data Fork* et le *Resource Fork*. Comme son nom l'indique, le Resource Fork contient les ressources utilisées par une application. On va trouver par exemple, des éléments de l'interface graphique (menus, fenêtre, messages, etc.) et d'autres éléments liés à la traduction de l'application en diverses langues. Et par opposition, le Data Fork contient le code binaire de l'application qui lui est a priori immuable. Cependant, dans le cas d'applications compatibles à la fois PowerPC et Motorola, le Data Fork contient les deux versions du code.

C'est donc pour une question d'interopérabilité entre systèmes que NTFS a intégré les data streams. Ces derniers jouant le rôle du « Resource Fork » version Windows. Cependant, dans la pratique, cette volonté d'interopérabilité n'a abouti à aucune application concrète. Aussi bien qu'aujourd'hui, les ADS servent principalement au système NTFS pour insérer des informations sur les fichiers (sortes de métadonnées). Par exemple, lorsque vous téléchargez un script PowerShell depuis Internet avec Internet Explorer, ce dernier va créer un Alternate Data Stream à votre script pour spécifier sa provenance. Et c'est ainsi que PowerShell va pouvoir déterminer si le script que vous souhaitez exécuter a été créé en local ou non.



Notez que seule la taille du flux principal des fichiers est prise en compte par les gestionnaires de quotas, celle des ADS est ignorée.

b. Créer et lire les ADS

Paradoxalement, PowerShell qui utilise lui-même les ADS pour connaître la provenance d'un script, ne les gère pas nativement. Et ce pour la simple et bonne raison que le Framework .NET ne les gère pas non plus. C'est pourquoi nous allons nous servir exceptionnellement de CMD.exe (mais promis, c'est l'unique fois) pour lire et créer des ADS.

Voici les instructions concernant la création d'un ADS contenant un autre fichier texte.

- Créons un fichier texte vide grâce à la commande suivante :

```
PS > New-Item -name MonFichier.txt -type file

Répertoire : C:\Temp

Mode                LastWriteTime         Length Name
----                -
-a---             10/10/2009      23:11             0 MonFichier.txt
```

Le résultat de la commande nous permet de constater que la taille de notre fichier est nulle.

- Créons ensuite un deuxième fichier texte qui cette fois ne sera pas vide mais contiendra l'aide de la commande **Get-Process**.

```
PS > Set-Content -path AideCommande.txt -value $(help get-process)
```

- Vérifions maintenant la taille de nos deux fichiers avec un simple Get-ChildItem.

```
PS > Get-ChildItem
```

Répertoire : C:\Temp

Mode	LastWriteTime	Length	Name
-a---	10/10/2009 23:11	1816	AideCommande.txt
-a---	10/10/2009 23:11	0	MonFichier.txt

La suite des opérations se passant sous CMD.exe, tapez simplement **cmd** dans la console PowerShell.

```
PS > cmd
Microsoft Windows [version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.
```

- Insérez maintenant le contenu du texte AideCommande.txt en tant qu'ADS du fichier vide MonFichier.txt avec la commande suivante :

```
C:\temp> type AideCommande.txt > MonFichier.txt:MonPremierAds.txt
```

Comme vous pouvez le constater, les ADS d'un fichier sont accessibles par l'intermédiaire du caractère « : » (<nom_fichier> :<nom_flux>) qui par conséquent est un caractère interdit lors de l'attribution du nom de fichier.

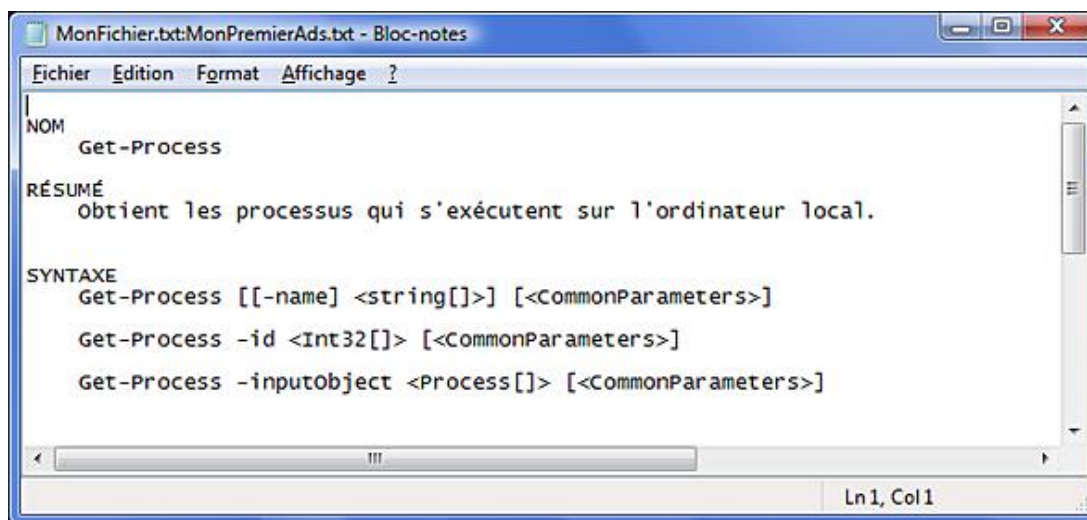
Si vous essayez de lire le fichier MonFichier.txt grâce à la commande **more** (toujours sous cmd.exe), en toute logique, rien ne s'affiche dans la console puisque le fichier est vide.

```
C:\temp> more MonFichier.txt
```

```
C:\temp>
```

Mais si vous essayez de lire l'ADS du nom de MonPremierAds.txt associé au fichier MonFichier.txt avec le bloc notes, utilisez la commande suivante :

```
C:\> notepad MonFichier.txt:MonPremierAds.txt
```



Extrait du contenu de l'ADS

Vous obtenez l'aide en ligne de la commande **Get-Process**. Et pourtant le fichier MonFichier.txt affiche toujours une taille nulle. Surprenant !

```
C:\temp> dir MonFichier.txt

Répertoire de C:\temp

10/10/2009 23:17                0 MonFichier.txt
               1 fichier(s)                0 octets
               0 Rép(s)   106 111 102 976 octets libres
```

Nous allons à présent insérer non plus du texte mais un exécutable en tant qu'ADS de notre fichier. Pour cela, tapez

la commande (toujours sous cmd.exe) :

```
C:\temp> type c:\WINDOWS\system32\calc.exe > MonFichier.txt:calc.exe
```

Vérifions de nouveau la taille de notre fichier !

```
C:\temp> dir MonFichier.txt

Répertoire de C:\temp

10/10/2009  23:17                0 MonFichier.txt
             1 fichier(s)                0 octets
             0 Rép(s)   106 111 102 976 octets libres
```

- Toujours nulle ! Enfin, pour exécuter l'ADS, tapez la commande :

```
C:\> start C:\temp\MonFichier.txt:calc.exe
```

Donc, en résumé sachez qu'il est possible de cacher énormément de choses avec les Alternate Data Streams, qu'il s'agisse d'images, d'exécutables, de vidéos, etc. bref tout flux de données peut être « hébergé » par un fichier « parent » sans que la taille de ce dernier ne change.



Les ADS sont propres à un seul ordinateur, c'est-à-dire que les ADS ne seront plus associés au fichier si vous le transférez (mail, clé USB, etc.).



Le lancement de fichiers exécutables dans un ADS ne fonctionne plus sous Windows 7 et Windows 2008 R2.

c. Observer et comprendre les ADS de vos fichiers .ps1

Si vous avez été attentifs, vous savez désormais que le mode d'exécution **RemoteSigned** reconnaît la provenance des scripts grâce aux ADS. Et nous allons voir exactement quels ADS sont créés et ce qu'ils contiennent. Mais la vie d'informaticien n'est pas un long fleuve tranquille, c'est donc pour cela qu'il n'est pas possible de lister les ADS nativement sous Windows. Nous allons donc nous accommoder d'un exécutable (streams.exe téléchargeable sur le site de Microsoft) permettant d'effectuer un listing des ADS associés à un fichier.

Utilisation de l'exécutable Streams

- Pour exécuter streams.exe, il vous suffit de lancer une première fois le fichier setup.exe, puis d'approuver la licence.



Affichage des termes de la licence relatif à l'installation de Streams.exe

Une fois la licence acceptée, il ne vous reste plus qu'à taper dans la console, le nom complet de l'exécutable streams.exe suivi du nom de fichier ou de répertoire.

L'exécutable streams.exe dispose de deux options :

- -s : liste récursivement tous les ADS associés au(x) fichier(s) d'un répertoire.
- -d : supprime les ADS associées au(x) fichier(s).

Bien évidemment, entrevoir les ADS associés à un script suppose que ce dernier soit en provenance d'un outil de communication Microsoft tel qu'Internet Explorer 7, Windows Live Messenger, Outlook, etc.

Prenons l'exemple du script *list-group.ps1* téléchargé depuis le site www.powershell-scripting.com. Puis, listons-on les ADS avec l'exécutable Streams.exe en utilisant la commande suivante :

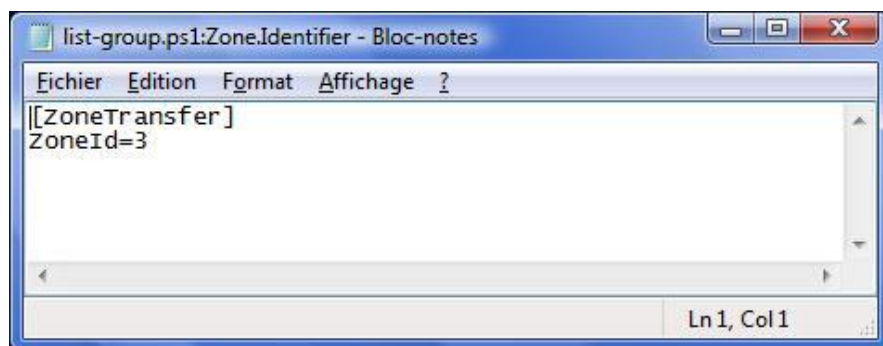
```
PS > ./streams.exe list-group.ps1

Streams v1.56 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2007 Mark Russinovich
Sysinternals - www.sysinternals.com

C:\Scripts\list-group.ps1:
:Zone.Identifier:$DATA          26
```

On observe clairement qu'un ADS du nom de Zone.Identifier a été détecté.

Et si l'on prend soin de voir ce qu'il contient, voici ce que l'on va trouver :



Affichage du contenu de l'ADS Zone.Identifier

En fait, lorsque vous téléchargez un script depuis un outil de communication Microsoft, ce dernier va créer un Data Stream pour y insérer des informations sur sa provenance. Cette information est traduite par un identifiant de zone (ZoneID) qui peut prendre plusieurs valeurs selon la provenance du script, et selon la sécurité d'Internet Explorer choisie. En effet, la notion de Zone Internet est largement utilisée par le navigateur. Les modifications apportées dans les options de sécurité avec notamment l'ajout de sites/serveurs de confiances ont un impact direct sur ZoneID et par conséquent sur ce que PowerShell considère comme local ou distant.

Zone Internet	Valeur	Considérée comme local
NoZone	-1	Oui
MyComputer	0	Oui
Intranet	1	Oui
Trusted	2	Oui
Internet	3	Non
Untrusted	4	Non

d. Modifier le ZoneId ou comment transformer un script distant en script local

Vous l'avez compris, l'identifiant de zone est un peu comme un traceur qui va suivre partout votre script pour rappeler à PowerShell que vous n'en êtes pas l'auteur.

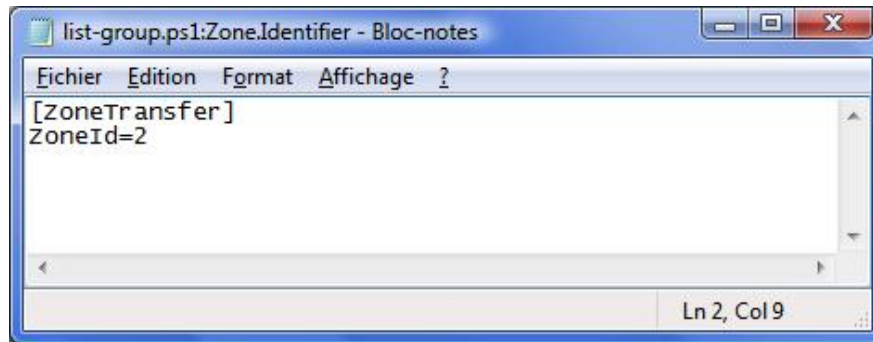
Seulement maintenant que vous êtes familiarisé avec les ADS et notamment celui créé par les outils de communication Microsoft, regardons comment faire croire à PowerShell qu'un script est un script local. Pour cela, il existe deux techniques :

- La première consiste à, comme énoncé précédemment dans ce chapitre, faire un clic droit sur le script en question et choisir **Propriétés**. Puis, sur l'onglet **Général**. Dans la partie inférieure de la fenêtre, choisir **Débloquer**.
- La seconde est un peu plus longue, mais toute aussi efficace, résulte dans le changement du ZoneID. Pour modifier l'identifiant de zone depuis le shell, commençons par ouvrir le contenu de l'ADS avec notepad :

```
PS > cmd
Microsoft Windows [version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. Tous droits réservés.

C:\> notepad PowerScript.ps1:Zone.Identifier
```

- Puis changeons la valeur du ZoneId (initialement à 3 si le script provient d'Internet) pour lui mettre la valeur 2 (Trusted).



Modification graphique de l'ADS Zone.Identifier

- Enfin, sauvegardons le fichier ADS par simple clic sur le bouton **Enregistrer**. Relançons PowerShell puis essayons d'exécuter à nouveau le script. Et là, « Eureka », le script s'exécute comme s'il s'agissait d'un script local.

5. Les chaînes sécurisées

Savoir masquer les données sensibles contenues dans vos scripts, devrait faire partie des choses courantes. Nous disons « devrait » car encore aujourd'hui, nombreux sont les scripts où des données confidentielles sont en clair. Il existe de nombreuses techniques pour dissimuler des chaînes de caractères dans un script, mais la plus efficace est la sécurisation de chaîne apportée par le Framework .NET.

Avec PowerShell, il faut bien dissocier une chaîne chiffrée d'une chaîne sécurisée. On parle de chaîne chiffrée lorsque son contenu est rendu incompréhensible pour toute personne ne disposant pas d'une clé de déchiffrement (cf. partie sur le chiffrement des chaînes), et on parle de chaînes sécurisées quand elles ont :

- **Un contenu chiffré** : le contenu des chaînes sécurisées est chiffré caractère par caractère puis enregistré en mémoire. Le chiffrement du contenu ne nécessite aucune clé, le Framework .NET chiffre lui-même les données.
- **Un accès en écriture contrôlé** : une fois créée, une chaîne sécurisée peut se voir ajouter du texte uniquement caractère par caractère. Sans oublier qu'une méthode propre au type SecureString permet de rendre l'accès en lecture seule, ce qui empêche toute modification ultérieure sur la chaîne.
- **Une non-duplication du contenu en mémoire** : PowerShell fait partie des langages objets, couramment appelés langages de haut niveau. Ce type de langage a la particularité de ne pas encombrer le scripteur avec

certains détails de programmation comme l'allocation ou la libération de la mémoire, ces opérations étant confiées au Garbage Collector (GC) ou ramasse miettes en français. Bien qu'extrêmement pratique, il arrive que le Garbage Collector effectue de nombreuses copies en mémoire pour optimiser l'allocation dynamique des variables. C'est ce qu'on appelle le « Mark and Compact ». Ainsi, pour pallier à ce problème de sécurité, une chaîne `SecureString` est stockée dans un espace mémoire non géré par le GC, et n'est jamais dupliquée en mémoire. Et une fois la variable supprimée, l'espace attribué est aussitôt effacé de la mémoire et n'y laisse aucune trace.

a. Sécuriser une chaîne

Pour sécuriser une chaîne via PowerShell, il existe deux méthodes.

La première méthode consiste à utiliser la commande **ConvertTo-SecureString** associée aux paramètres :

- **-asplaintext**, spécifiant que vous utilisez cette commande pour convertir une chaîne standard en une chaîne sécurisée.
- **-force**, spécifiant le fait que vous confirmez l'utilisation de cette commande.

Paramètre	Description
String	Le paramètre String permet de déterminer la chaîne à déchiffrer.
SecureKey	Le paramètre SecureKey permet d'utiliser une chaîne sécurisée comme valeur de clé. En réalité, la valeur de la chaîne sécurisée est convertie en tableau d'octets et peut être ainsi utilisée comme clé.
Key	Le paramètre key détermine la clé à utiliser. Pour rappel, la clé doit avoir une longueur de 128, 192 ou 256 bits. C'est-à-dire que si vous utilisez un tableau d'entiers comme clé, sachant qu'un entier est codé sur 8 bits, vous pouvez utiliser des tableaux de 16, 24 ou 32 entiers.
AsPlainText	Ce paramètre n'est pas utilisé dans le cadre du déchiffrement, il sert uniquement lorsque cette commande est utilisée pour transcrire une chaîne en une chaîne sécurisée.
Force	Le paramètre Force est utilisé en complément du paramètre <code>asPlainText</code> pour spécifier que vous souhaitez réellement sécuriser une chaîne par le biais de <code>asPlainText</code> .

Par exemple, voici la sécurisation d'un texte brut : « Bonjour »

```
PS > $chaîne = ConvertTo-SecureString 'Bonjour' -asplaintext -force
PS > $chaîne
System.Security.SecureString
```

Vous remarquerez que lorsque que nous essayons de lire cette valeur dans le shell, le résultat ne s'affiche pas, seul le type est affiché.

La seconde méthode consiste quant à elle à saisir un texte dans la console avec la commande **Read-Host** et de convertir ce texte en chaîne sécurisée grâce au paramètre **-AsSecureString**. Exemple :

```
PS > $chaîne = Read-Host -assecurestring
****
PS > $chaîne
System.Security.SecureString
```

Dans les deux cas, l'objet retourné est de type `SecureString`, et ne peut être lu directement.

Pour avoir un aperçu de ce qu'il est possible de faire avec la chaîne sécurisée que nous venons de créer, jetez un rapide coup d'œil sur le tableau suivant, qui liste les différentes méthodes associées à l'objet `SecureString`.

Méthode	Description
AppendChar	Permet de rajouter un caractère à la fin de la chaîne sécurisée.

Clear	Efface la chaîne.
Copy	Crée une copie de la valeur stockée.
Dispose	Libère toutes les ressources employées par l'objet Secure-String.
GetHashCode	Récupère sous forme d'un entier 32 bits le code de hachage.
GetType	Identifie le type : SystemString.
get_Length	Renvoie sous la forme d'un entier de 32 bits la longueur de la chaîne.
InsertAt	Permet d'insérer un caractère à un indice spécifique de la chaîne sécurisée.
IsReadOnly	Renvoie la valeur booléenne True si la chaîne est en lecture seule et False si elle ne l'est pas.
MakeReadOnly	Rend le contenu de la chaîne inaltérable. Cette opération est irréversible.
RemoveAt	Permet de supprimer un caractère à un indice spécifique de la chaîne sécurisée.
SetAt	Permet le remplacement d'un caractère par un autre à l'index spécifié.

En listant les méthodes d'un objet SecureString avec la commande que vous manipulez depuis le début du livre (à savoir Get-Member), l'observateur attentif que vous êtes n'aura pas manqué de noter l'absence de deux méthodes omniprésentes avec les objets rencontrés jusqu'à maintenant : Equals et ToString.

Ceci n'est pas un oubli de notre part mais plutôt une volonté de la part de Microsoft de ne pas laisser ces méthodes avec un objet de type SecureString, ce qui constituerait évidemment un problème de sécurité. La méthode Equals permet de tester si deux objets sont identiques : si l'égalité est respectée, alors le booléen true est renvoyé sinon c'est la valeur false qui l'est. Seulement cette méthode appliquée à un objet de type SecureString renverra toujours la valeur false même si les deux chaînes sécurisées sont identiques, exemple :

```
PS > $chaine1 = Read-Host -assecurestring
****
PS > $chaine2 = $chaine1.Copy()
PS > $chaine1.Equals($chaine2)
False
```

Cette sécurité permet ainsi d'éviter la découverte de chaînes par des méthodes automatisées de tests successifs, dites de « force brute ».

Et la méthode ToString quant à elle, permettant de transformer l'objet en chaîne de caractères, renvoie uniquement le type de l'objet System.Security.SecureString.

Regardons de plus près ce qu'il se passe lorsque vous utilisez quelques méthodes définies dans le tableau précédent.

- Tout d'abord, créons une chaîne sécurisée avec la commande **Read-Host** et insérons-y un mot de quatre lettres :

```
PS > $chaine = Read-Host -assecurestring
****
```

Vérifions ensuite sa longueur avec la commande suivante :

```
PS > $chaine.Get_Length()
4
```

La console affiche en toute logique le chiffre 4.

- Essayons maintenant d'y ajouter un caractère :

```
PS > $chaine.AppendChar('P')
PS > $chaine.Get_Length()
5
```

La longueur de la chaîne a bien augmenté d'un caractère.

Si maintenant vous souhaitez insérer plusieurs caractères, cela ne vous sera pas permis directement :

```
PS > $chaîne.AppendChar('Bonjour')
Impossible de convertir l'argument « 0 » (valeur « Bonjour »)
de « AppendChar » en type « System.Char » : « Impossible de convertir
la valeur « Bonjour » en type « System.Char ». Erreur : « La chaîne doit
avoir une longueur d'un et un seul caractère. » »
```

Mais comme rien n'est impossible, voici comment contourner le problème :

```
PS > $insert= 'Bonjour'
PS > For($i=0;$i -lt $insert.length;$i++)
{$chaîne.AppendChar($insert[$i])}
```

- Vérifions si la chaîne a bien été incrémentée :

```
PS > $chaîne.Get_Length()
12
```

- Rendons maintenant cette chaîne accessible uniquement en lecture seule :

```
PS > $chaîne.MakeReadOnly()
```

- Tentons de lui ajouter un caractère :

```
PS > $chaîne.AppendChar('P')
Exception lors de l'appel de « AppendChar » avec « 1 »
argument(s) : « L'instance est en lecture seule. »
```

En toute logique, PowerShell génère un message d'erreur car l'objet SecureString n'est plus accessible en écriture.

b. Lire une chaîne sécurisée

Vous ne serez pas surpris si nous vous disons qu'il n'existe pas de méthode proprement dite pour convertir une chaîne sécurisée en une chaîne classique. Cela est facilement compréhensible de part le fait que cela aurait pour conséquence de réduire à néant les précautions prises pour respecter certains points de sécurité énoncés plus haut. En effet, si le contenu d'une chaîne sécurisée est recopié dans une chaîne standard, elle sera alors recopiée en mémoire par le Garbage Collector, et l'information ne sera donc plus à proprement parler confidentielle.

Mais vous y êtes désormais habitués, nous avons encore une fois une solution à vous proposer, puisqu'il existe avec le Framework .NET, une classe du nom de Runtime.InteropServices.Marshal proposant deux méthodes :

- SecureStringToBSTR qui va nous permettre d'allouer la mémoire non gérée par le Garbage Collector pour y recopier le contenu de la chaîne sécurisée,
- PtrToStringUni qui recopie le contenu de la chaîne stockée dans une partie de la mémoire non gérée vers un objet de type String en Unicode qui, lui, est géré par le Garbage Collector.

Exemple :

Lire une chaîne sécurisée.

Tout d'abord, créons une chaîne sécurisée avec la commande **Read-Host** et insérons-y un mot de quatre lettres :

```
PS > $chaîneSec = Read-Host -assecurestring
****
```

- Procédons maintenant à la lecture de cette SecureString en utilisant les méthodes statiques de la classe Marshal :

```
PS > $ptr = [System.Runtime.InteropServices.Marshal]
::SecureStringToBSTR($chaineSec)
PS > $chaineClaire = [System.Runtime.InteropServices.Marshal]::
PtrToStringUni($ptr)
```

Vous remarquez que la variable \$chaineClaire est de type String et contient bien la valeur de la chaîne sécurisée.

Cependant, une fois que vous avez récupéré votre chaîne en clair, mieux vaut tout de suite l'effacer de la zone mémoire pour limiter les risques. Cela peut paraître extrême, mais nombreux sont les outils de piratage qui reposent sur une lecture des zones mémoire. Pour cela, procédons premièrement par libérer le pointeur de la chaîne non gérée par le Garbage Collector. Puis par remplacer le contenu de la variable \$chaineClaire par une valeur quelconque, et enfin forcer le Garbage Collector à s'exécuter. Traduit en PowerShell cela donne le code suivant :

```
# Libère le pointeur de chaîne
PS > [System.Runtime.InteropServices.Marshal]::
ZeroFreeCoTaskMemUnicode($ptr)

# Modification du contenu de la variable $chaine par 40 étoiles
PS > $chaineClaire = '*' * 40

# Appel du Garbage Collector
PS > [System.GC]::Collect()
```

6. Le chiffrement

Le chiffrement est une technique vieille de plus de trente siècles consistant à transformer une information claire (intelligible) en une information qui ne peut être comprise par une personne non autorisée. Cette transformation était généralement réalisée par permutation des lettres du message (transposition), ou par le remplacement d'une ou plusieurs lettres par d'autres (substitution).

Le schéma suivant met en scène les traditionnels personnages Alice et Bob cherchant à communiquer par l'intermédiaire d'un canal de transmission public tel que le réseau Internet.



Envoi d'un message confidentiel

Dans le schéma ci-dessus, les opérations de chiffrement et de déchiffrement sont symbolisées par des clés et les envois/réceptions de messages par des flèches.

Dans cette mise en scène, Alice transforme son message édité en clair en message chiffré. Puis elle le transmet à Bob, qui lui va faire la transformation inverse, à savoir, déchiffrer le message. Ainsi Alice et Bob rendent leur message incompréhensible par Oscar qui aurait pu l'intercepter durant l'échange. En effet, Oscar n'a pas de clé : il ne sait pas comment Alice a chiffré le message ni comment Bob va le déchiffrer.

Les premiers systèmes de chiffrement étaient essentiellement basés sur l'alphabet, comme le fameux code dit « César », où chaque lettre du texte à chiffrer est remplacée par une autre lettre située à la *én*ième place plus loin dans l'alphabet. Ainsi, si le décalage est de 1 le A vaut B et le B vaut C. Par exemple, prenons un décalage de 3.

```
alphabet original : A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
alphabet codé : D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
```

Ce qui donnerait :

Message original = PowerShell c'est facile

Message chiffré = SrzhuVkhoo f'hvw idfloh



L'avantage d'utiliser une chaîne chiffrée est que vous pouvez l'enregistrer dans un fichier pour éventuellement la réutiliser ultérieurement, ce qui n'est pas possible avec une chaîne sécurisée.

Notion de clé de chiffrement

Un chiffrement est généralement composé d'un algorithme fixe, auquel on va associer une clé qui elle est variable. Dans le cas du code César, l'algorithme est le décalage de lettres dans l'alphabet et la clé est le nombre de décalages à effectuer. Par exemple, si on vous donne le message suivant : « Eudyr yrxv hwhv wuhv iruw » et que l'on vous dit qu'il est chiffré avec le code César vous n'allez pas pouvoir le déchiffrer. Il faut que nous vous transmettions la valeur de la clé qui correspond au nombre de décalages à faire. À présent, si nous vous disons que la clé est 3, vous pouvez décoder le message.

Exemple :

Script qui chiffre un message avec le code César.

Comme vous pouvez le constater, le script nécessite les paramètres -texte et -cle contenant respectivement le texte à chiffrer et la clé à utiliser.

```
# Script Cesar.ps1
# Chiffrement d'un message grâce au code César

Param ([string]$texte, [int]$cle)

$message_origine = $texte
$alphabet_MAJ='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
$alphabet_MIN='abcdefghijklmnopqrstuvwxyz'

for($i=0;$i -lt $message_origine.length;$i++)
{
    $trouve = 0
    for($j=0;$j -lt $alphabet_MAJ.length;$j++)
    {
        $tmp = $cle
        while(($j+$tmp) -ge $alphabet_MAJ.length)
        {
            $tmp -= 26
        }
        If($message_origine[$i] -ceq $alphabet_MAJ[$j])
        {
            $message_modif += $alphabet_MAJ[$j+$tmp]
            $trouve = 1
        }
        ElseIf($message_origine[$i] -ceq $alphabet_MIN[$j])
        {
            $message_modif += $alphabet_MIN[$j+$tmp]
            $trouve = 1
        }
    }
    if(!$trouve) {$message_modif += $message_origine[$i]}
}
Write-host "`nMessage origine : $message_origine "
Write-host "`n`nMessage Code : $message_modif `n"
```

Résultat dans la console PowerShell :

```
PS > ./cesar.ps1 -texte "PowerShell c'est facile" -cle 14

Message origine : PowerShell c'est facile

Message Code : DcksfGvszz q'sgh toqwsz
```

Le chiffrement à clé symétrique

Également appelé chiffrement à une clé, c'est ce système qui est utilisé par PowerShell pour chiffrer un message. Sa principale caractéristique est que l'émetteur et le récepteur utilisent tous deux la même clé pour chiffrer et déchiffrer le message. Dans l'exemple du code César, si l'émetteur chiffre son message avec une clé de 13, le récepteur doit impérativement utiliser la même valeur pour effectuer la rotation dans le sens inverse et pouvoir ainsi déchiffrer le message. C'est donc un système symétrique.

Bien évidemment ce principe nécessite que la clé reste secrète durant toute la transaction.

Le chiffrement de chaînes avec PowerShell repose sur l'algorithme de Rijndael en version AES pour *Advanced Encryption Standard* (Standard de Chiffrement Avancé).

Ce système, créé à la fin des années 90 par deux chercheurs belges, utilise le principe du chiffrement à clé symétrique de longueur 128, 192 ou 256 bits. Par conséquent, lors du chiffrement de vos messages, vous devrez prendre soin de bien noter votre clé.



Le système de clé symétrique trouve ses limites lorsque plusieurs personnes cherchent à transmettre des messages chiffrés entre eux. Si cinq personnes constituent un réseau d'échange de message secret, chaque personne doit connaître la clé secrète des quatre autres personnes. Ce qui constitue déjà un bon nombre de clés.

a. Chiffrer une chaîne

Le chiffrement de chaînes avec PowerShell est réalisé à partir de la commande suivante : **ConvertFrom-SecureString**. Le nom explicite de la commande (« convertir depuis une chaîne sécurisée ») laisse deviner que pour chiffrer une chaîne, il faut auparavant s'assurer qu'il s'agisse bien d'une chaîne sécurisée de type `SecureString`. À vous donc de transformer une chaîne de caractères en une chaîne sécurisée puis d'utiliser la commande **ConvertFrom-SecureString**.

Cette commande dispose de trois paramètres (hors paramètres communs), dont voici le détail :

Argument	Description
<code>secureString</code>	Le paramètre <code>secureString</code> permet de déterminer la chaîne à chiffrer. Notez bien que cette chaîne doit être de type <code>SecureString</code> .
<code>key</code>	Le paramètre <code>key</code> détermine la clé à utiliser. Pour information, la clé doit avoir une longueur de 128, 192 ou 256 bits. C'est-à-dire que si vous utilisez un tableau d'entiers comme clé et qu'un entier est codé sur 8 bits, vous pouvez utiliser des tableaux de 16, 24 ou 32 entiers.
<code>secureKey</code>	Le paramètre <code>secureKey</code> permet d'utiliser une chaîne sécurisée comme valeur de clé. En réalité, la valeur de la chaîne sécurisée est convertie en tableau d'octets et peut être ainsi utilisée comme clé.



Si aucune clé n'est spécifiée, PowerShell utilisera l'API Win32 DPAPI (Data Protection API) pour chiffrer et déchiffrer les données.

Pour mieux comprendre comment chiffrer un texte, voici quelques exemples d'applications.

Exemple :

Chiffrer une chaîne sans clé.

Premièrement, commençons par créer une chaîne sécurisée qui va contenir nos informations confidentielles :

```
PS > $secure_string_pwd = ConvertTo-SecureString `
"Code d'entrée bâtiment : 101985" -asplaintext -force
```

*Convertissons ensuite la chaîne sécurisée en chaîne chiffrée avec la commande **ConvertFrom-SecureString** sans spécifier de clé puis, redirigeons le résultat dans un fichier texte :*

```
PS > ConvertFrom-SecureString $secure_string_pwd > c:\chaine_c1.txt
```

En récupérant le contenu du fichier par la commande **Get-Content**, on aperçoit que celui-ci contient bien des informations chiffrées.

```
PS > Get-Content chaine_c1.txt
```

```
01000000d08c9dddf0115d1118c7a00c04fc297eb010000004fead97202caa34ea690fa1977
c9f65c00000000020000000000106600000001000020000000e52c11a9585b3b5e806c96ec
5d842a9ec023260186eaffcef7600174e2b5823900000000e8000000002000020000000914
e80fd65aaf64d1c4115760e1843fd052e87e420776ebb880e7c816d1d3ab54000000090dd
7b8320b41021335ce0b55e0ab0e3e0639c065aab25f8de74997539fb18b99658b241f3ffb6
dd564e4416856d394758381b6dd0b0d4f88ce12ab6b33252b40000000157f05d540c2899085
8c5016669d5a952de2e6b44257d0775b83c32badb2e90055d3bd7ef9f844bc00efe20c0ca5
f1ea02e28f51e25b0b56c48444596a703d73
```

Exemple : chiffrer une chaîne avec une clé de 256 bits.

Regardons à présent de plus près comment chiffrer une chaîne en utilisant une clé de 256 bits, c'est-à-dire de 32 octets ($32 \times 8 = 256$). Commençons une nouvelle fois par créer la chaîne sécurisée :

```
PS > $secure_string_pwd = ConvertTo-SecureString "Code d'entrée
batiment : 101985" -asplaintext -force
```

Puis, créons notre clé de 32 octets en spécifiant des valeurs inférieures à 256 et affectons-la à la commande **ConvertFrom-SecureString** via le paramètre **Key**, pour enfin renvoyer le tout vers un fichier texte :

```
PS > $cle = (6,10,19,85,4,7,19,87,13,3,20,13,3,6,34,43,56,34,23,14,87,56,
34,23,12,65,89,8,5,9,15,17)

PS > ConvertFrom-SecureString -secureString $secure_string_pwd `
-key $cle > c:\temp\chaine_c2.txt
```

Le fichier contient cette fois des informations chiffrées, mais cette fois-ci, elles l'ont été avec la clé de 256 bits que nous avons nous-même spécifiée.

```
PS > Get-Content chaine_c2.txt
```

```
76492d1116743f0423413b16050a5345MgB8AFQARgB1AGQAYwBCADYAeQB0AFIAUwBjADEAVw
AxAEwAMQBFAFUArgBwAEEAPQA9AHwAZAA0ADQA0ABhADgAYQBmADcAMAA0AGMAMgA1ADkAMAAxA
GYANQAxADkAMwBLAGeAZQBjADgAYwA0ADQAYwBkADUA0ABiAGYAMABiADcAMQBkADEAYgAwAGYA
NgA2AGMAZgAxADUAMQA1ADEA0QA2AGEAMQB1ADkAYgBmADUAYgA0ADAAZgBkAGQAMgA4AGYAZgA
yAGEAZgA0ADEAYwA5ADUAYwA3AGUANwBhADIAYQBkADMANABLAGeAMgA3ADUANABmADEAMwBLAG
QANABhAGIANAAZADEAZQA2AGEA0QA3AGIAMwA4AGEAMwBLAGUAMgBjADYAMQAwADkANAAyAA==
```

Exemple : chiffrer un texte avec une chaîne sécurisée.

Enfin, pour terminer, essayons maintenant de chiffrer du texte avec une clé de 128 bits en utilisant une chaîne de caractères comme clé, ce qui est beaucoup plus pratique que de retenir ne serait-ce que 16 nombres.

On sait qu'une variable de type « char » est destinée à représenter n'importe lequel des 65 536 caractères Unicode sur deux octets (soit 16 bits). Par conséquent, il faut utiliser 8 caractères ($128/16=8$) pour atteindre 128 bits. Pour qu'une chaîne puisse être utilisée comme clé, celle-ci doit absolument être sécurisée.

Commençons par créer notre chaîne sécurisée qui va nous servir de clé (sur 8 caractères exactement) :

```
PS > $cle = ConvertTo-SecureString 'tititata' -asplaintext -force
```

Puis, une fois encore, sécurisons la chaîne qui contiendra nos informations confidentielles :

```
PS > $secure_string_pwd = ConvertTo-SecureString `
"Code d'entrée batiment : 101985" -asplaintext -force
```

Et pour finir, chiffrons la chaîne avec la commande **ConvertFrom-SecureString**, mais cette fois en spécifiant le paramètre **-securekey** associée à la chaîne chiffrée qui nous sert de clé :

```
PS > ConvertFrom-SecureString -secureString
$secure_string_pwd `
```



```
-securekey $cle > c:\temp\chaine_c3.txt
```

Et si nous regardons le contenu du fichier, nous observons bien entendu une chaîne chiffrée, mais qui l'aura été avec pour clé de chiffrement, une chaîne sécurisée.

```
PS > Get-Content chaine_c3.txt
01000000d08c9ddf0115d1118c7a00c04fc297eb010000004fead97202caa34ea690fa1977
c9f65c00000000020000000000106600000001000020000000e31d5db17eae3b1c090e8675
c121bee68cb020a7915a757a694182edf5ff527000000000e8000000002000020000000e
984c798e87b81ea7097da0be4928fd0d7801f49f96cc2634c45606bbb65ed040000000a2d
4d91360ae371e89cbf3380b42e6da662d3e135d96832798147392bc3ba19473876ac5b99
dcfe68c7bde416f90fd8f774c7666487594526e9061b53569dcd54000000045c391b33970
cd3341e1737605d3462a90bcb151cbb3591c5c341e2cab16a360cfc82cdfec8496453ea8
b5987f422e5a66d25e1e2575b5ef84e10be4f748c1e
```

b. Déchiffrer un texte

Reprenons le texte chiffré dans la section précédente grâce à une clé de 256 bits. Pour réaliser l'opération inverse, à savoir déchiffrer, nous utilisons la commande **ConvertTo-SecureString** assortie de la clé correspondante :

```
PS > $chaine_chiffree = Get-Content .\chaine_c2.txt
PS > $chaine_originale = ConvertTo-SecureString -key $cle -string
$chaine_chiffree
```

À la suite de cette commande, la variable `$chaine_originale` contient non pas le texte en clair, mais le texte brut sous forme d'une chaîne sécurisée.

Voici ce qu'indique PowerShell lors de la lecture de la variable `$chaine_originale` dans la console :

```
PS > $chaine_originale
System.Security.SecureString
```

Reste donc à utiliser une méthode du Framework .NET pour voir ce que nous cache cette chaîne.

```
PS > $ptr = [System.Runtime.InteropServices.Marshal]
::SecureStringToBSTR($chaine_originale)
PS > [System.Runtime.InteropServices.Marshal]
::PtrToStringUni($ptr)
```

Et bien entendu, le résultat correspond bien au texte saisi initialement.

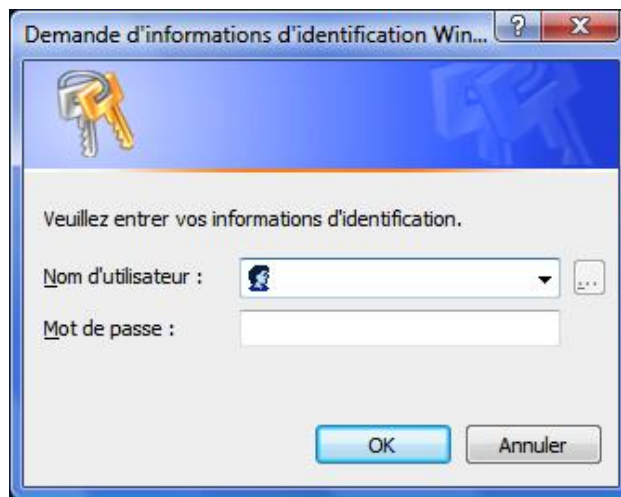
```
Code d'entrée bâtiment : 101985
```

7. Les credentials

Dans le dur monde de la sécurité informatique, la première façon de minimiser les risques est de ne pas travailler au quotidien avec un compte administrateur. Bien que très connue, cette règle est loin d'être appliquée partout. Il est clair qu'effectuer plusieurs fois la même opération, à savoir : fermer une session utilisateur pour en ouvrir une avec un compte administrateur pour effectuer une action particulière, puis refermer pour enfin rouvrir une session avec son compte utilisateur limité peut vous faire perdre patience...

Remarquez que nous pouvons aussi lancer une console PowerShell avec l'option **Exécuter en tant qu'administrateur**. Dans ce mode, toutes les commandes tapées dans la console seront lancées avec les droits administrateurs.

La commande **Get-Credential** d'un genre particulier permet d'obtenir les « credentials », c'est-à-dire le couple login/mot de passe d'un utilisateur. Ainsi grâce à ce mécanisme, un utilisateur peut s'authentifier sous un autre compte. Utilisée simplement dans la console, `Get-Credential` affiche une interface graphique, qui propose d'entrer un nom d'utilisateur ainsi que le mot de passe associé, et retourne sous forme d'un objet `PSCredential` le nom d'utilisateur en clair ainsi que le mot de passe sous forme de chaîne sécurisée.



Interface graphique de la commande `Get-Credential`

Exemple :

Valeur de retour pour le nom d'utilisateur « Administrateur ».

UserName	Password
-----	-----
\Administrateur	System.Security.SecureString

Cette commande dispose également du paramètre facultatif : **-credential** qui permet de saisir le nom de l'utilisateur directement dans la ligne de commande. De cette façon, il ne reste plus qu'à saisir le mot de passe.

Prenons un autre exemple avec la suppression d'un fichier avec la commande **Remove-Item**. Si vous effectuez la commande avec un compte limité avec lequel vous n'avez aucun droit sur ce fichier, voici ce que PowerShell affiche :

```
Remove-Item : Impossible de supprimer l'élément C:\temp : L'accès
au chemin d'accès 'C:\temp' est refusé.
```

Maintenant, appliquons à cette même commande les droits d'un utilisateur autorisé à supprimer ce type de fichier. Pour cela, il suffit d'ajouter au paramètre **-credential** la valeur retournée par la commande **Get-Credential**.

Exemple :

```
PS > Remove-Item FichierASupprimer -credential $(get-credential)
```

Ainsi lors de l'exécution, la fenêtre associée à la commande **Get-Credential** propose d'entrer les informations d'identification d'un utilisateur avec les droits appropriés, et transmet ces informations à la commandelette qui peut alors s'exécuter en tant qu'utilisateur spécifié. Voici un regroupement des commandes comptant **-credential** parmi leurs paramètres.

Add-Content	Move-ItemProperty
Clear-Content	New-Item
Clear-Item	New-Item-Property
Clear-ItemProperty	New-PSDrive
Copy-Item	New-Service
Copy-ItemProperty	Remove-Item
Get-Content	Remove-ItemProperty
Test-Path	Rename-Item
Get-Item	Rename-ItemProperty
Get-ItemProperty	Resolve-Path
Get-WmiObject	Set-Content
Invoke-Item	Set-Item
Join-Path	Set-ItemProperty
Move-Item	Split-Path

Exemple :

Prenons une situation courante, la création d'un fichier. Seulement voilà, pour que cette opération soit réalisée, il faut que vous disposiez des droits adéquats. Regardons ce qu'il se passe lors d'une tentative de création de fichier sans avoir les droits suffisants pour le faire.

```
PS > New-Item -name essai.txt -type file

New-Item : L'accès au chemin d'accès 'C:\Users\Robin\Documents\Temp\
essai.txt' est refusé.
Au niveau de ligne : 1 Caractère : 9
+ New-Item <<<< -name essai.txt -type file
```

- Essayons maintenant d'utiliser la même commande, mais en y ajoutant le paramètre **-credential**. Pour cela, tapons cette commande :

```
PS > New-Item -name essai.txt -type file -credential `
$(get-credential -credential administrateur)
```

La boîte de dialogue nous demandant nos informations d'authentification s'ouvre, et en saisissant le couple login/mot-de-passe ayant les droits d'accès adéquats le tour est joué.

Exemple :

Utilisation d'un objet WMI avec utilisation des credentials.

Bien que nous n'ayons pas encore abordé les objets WMI (*Windows Management Instrumentation*), nous allons faire un petit saut en avant pour vous montrer comment appliquer les authentifications d'utilisateur aux requêtes WMI distantes. Imaginons un instant que pour une raison diverse, vous soyez contraint d'interroger l'ordinateur local pour connaître le nom des disques logiques présents. La requête locale est la suivante :

```
PS > foreach($i in $(Get-WmiObject Win32_LogicalDisk)){ $i.name }
```

Jusque-là pas de difficulté particulière. À présent, pour effectuer cette requête sur un ordinateur distant nous devons utiliser le paramètre **-computer**. Heureusement pour nous, `Get-WmiObject` prend en charge les credentials ; nous allons donc pouvoir lui passer des credentials administrateur pour accéder à cette machine.

```
PS > foreach($i in $(Get-WmiObject Win32_LogicalDisk `
-credential $(get-credential) -computer 192.168.1.2)){ $i.name }
```

8. Masquer un mot de passe

Vous connaissez sûrement l'expression : « Pour vivre heureux, vivons cachés ! » ? Et bien à n'en pas douter, cette maxime s'applique également aux mots de passe. Le mot de passe est un élément critique de la chaîne « sécurité informatique » et c'est une erreur que de le saisir en clair dans un fichier. C'est un peu comme écrire son code bancaire sur sa carte bleue.

Avant de penser à masquer la saisie des mots de passe, regardez en premier lieu si l'utilisation de solutions comme les credentials (voir section Les credentials, de ce chapitre) ou encore l'exécution de script avec un compte de service ne suffirait pas à vos affaires.

Et si vous avez véritablement besoin d'inviter l'utilisateur à rentrer un mot de passe durant l'exécution du script, nous vous proposons plusieurs méthodes pour rendre cette opération la plus confidentielle possible.

a. Utilisation de la commande **Read-Host**

Comme vu dans la partie traitant des chaînes sécurisées, la commande **Read-Host** associée au paramètre **-AsSecureString** permet de rendre confidentielle la saisie d'un mot de passe. En tapant la commande suivante, chaque caractère saisi dans le Shell sera traduit par l'affichage du caractère étoile (*), et lorsque la saisie est finie, la variable \$psw reçoit un objet de type `SecureString`.

```
PS > $psw = Read-Host -assecurestring
****
```

L'utilisation de la commande **Read-Host** est à la fois la façon la plus simple de masquer la saisie d'un mot de passe,

mais également la façon la moins conviviale.

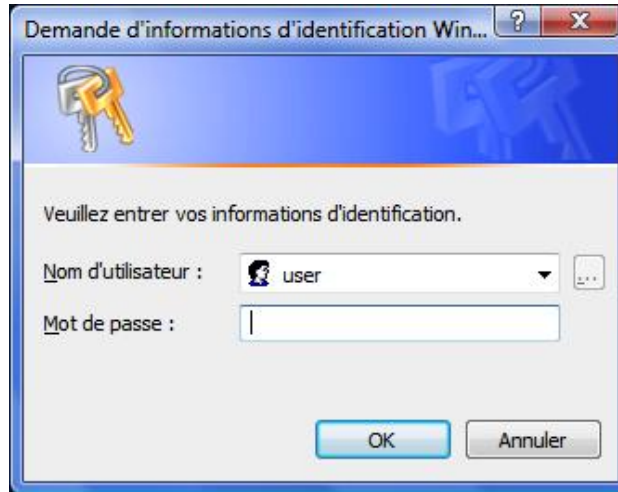
b. Utilisation de la commande Get-Credential

Bien que le but de cette commande soit de récupérer les informations liées à un autre compte que l'utilisateur courant, elle peut également servir à récupérer un mot de passe via un usage détourné.

En tapant la commande suivante, l'interface graphique de Get-Credential, native de PowerShell, vous invite à saisir un mot de passe que nous stockons, sous forme de chaîne sécurisée, dans la variable \$password. Notez que le champ « nom d'utilisateur » est déjà saisi, et cela grâce au paramètre **-credential**.

Exemple :

```
PS > $password = (get-credential -credential 'user').password
```



Interface graphique Get-member pré-remplie

➤ Si seul le mot de passe vous intéresse, le nom d'utilisateur n'a donc aucun intérêt et vous n'utiliserez que la propriété password de l'objet PSCredential. Cependant, si vous souhaitez également récupérer le nom d'utilisateur, vous pouvez utiliser la propriété UserName. Exemple : \$User = (Get-Credential).Username

c. Utilisation d'une interface graphique personnalisée

Une troisième option consiste à développer une petite interface graphique qui pourra également vous servir par la suite dans diverses applications. L'avantage est que vous allez pouvoir la personnaliser de façon à mieux l'adapter à vos besoins. Dans celle que nous vous proposons, le mot de passe est retourné via la variable \$retour sous forme d'une chaîne sécurisée. Bien que nous n'ayons pas encore abordé les formulaires graphiques de PowerShell (voir le chapitre .NET), cette fonction vous donne un rapide aperçu des possibilités graphiques offertes grâce au Framework .NET.

```
#Get-password.ps1
#Interface graphique permettant de récupérer un mot de passe

# Chargement de l'assembly pour les forms graphiques
[void][System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')

# création de la forme principale
$form = new-object Windows.Forms.form
# dimensionnement de la forme
$form.Size = new-object System.Drawing.Size(360,140)
$form.Text = 'Get-Password'
# Création bouton valider
$bouton_valider = new-object System.Windows.Forms.Button
$bouton_valider.Text = 'Valider'
#Positionnement du bouton
$bouton_valider.Location = new-object System.Drawing.Size(135,60)
```

```

# Ajustement de la taille
$bouton_valider.size = new-object System.Drawing.Size(90,25)
# Création d'une boîte de texte
$txtbox = new-object System.Windows.Forms.textbox
#Positionnement de la zone de texte
$txtbox.Location = new-object System.Drawing.Size(80,20)
# Ajustement de la taille
$txtbox.size = new-object System.Drawing.Size(200,25)
# Utilisation de la méthode permettant de cacher le texte saisi
$txtbox.set_UseSystemPasswordChar($true )

# Action du bouton Valider
$bouton_valider.add_Click(
    {
        # Conversion de la chaîne reçue en chaîne sécurisée
        $form.hide()
        $result = ConvertTo-SecureString -string "($txtbox.get_Text())" -asplaintext
    -force
    })

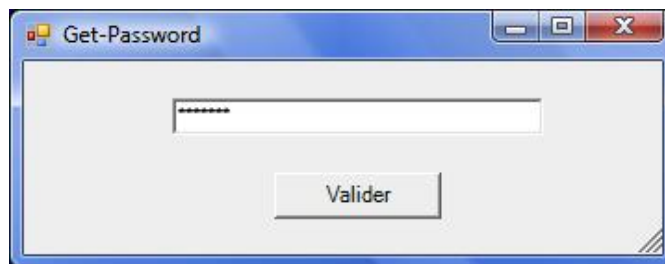
#Ajout des composants et affichage de la forme
$form.Controls.Add($bouton_valider)
$form.Controls.Add($txtbox)
$form.Add_Shown(
    {
        $form.Activate()
    })

[void]$form.showdialog()
$result

```

Notez que nous utilisons une méthode particulière de l'objet TextBox :

Set_UseSystemPasswordChar : permet de remplacer le texte saisi dans une zone de texte par des étoiles.



Interface graphique du script Get-Password.ps1



Pour augmenter encore d'un cran la sécurité dans cet exemple, nous devrions comme nous l'avons fait pour d'autres exemples « nettoyer proprement » la variable ayant contenu le mot de passe en clair, puis forcer le déclenchement du garbage collector.

Signature des Scripts

1. Les signatures numériques

Une signature numérique est un « procédé » qui permet l'identification du signataire et qui garantit l'intégrité du document. Les signatures numériques, appelées aussi signatures électroniques, sont utilisées par PowerShell pour modérer l'exécution de script selon la stratégie d'exécution choisie.

D'un point de vue conceptuel, une signature numérique correspond généralement au chiffrement, à l'aide d'une clé privée, d'une forme abrégée du message appelée « empreinte ». L'empreinte d'un message est le résultat obtenu après avoir appliqué une fonction de hachage au contenu du document.

De l'autre côté de la chaîne, le destinataire s'assure que les données n'ont pas été modifiées durant le transfert en effectuant une comparaison entre : l'empreinte qui accompagne le document déchiffré grâce à la clé publique, et l'empreinte qu'il a lui-même recalculée. Si les deux empreintes sont identiques, alors cela signifie que les données sont intègres.

2. Les certificats

Un certificat est indissociable d'une clé publique, c'est cet élément-là qui va permettre d'associer une clé à son propriétaire. C'est-à-dire que lorsque l'on déchiffre une signature avec une clé publique, il est plus que nécessaire de vérifier que cette clé est bien celle du signataire. À l'instar d'un document officiel, un certificat contient des informations sur l'identité du propriétaire. Ajouter une signature à un script, signifie que vous devez être en possession d'un certificat électronique de signature, qui permet d'identifier de façon unique la personne qui signe le script. Ce certificat peut être obtenu de diverses façons : soit vous choisissez d'acheter un certificat de signature auprès d'autorités de certification reconnues, soit vous créez vous-même un certificat (certificat « auto-signé »).

a. Acheter un certificat

Pour acheter un certificat, il faut passer par un organisme de certification (*Certificate Authority* ou CA) qui vous délivre un certificat de classe 1, 2 ou 3, défini selon un usage et un niveau de protection donnés. Ces certificats sont également associés à une durée de vie et une certaine garantie en fonction du prix qui peut aller de quelques dizaines à plusieurs centaines d'euros. Certains organismes sont même reconnus nativement par Windows, ce qui permet de déployer des scripts signés sans manipulation particulière.

b. Créer un certificat auto-signé

Créer un certificat auto-signé est, comme on peut s'y attendre, la solution la moins onéreuse. En réalité elle est même gratuite. Cette solution est donc à privilégier pour ceux qui disposent d'un budget serré.

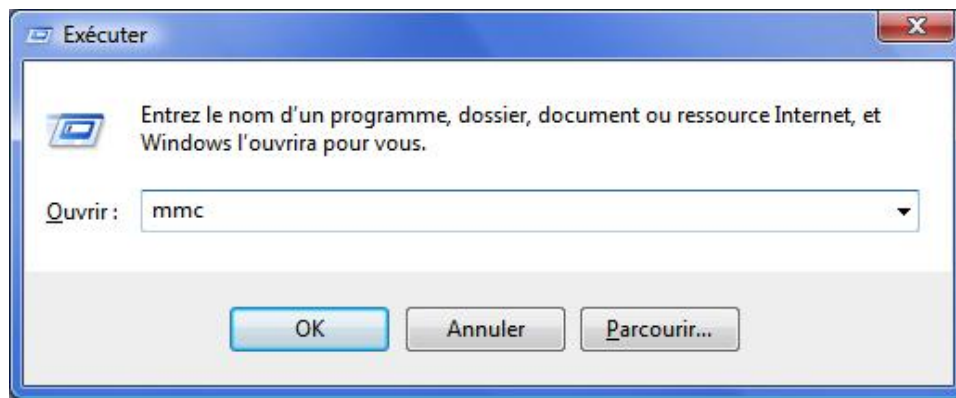
Une des choses importantes à savoir est que lorsque vous créez vous-même un certificat, l'ordinateur sur lequel ce certificat a été créé, devient une « autorité de certification », et cette autorité devra être approuvée par tous les ordinateurs exécutant les scripts que vous avez signés.

Pour créer un certificat auto-signé, il faut dans un premier temps télécharger et installer le SDK (*Software Development Kit*) du Framework .NET 3.5 (ou 2.0) qui est disponible sur le site de Microsoft.

Le SDK comprend de nombreux outils intéressants, mais celui qui nous intéresse plus particulièrement est l'outil **makecert.exe** qui comme son nom laisse à deviner, sert à créer des certificats.

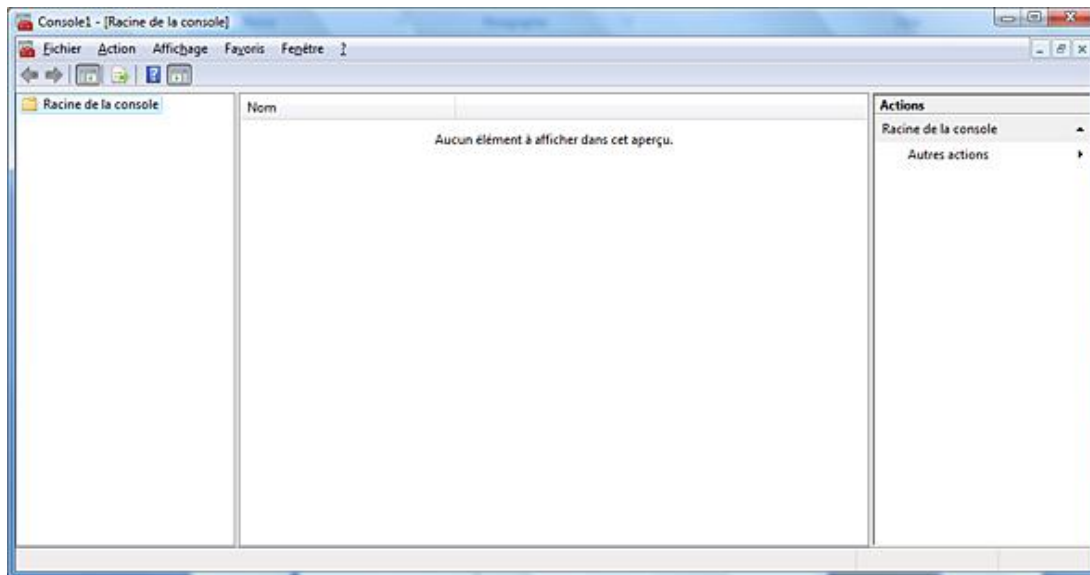
Avant d'utiliser l'exécutable **makecert.exe**, nous vous proposons de vous familiariser avec la MMC (*Microsoft Management Console*) de façon à jeter un rapide coup d'œil sur les éditeurs déjà approuvés sur votre ordinateur.

Pour la lancer, il suffit de taper **mmc** depuis le programme **Exécuter**.

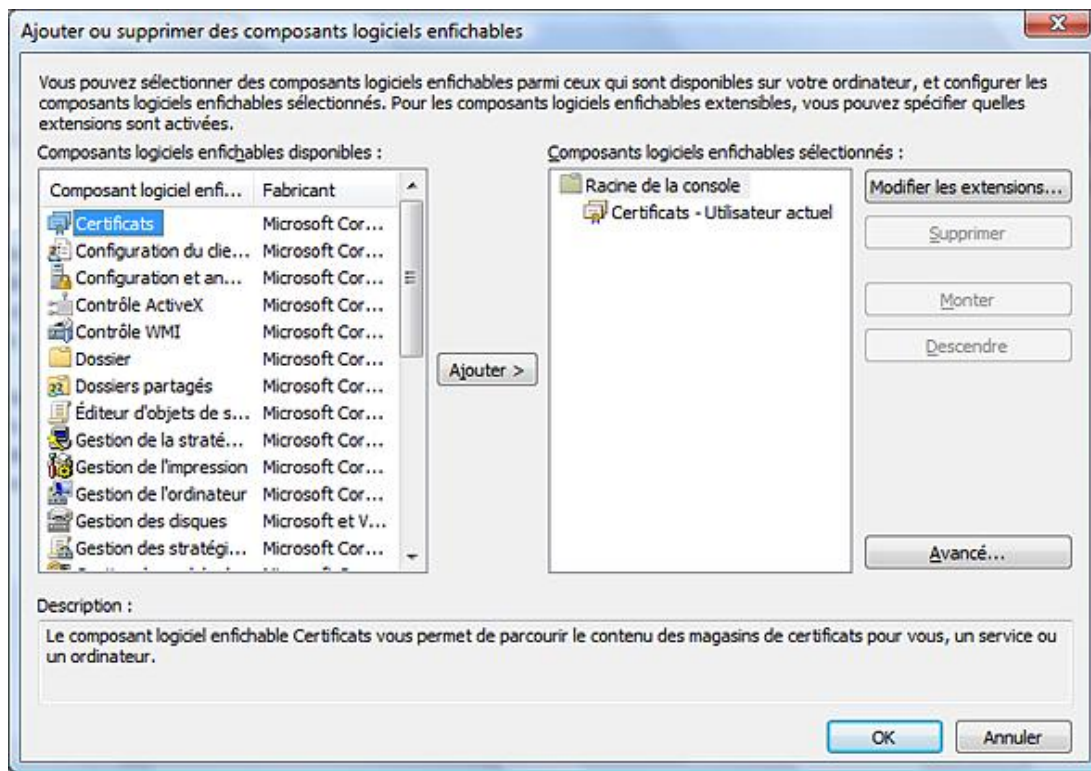


Interface de l'application « exécuter » pour lancer la console de management

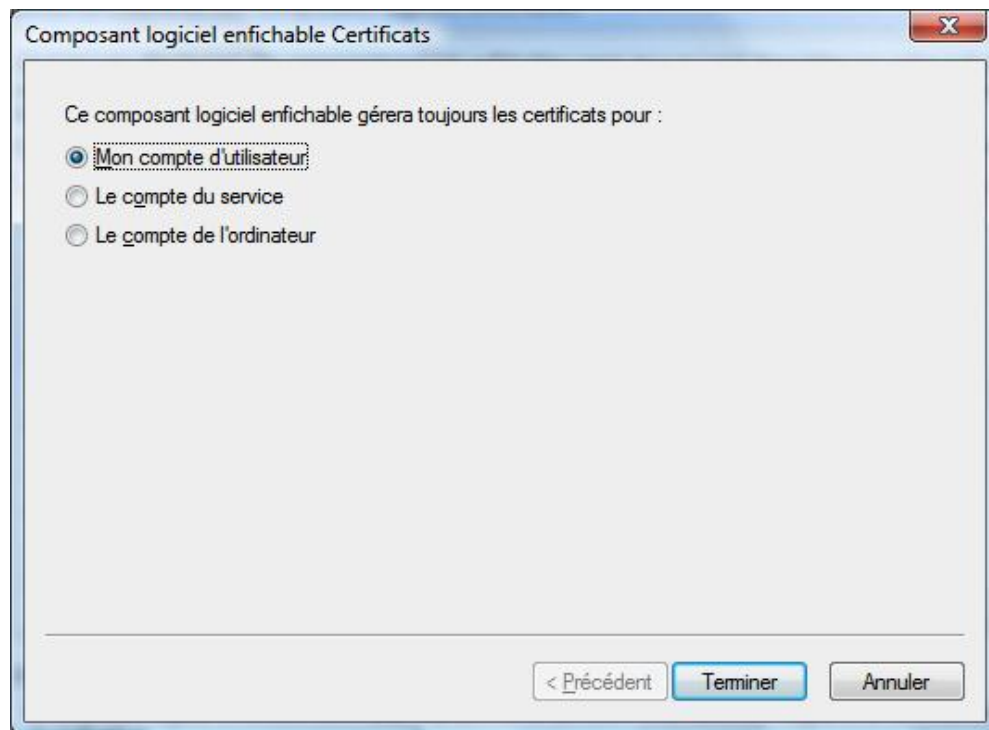
À l'ouverture la console est vide, et c'est à vous d'ajouter ce que l'on appelle des Snap-ins (« composants logiciels enfichables »). Pour cela, cliquez sur **Fichier** et **Ajouter/supprimer un composant logiciel enfichable**.



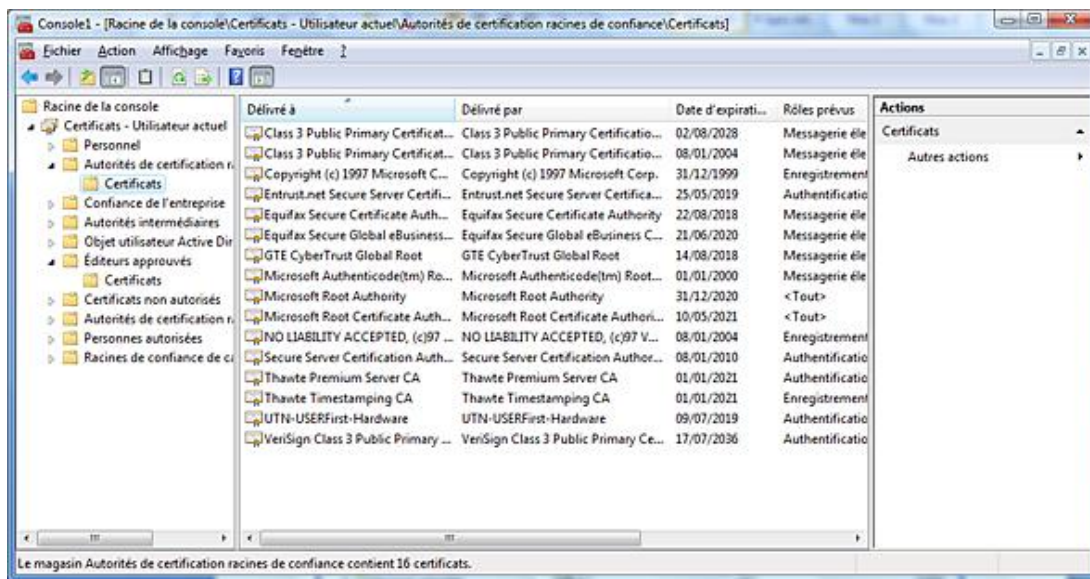
- Sélectionnez **Ajouter** puis dans la nouvelle fenêtre choisissez le composant certificats.



- Terminez la sélection en choisissant **Mon compte d'utilisateur** à la question : **Ce composant logiciel enfichable gèrera toujours les certificats pour :**



Vous voilà enfin avec la console bien configurée pour observer vos certificats.



Dans cette console, nous nous intéresserons essentiellement aux certificats personnels, aux autorités de certification racine de confiance et aux éditeurs approuvés de façon à voir comment ils évoluent au cours du temps.



N'oubliez pas de rafraîchir la console avec l'icône approprié, ou la touche [F5] pour voir apparaître les modifications que l'on va apporter.

Maintenant que vous êtes prêts, passons aux choses sérieuses.

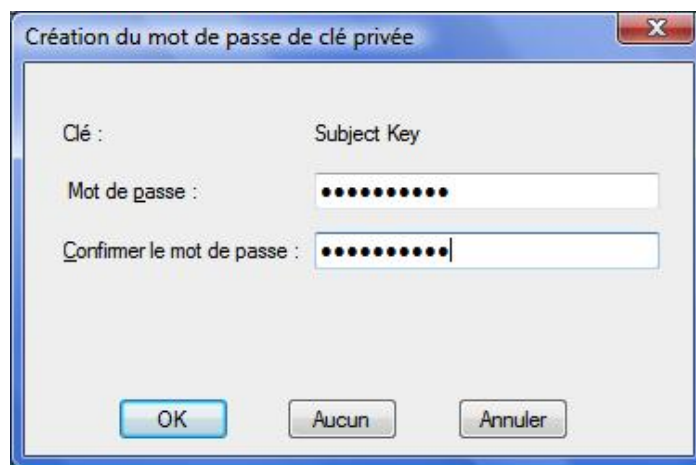
La première commande est à taper en mode administrateur dans l'invite de commandes du kit de développement :

```
C:\> makecert.exe -n "CN=Certificat Racine PowerShell" -a sha1 \
-eku 1.3.6.1.5.5.7.3.3 -r -sv root.pvk root.cer \
-ss Root -sr localMachine
```

La ligne de commande ci-dessus fait appel à l'outil **makecert.exe** pour créer un certificat d'autorité de certification racine sur votre machine. Pour mieux comprendre cette commande détaillons les options utilisées :

Option	Description
-n	Définit le nom du publicateur du certificat.
-a	Définit un algorithme de chiffrement à utiliser.
-eku	Spécifie un <i>Object Identifier</i> (OID) qui sert à préciser comment le certificat sera utilisé par une application. Par exemple, 1.3.6.1.5.5.7.3.3 désigne le certificat comme étant utilisable pour signer des scripts. Pour plus d'informations, une recherche sur msdn avec le mot clé <i>IX509ExtensionMSApplicationPolicies</i> vous indiquera les différents OID utilisables dans le cadre de la création d'un certificat.
-r	Indique la création d'un certificat auto-signé.
-sv	Définit le fichier «.pvk» de clé privée associé au certificat «.cer».
-ss	Définit le nom du magasin de certificat qui va contenir le certificat créé.
-sr	Définit à quel endroit dans la base de registre le magasin de certificat est enregistré.

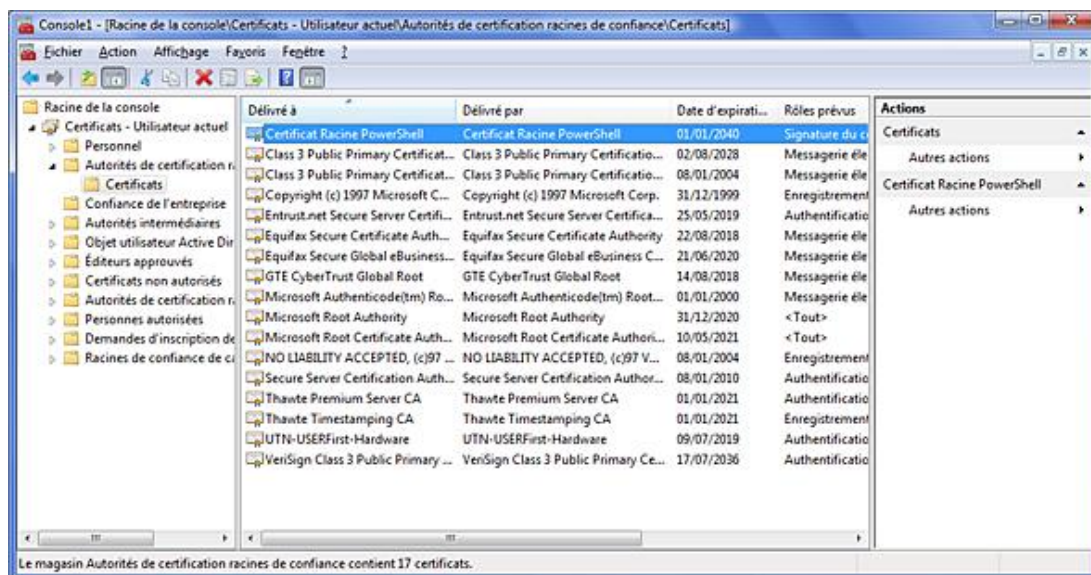
La commande exécutée, vous êtes invité à saisir le mot de passe de votre clé privée, clé qui sera indispensable à la création du certificat.



Après avoir cliqué sur **OK**, il vous sera demandé une nouvelle fois de ressaisir le mot de passe de votre clé privée.



Si vous rafraîchissez votre console de management, vous pourrez constater que la nouvelle autorité de certification que nous venons de créer est présente dans le magasin *Autorités de certification racine*.



Maintenant que votre ordinateur est une autorité de certification. Nous allons donc pouvoir créer un certificat personnel délivré par cette même autorité de certification. Et pour cela, nous devons utiliser la commande suivante :

```
C:\> makecert.exe -pe -n "CN=Mon Entreprise" -ss MY -a sha1`  
-eku 1.3.6.1.5.5.7.3.3 -iv root.pvk -ic root.cer
```

Cette commande comprend de nouvelles options dont voici le détail :

Option	Description

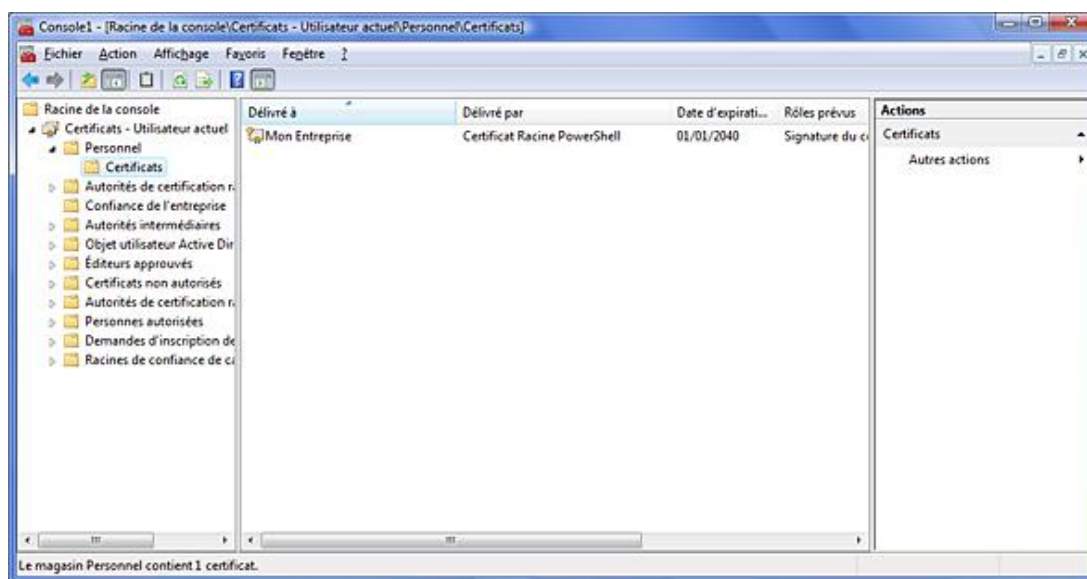
-pe	Permet d'inclure la clé privée dans le certificat.
-iv	Spécifie le fichier de clé privée .pvk.
-ic	Spécifie le fichier de certificat «.cer» racine.

- Dans l'interface, saisissez le mot de passe de votre clé privée (la même que précédemment) et cliquez sur **OK**.



L'opération est maintenant terminée, vous avez créé un certificat qui va vous permettre par la suite de signer vos scripts. Pour en vérifier la création, retournez dans la console de management et rafraîchissez-la. Sélectionnez ensuite dans l'arborescence de gauche **Personnel** puis **Certificats**.

Comme vous pouvez le constater, un certificat a bien été délivré par l'autorité de certification que nous avons nous même créée.



3. Signer votre premier script

Pour signer un script, il faut dans un premier temps que vous soyez en possession d'un certificat adéquat. Pour le vérifier, listons tous les certificats contenus dans le lecteur « cert : » avec la commande : **Get-ChildItem cert: -r -codesign**

En toute logique, si vous avez suivi la démarche sur « comment créer un certificat auto-signé », vous devriez apercevoir le certificat suivant :

```
PS > Get-ChildItem cert: -r -codesign
```

```

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

```

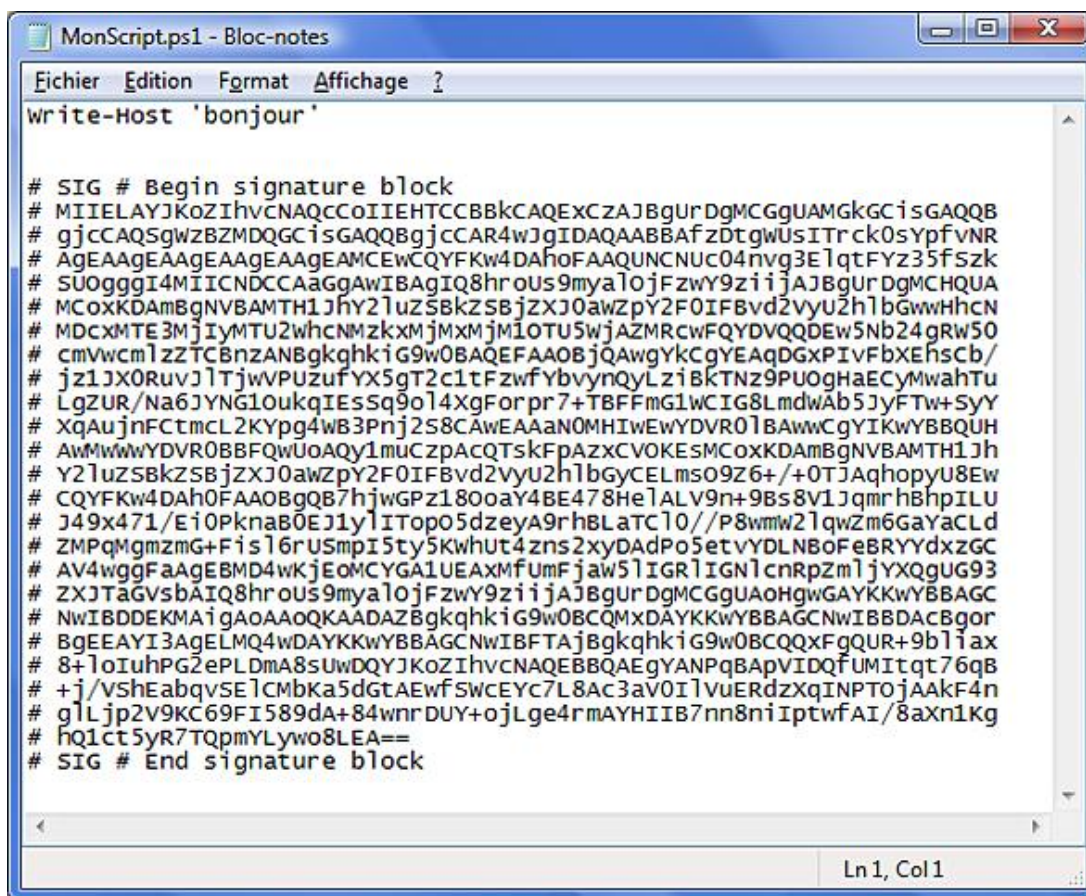
Thumbprint ----- 63044B1785C5A3ED410E487030A91BD4D99B9800	Subject ----- CN=Mon Entreprise
---	---------------------------------------

L'application d'une signature numérique à un script nécessite l'utilisation de la commande suivante :

```
PS > $cert = Get-ChildItem cert:\CurrentUser\my -CodeSigningCert
PS > Set-AuthenticodeSignature 'c:\Temp\MonScript.ps1' -cert $cert
```

Répertoire : C:\Temp	Status	Path
SignerCertificate	-----	-----
59D92A70DAAEE402A0BDFCFB3C4FD25B7A984C7E	Valid	MonScript.ps1

Après signature du script, vous pouvez l'ouvrir avec Notepad et observer votre signature à la fin du fichier.



Vous venez de signer votre premier script.

4. Exécuter des scripts signés

Lorsque vous exécutez pour la première fois un script signé sur un poste autre que celui sur lequel vous avez signé le script, PowerShell affiche le message d'erreur suivant :

```
PS > .\MonScript.ps1
Impossible de charger le fichier C:\Temp\MonScript.ps1.
Une erreur interne de chaînage des certificats s'est produite.
```

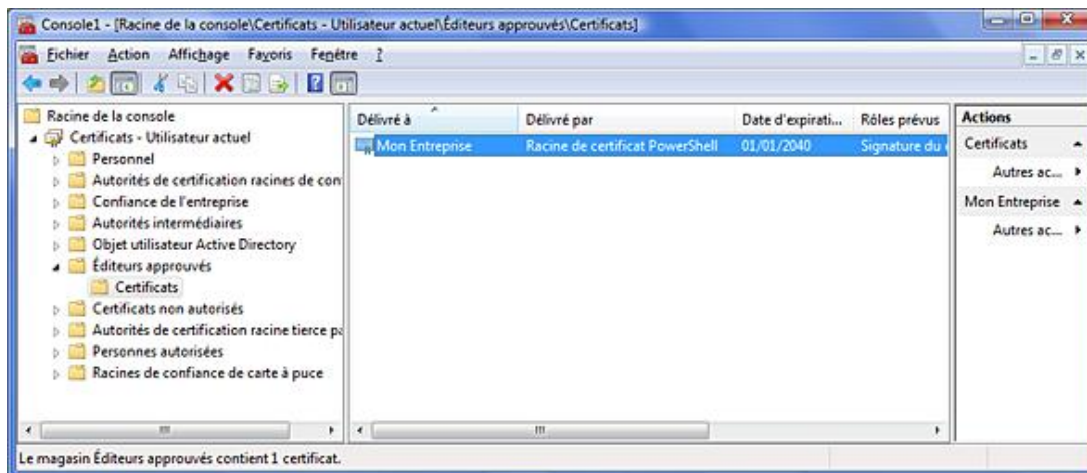
Car pour exécuter un script en mode **AllSigned**, il ne suffit pas que le script soit signé, il faut aussi que vous soyez en possession du certificat racine ET que l'éditeur de ce script soit approuvé. Si vous êtes déjà en possession ou que vous avez importé le certificat racine (voir comment importer un certificat dans la partie traitant du déploiement de certificats), alors il vous faut maintenant approuver l'éditeur. Sous peine de voir PowerShell afficher ceci :

```
PS > .\MonScript.ps1
```


Voulez-vous exécuter le logiciel de cet éditeur non approuvé ?
Le fichier C:\Temp\MonScript.ps1 est publié par CN=Mon Entreprise 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 [?]

En répondant au message suivant par **[T] Toujours exécuter** alors l'éditeur du script devient éditeur approuvé.

- Pour consulter la liste des éditeurs approuvés, choisissez dans la console de management le magasin **Editeurs approuvés** puis dans l'arborescence cliquez sur **Certificats**.



Attention à bien vous mettre en mode **AllSigned** pour vérifier l'exécution de vos scripts signés.

5. Déployer vos certificats

Comme vous venez de le voir, l'exécution d'un script signé nécessite deux choses :

- Être en possession d'un certificat d'autorité de certification racine.
- Et que son éditeur soit approuvé.

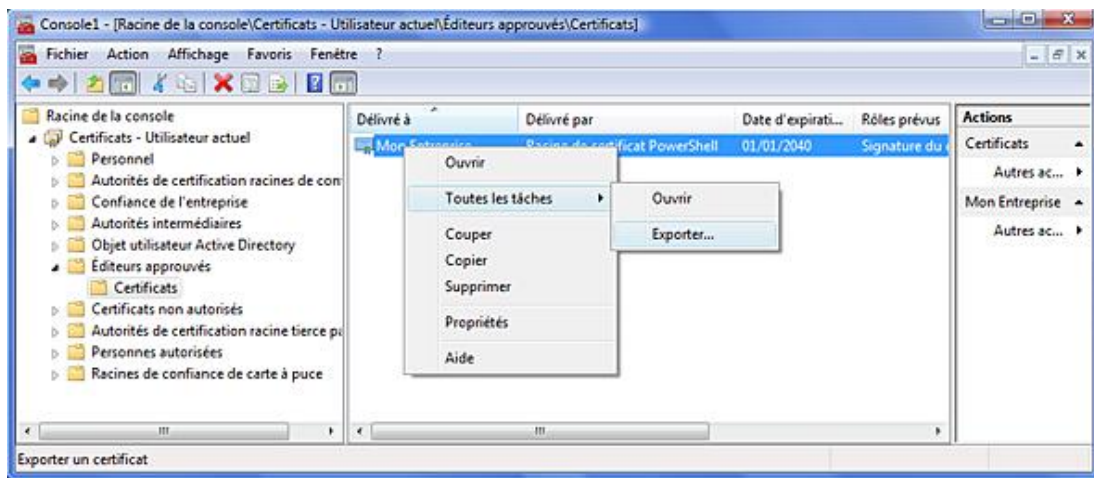
Or, si vous avez opté pour une politique de sécurité qui consiste à choisir le mode **AllSigned** sur tous les postes informatique, vous allez devoir :

- Déployer le certificat d'autorité de certification racine.
- Approuver l'éditeur (c'est-à-dire vous-même) de vos scripts via la console PowerShell en choisissant l'option **[T]**, soit déployer le certificat se trouvant dans le magasin **Éditeurs approuvés**.

Bien entendu, si vous disposez de quelques machines, cela ne pose pas de réel problème. Mais dans le cas où vous prévoyez de déployer des scripts PowerShell sur un parc informatique de moyenne ou grande taille, c'est une autre paire de manches.

Le déploiement de certificats se fait en deux temps. Tout d'abord, l'exportation du certificat depuis votre ordinateur, puis son importation sur tous les postes exécutant vos scripts signés.

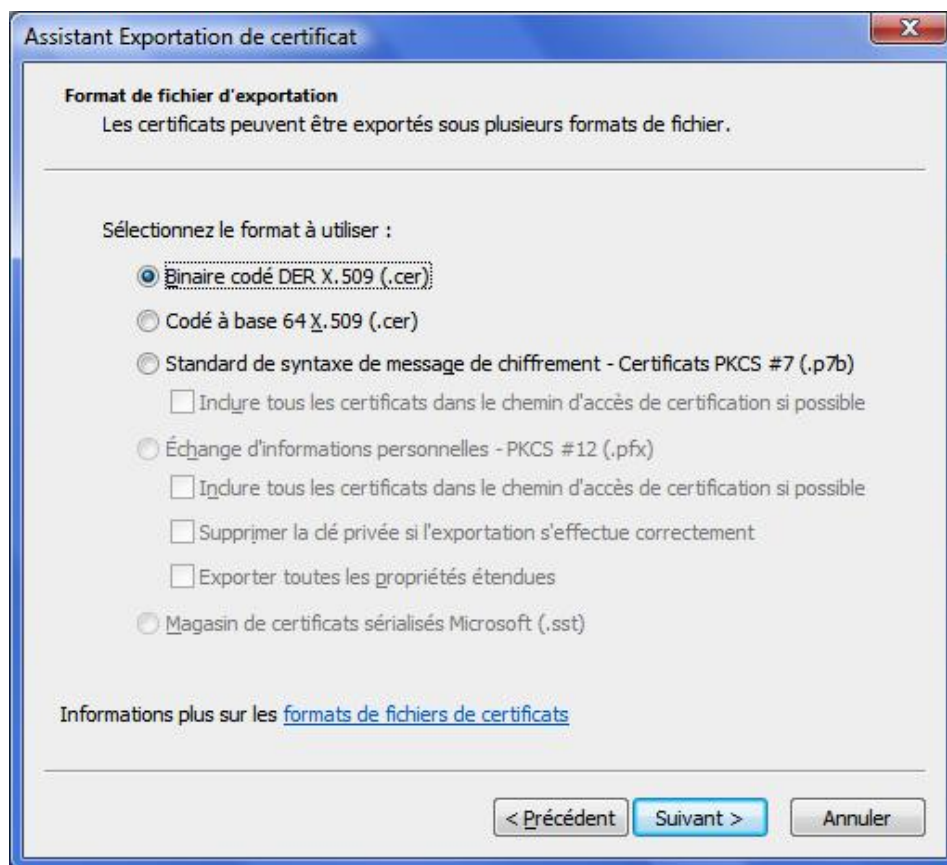
- Pour exporter le certificat, sélectionnez-le avec le clic droit de votre souris, puis choisissez **Toutes les tâches et Exporter**.



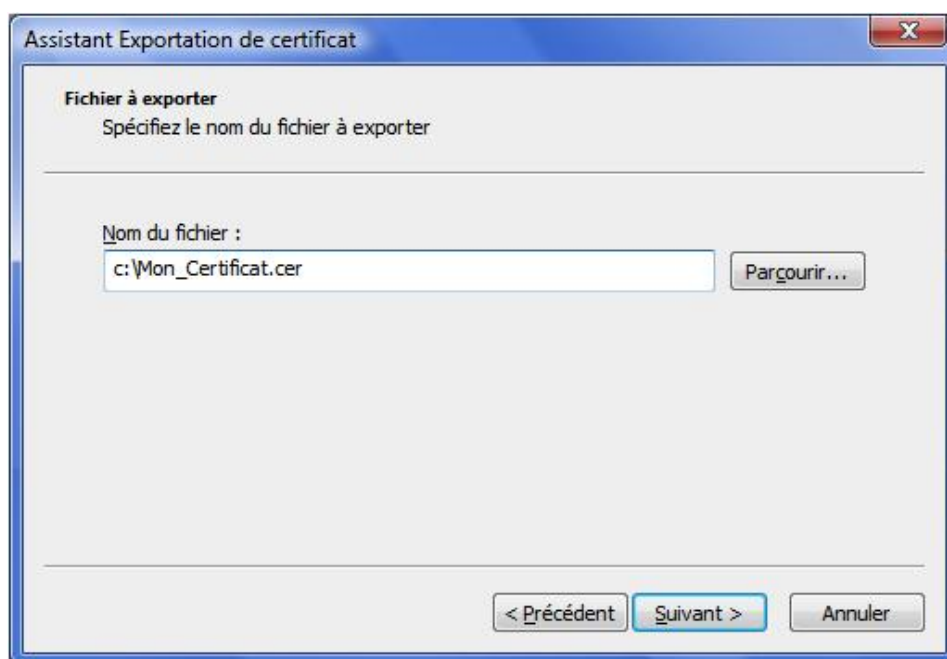
- L'assistant d'exportation se lance, cliquez sur **Suivant**.



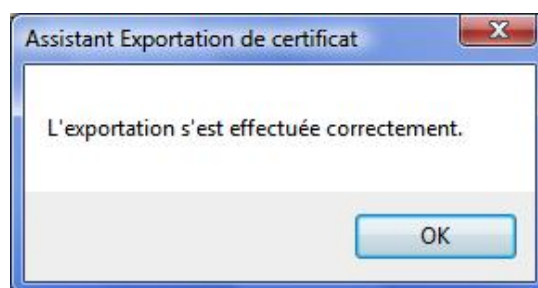
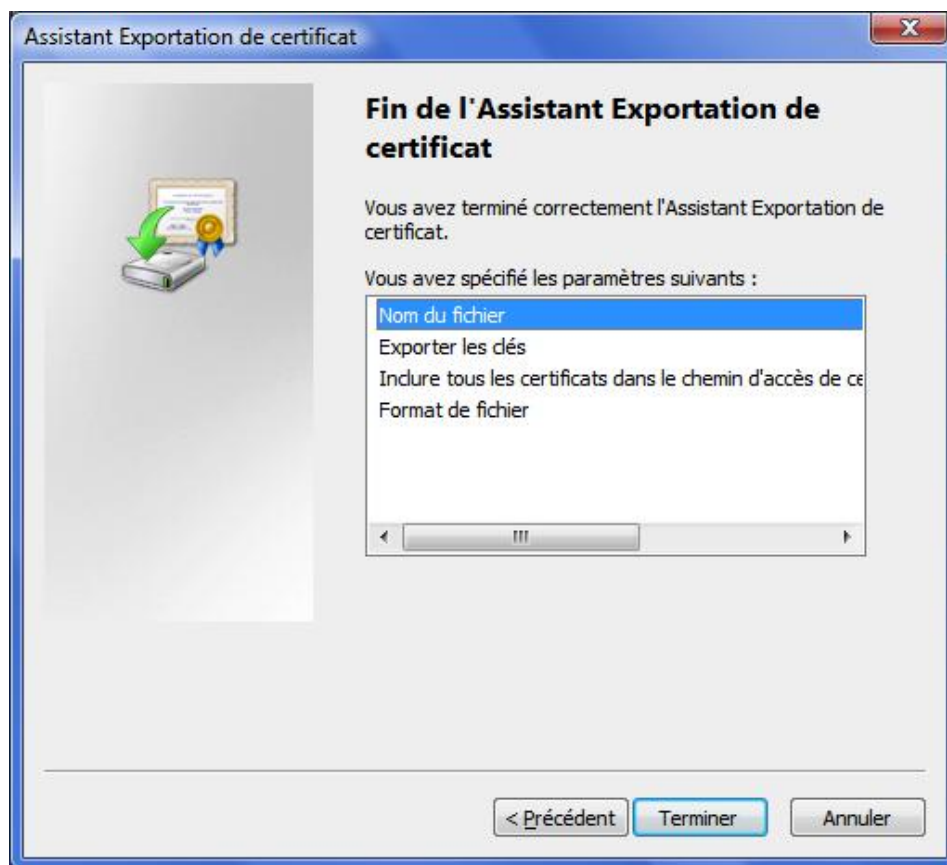
- Choisissez le format d'exportation, et cliquez sur **Suivant**.



- Donnez-lui un nom, et cliquez sur **Suivant**.



- Pour finir, cliquez sur **Terminer**.

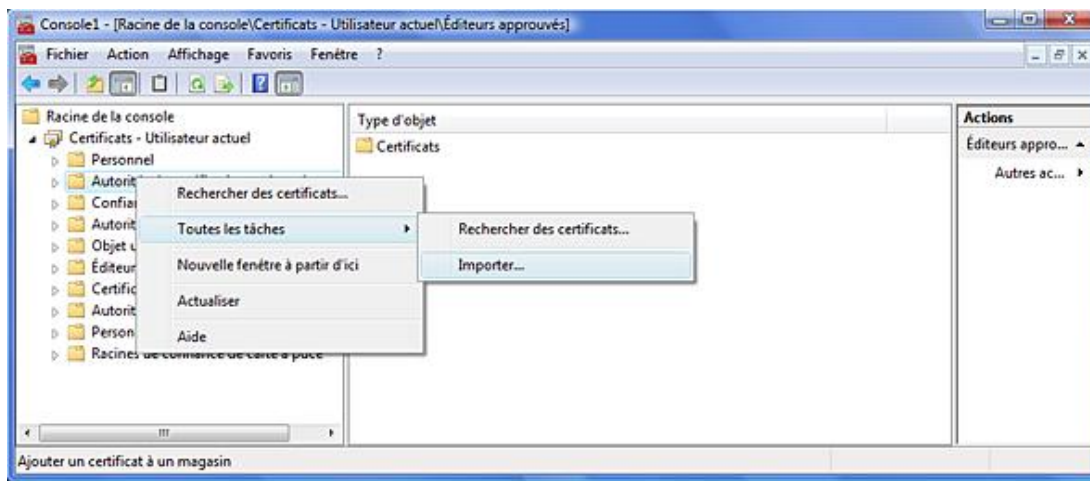


Nous venons d'exporter un certificat. Il ne reste plus qu'à réaliser l'importation sur les postes de travail cibles. Pour cela deux solutions : l'importation manuelle, ou l'importation par stratégie de groupe (GPO) depuis Active Directory (cette solution nécessite que vos postes de travail soient membres d'un domaine).

a. Importation manuelle

L'importation manuelle n'est ni plus ni moins que l'opération inverse de l'exportation.

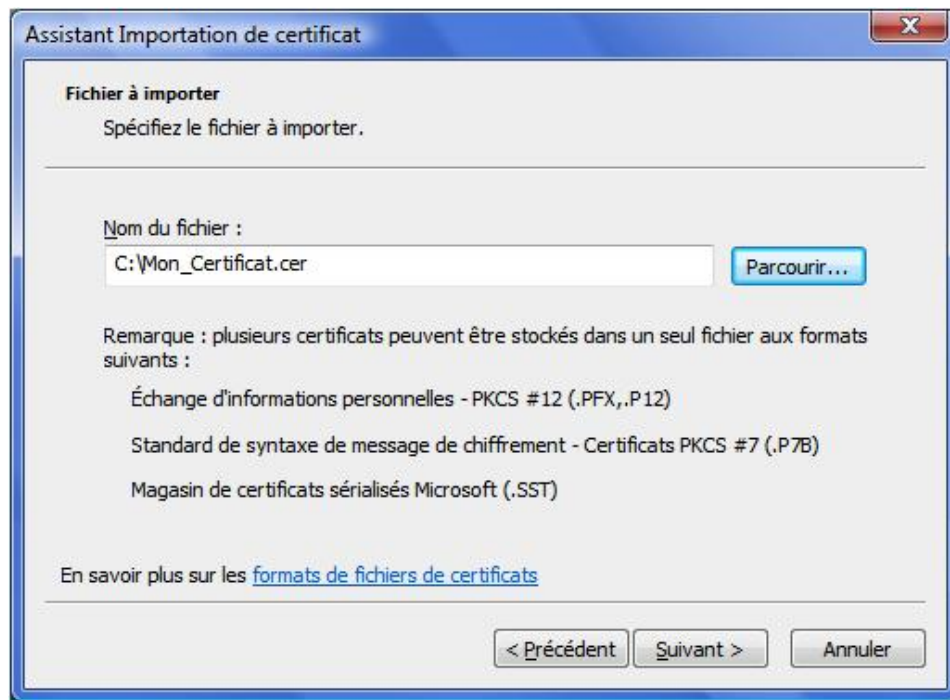
- Pour importer un certificat, sélectionnez avec le clic droit **Certificats** dans le magasin choisi, puis cliquez sur **Toutes les Tâches** puis **Importer**.



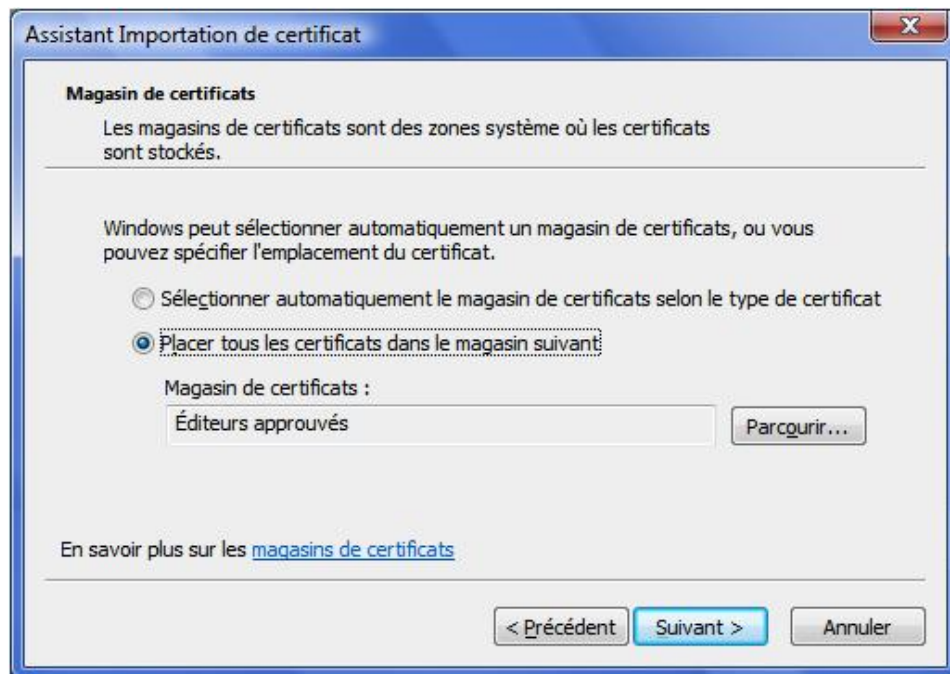
- C'est l'assistant d'importation qui se lance cette fois-ci, cliquez sur **Suivant**.



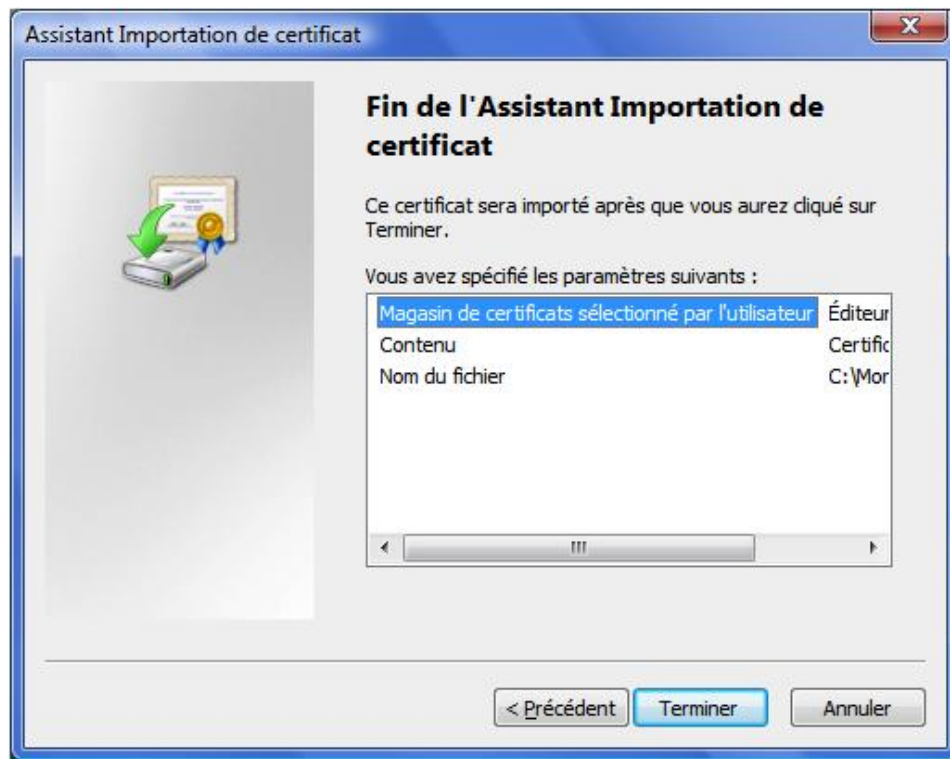
- Cliquez sur **Parcourir** pour atteindre votre certificat, puis cliquez sur **Suivant**.



- Renseignez le magasin dans lequel vous voulez voir votre certificat publié, choisissez le magasin **Éditeurs approuvés** et cliquez sur **Suivant**.



- Vérifiez les informations dans la fenêtre de fin de l'assistant, puis cliquez sur **Terminer**.

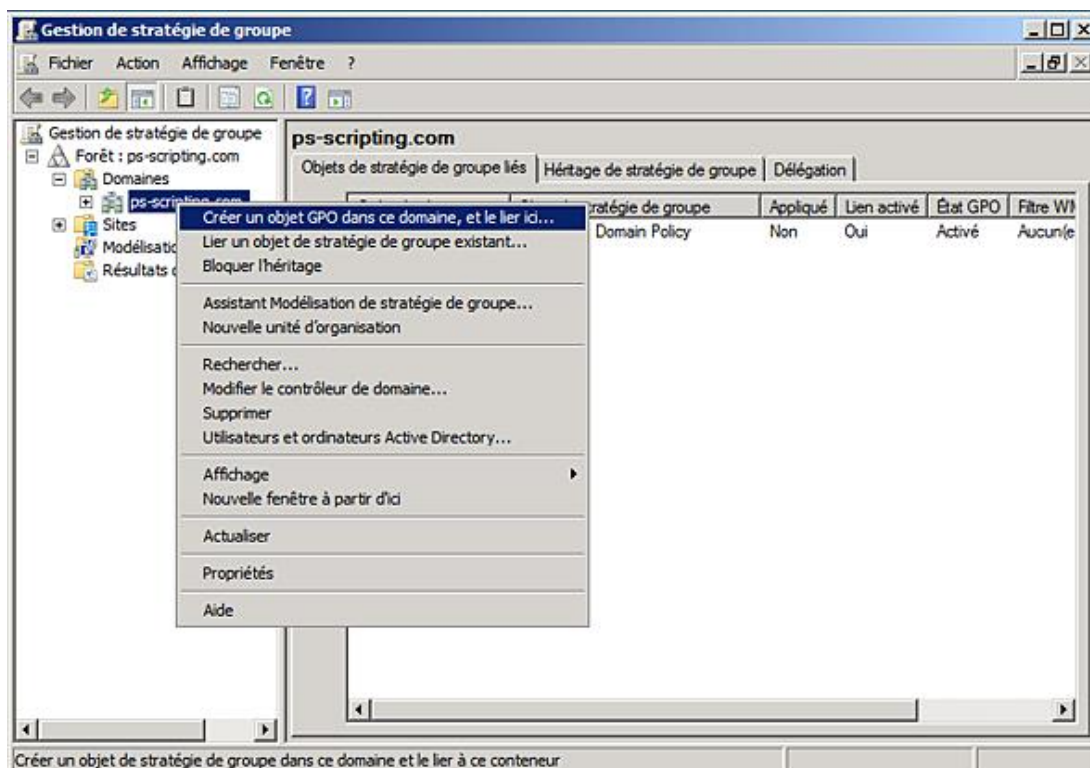


Votre certificat est maintenant disponible dans le magasin correspondant.

b. Importation par GPO

L'importation par GPO est la solution la moins contraignante lorsque vous disposez d'un parc de machines conséquent, ou géographiquement distant. Cependant, l'importation par GPO nécessite que chaque machine soit membre d'un domaine appartenant au même domaine Active Directory.

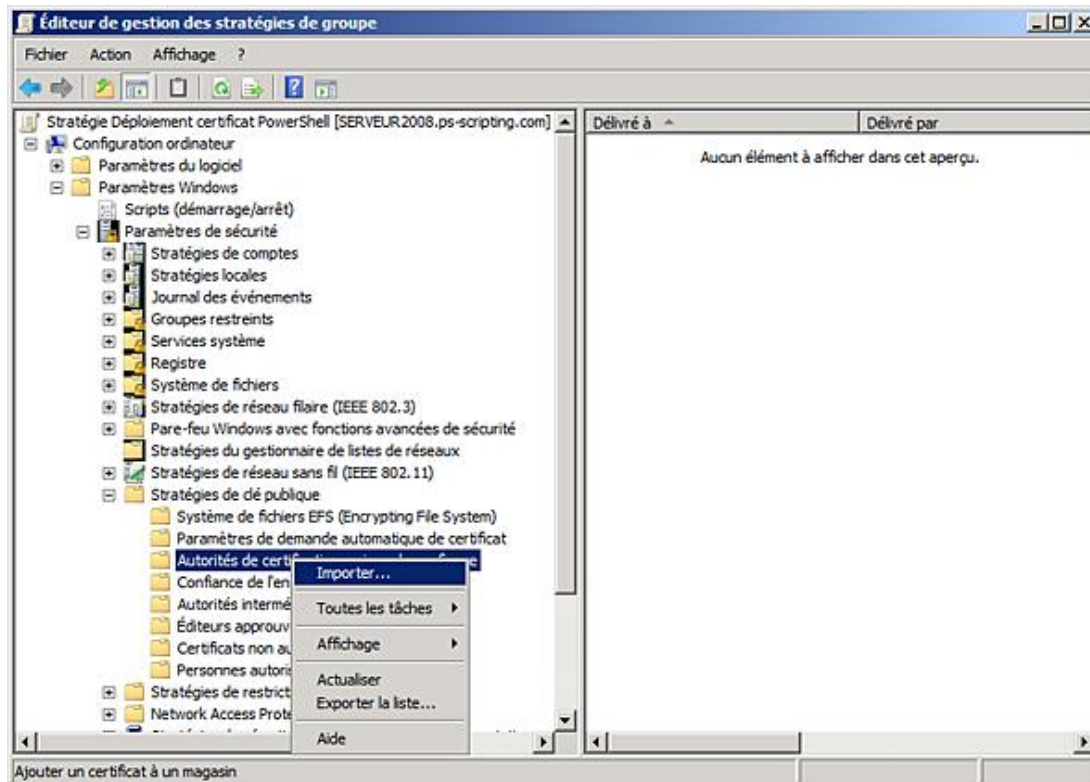
- Pour importer un certificat par GPO sous Windows Server 2008, ouvrez la console **Gestion de stratégie de groupe**. Puis, dans la console, sélectionnez le domaine avec un clic droit et choisissez **Créer un objet GPO dans ce domaine, et le lier ici...**



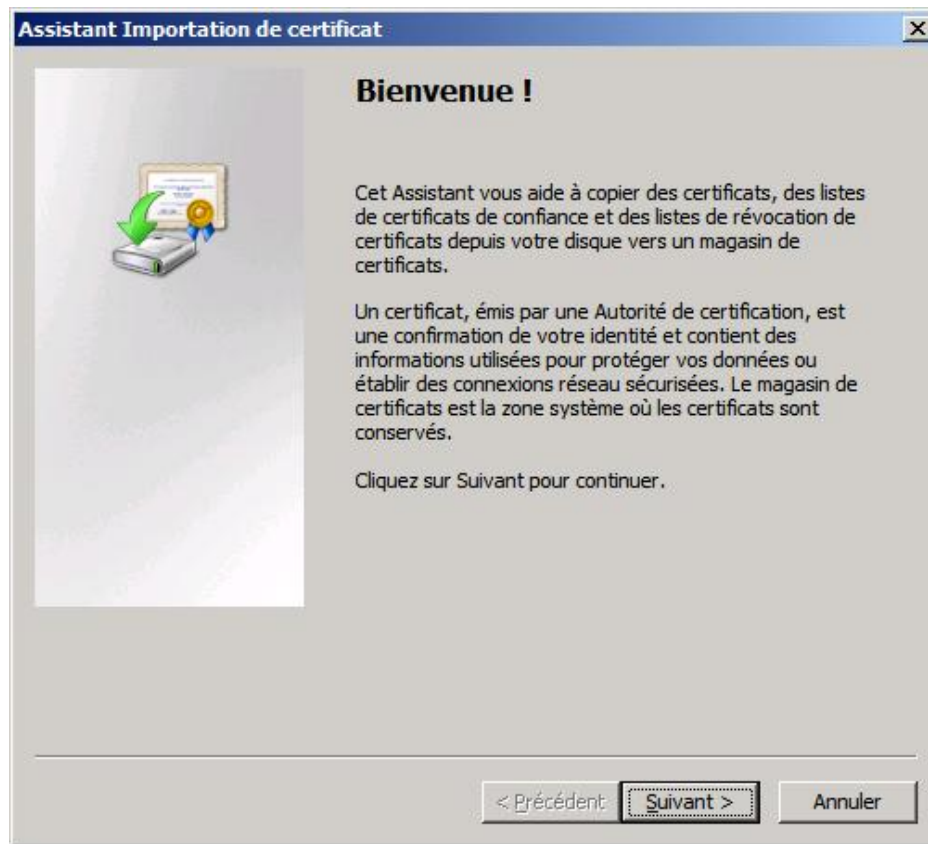
- Donnez un nom à la stratégie de groupe, par exemple **Déploiement certificat PowerShell**.



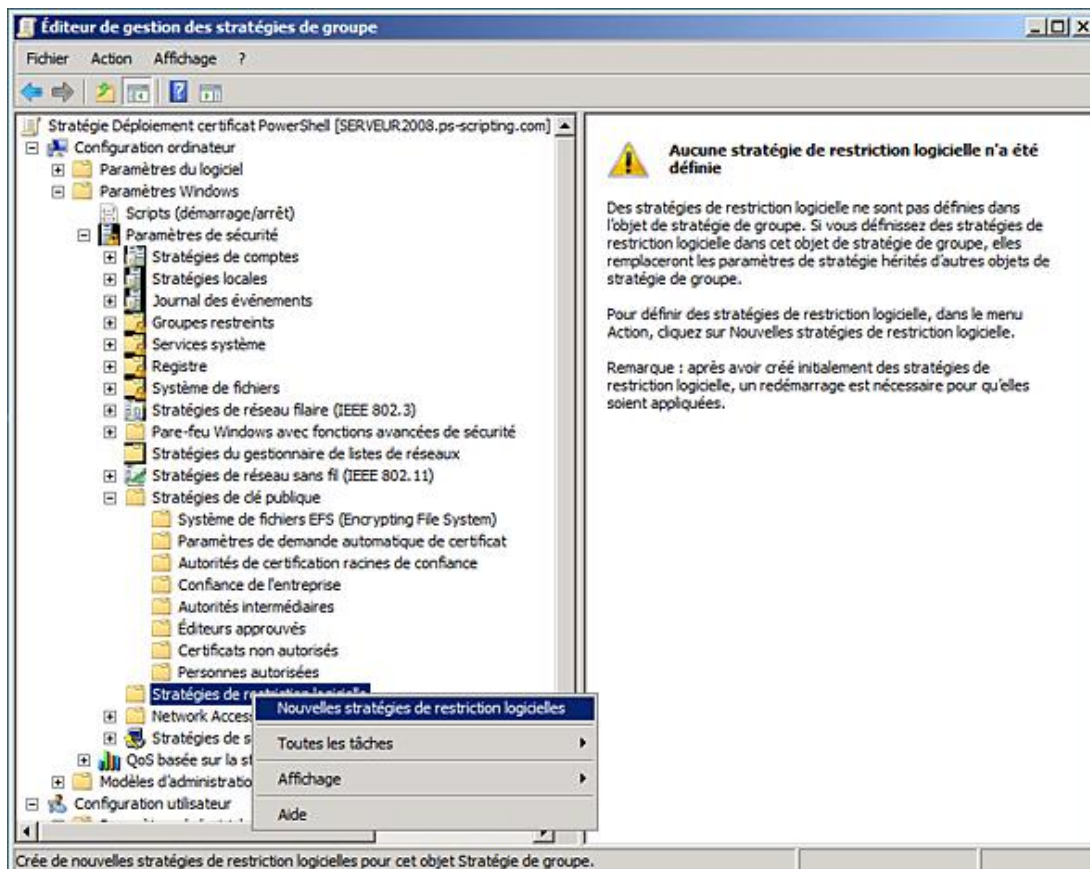
- Sélectionnez la stratégie ainsi créée, faites un clic droit et choisissez **Modifier**.
- La console de gestion des Stratégies de groupe s'affiche. Faites un clic droit sur **Configuration ordinateur - Paramètres de sécurité - Stratégie de clé publique - Autorités de certification racines de confiance**. Sélectionnez **Importer**.



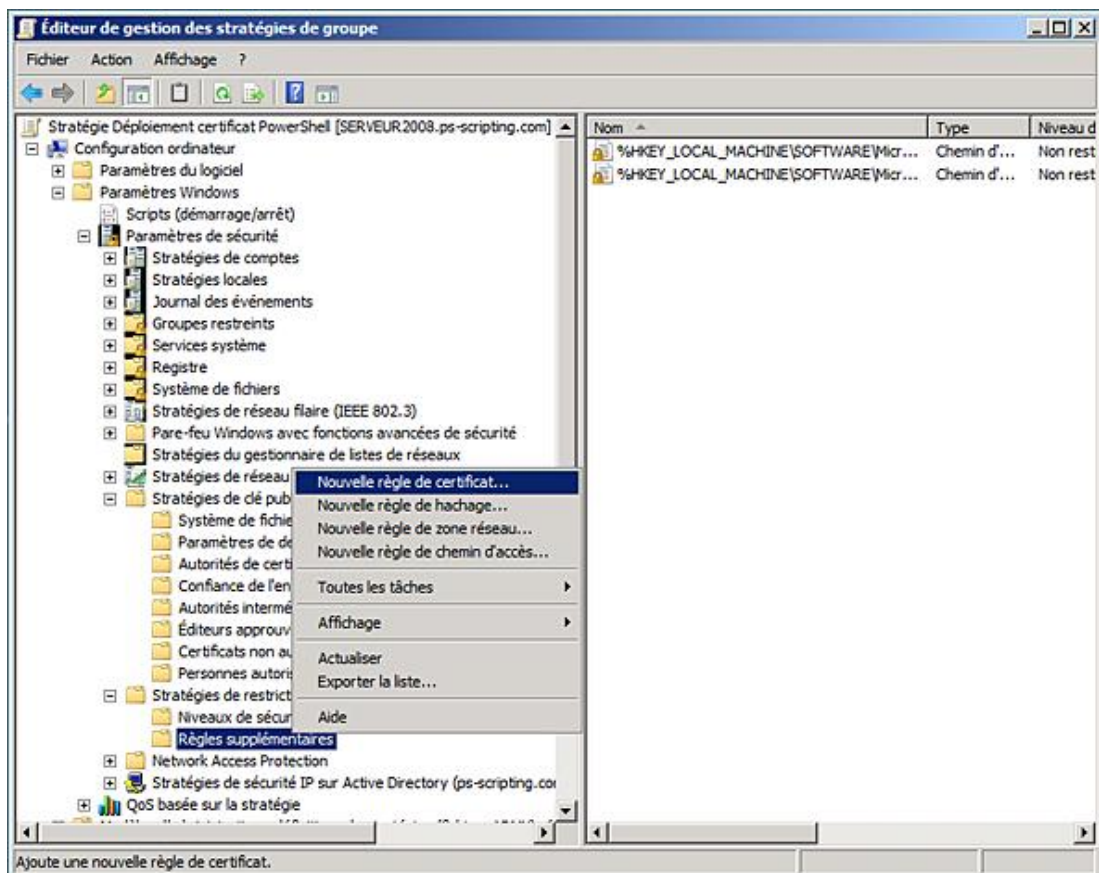
- L'assistant d'importation vous guide pour importer le certificat d'autorité racine.



- Une fois le certificat racine importé, il faut également permettre à tous les postes de travail d'approuver automatiquement l'éditeur de la signature. Et pour cela, faites un clic droit sur **Configuration de l'ordinateur - Paramètres Windows - Paramètres de sécurité - Stratégie de restriction logicielle**. Puis sélectionnez **Nouvelles stratégies de restriction logicielle**.

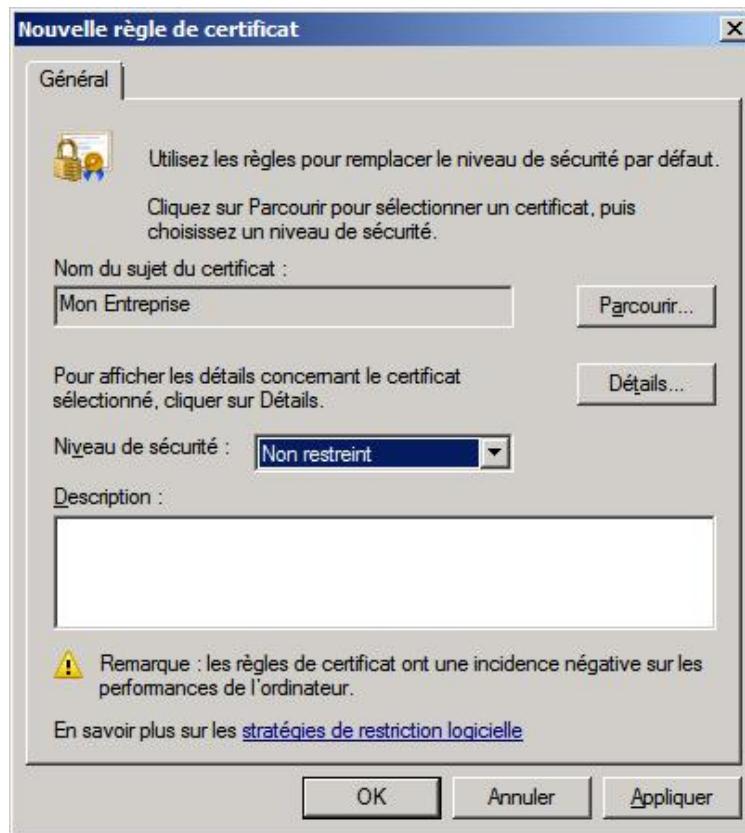


Deux nouveaux sous-répertoires apparaissent. Faites un clic droit sur **Règles supplémentaires** et sélectionnez **Nouvelle règle de certificat**.



- Dans l'éditeur de la nouvelle règle, sélectionnez **Parcourir** pour atteindre votre certificat. Puis choisissez le niveau de sécurité **Non restreint**. Cliquez sur **Appliquer** puis **OK**.

L'application du niveau **Non restreint** aura pour effet de déterminer les droits d'accès au logiciel selon les droits d'accès de l'utilisateur.



- Répondez **Oui** à la question qui vous est posée afin d'activer les règles de certificats.

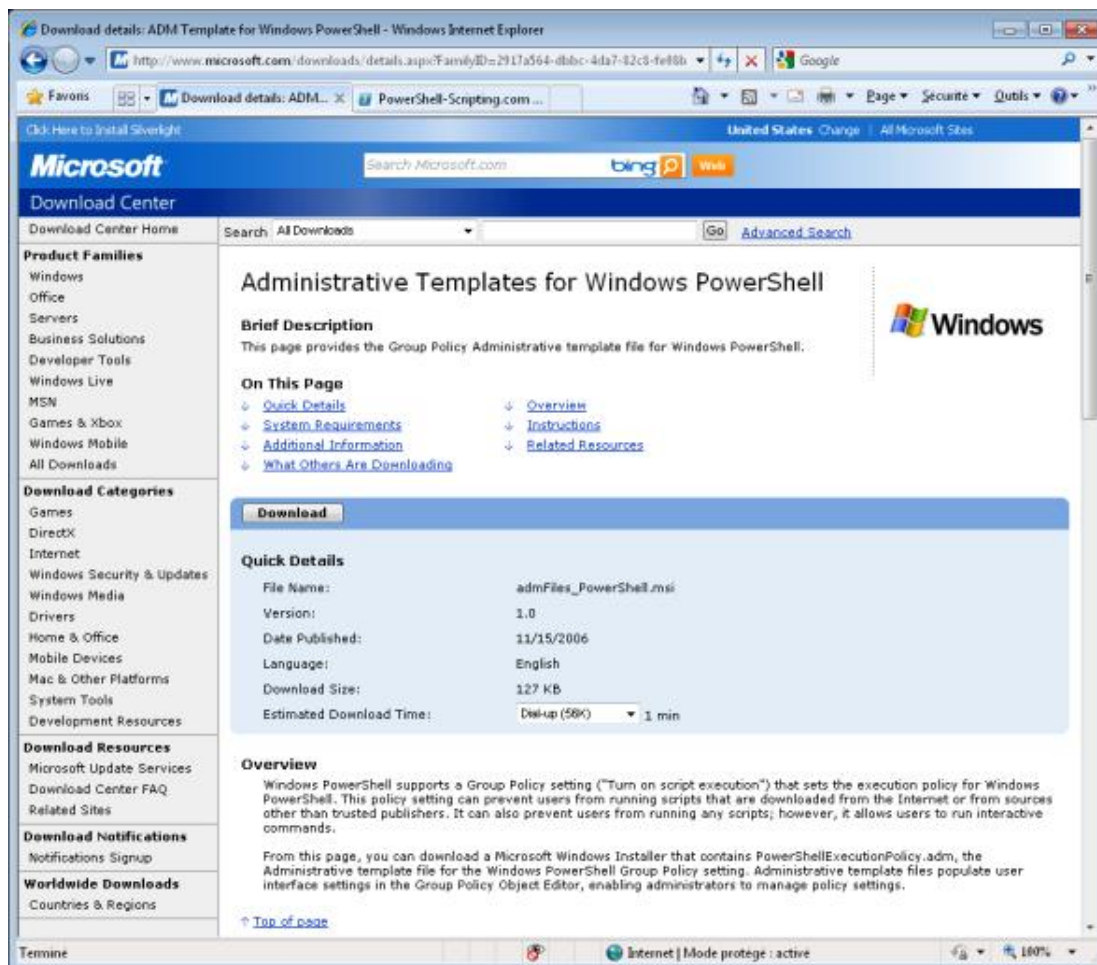
Vos certificats seront maintenant déployés correctement sans que vous ayez à faire une quelconque opération sur les postes de travail, si ce n'est que de les redémarrer pour qu'ils puissent prendre en compte la nouvelle stratégie de groupe.

Gérer les stratégies d'exécution de PowerShell via les stratégies de groupe

Maîtriser la stratégie d'exécution PowerShell au sein d'un groupe d'ordinateurs n'est pas chose aisée, surtout lorsqu'il s'agit d'un parc de plusieurs dizaines de postes informatiques. C'est pour cela, que Microsoft distribue un modèle d'administration (fichier ADM, *Administration Model*) de façon à pouvoir appliquer la stratégie d'exécution via les stratégies de groupe (ou GPO en anglais, *Group Policy Object*). Pour rappel, les GPO permettent la gestion des ordinateurs et des utilisateurs dans un environnement Active Directory.

1. Installation du fichier ADM

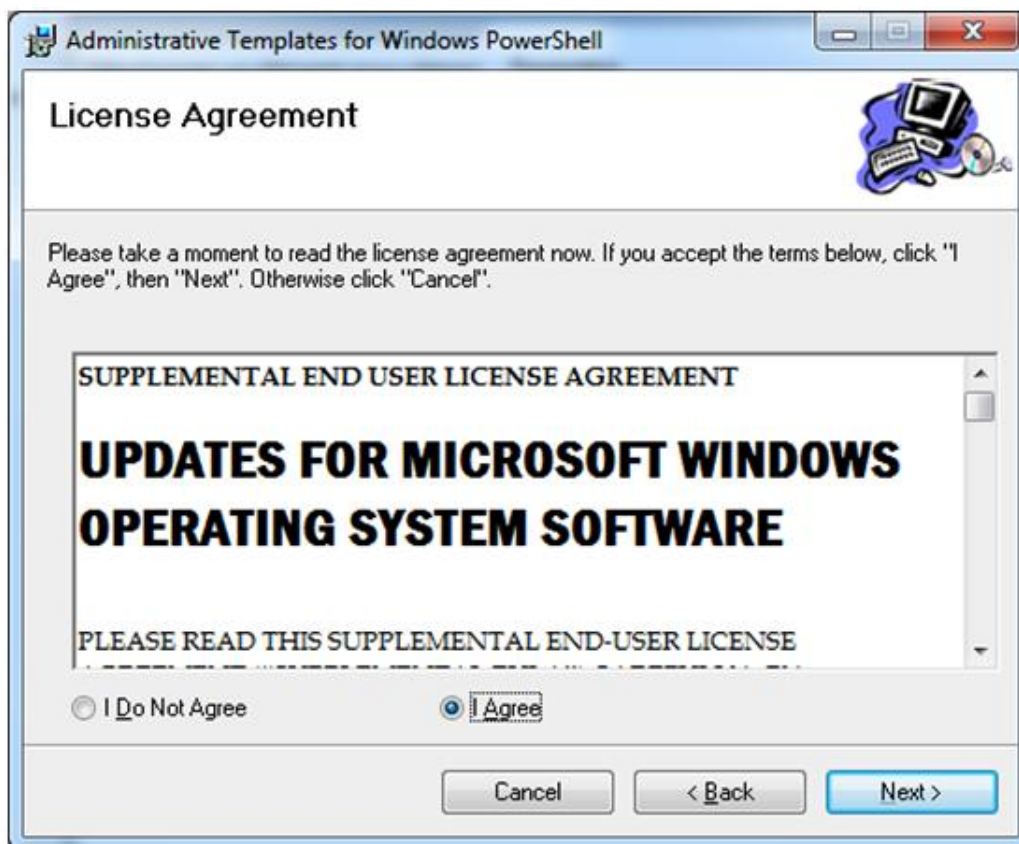
Comme l'ensemble des modèles d'administration pour les produits Microsoft, le fichier ADM est téléchargeable sur le site de l'éditeur sous le nom de « Administrative Templates for Windows PowerShell ».



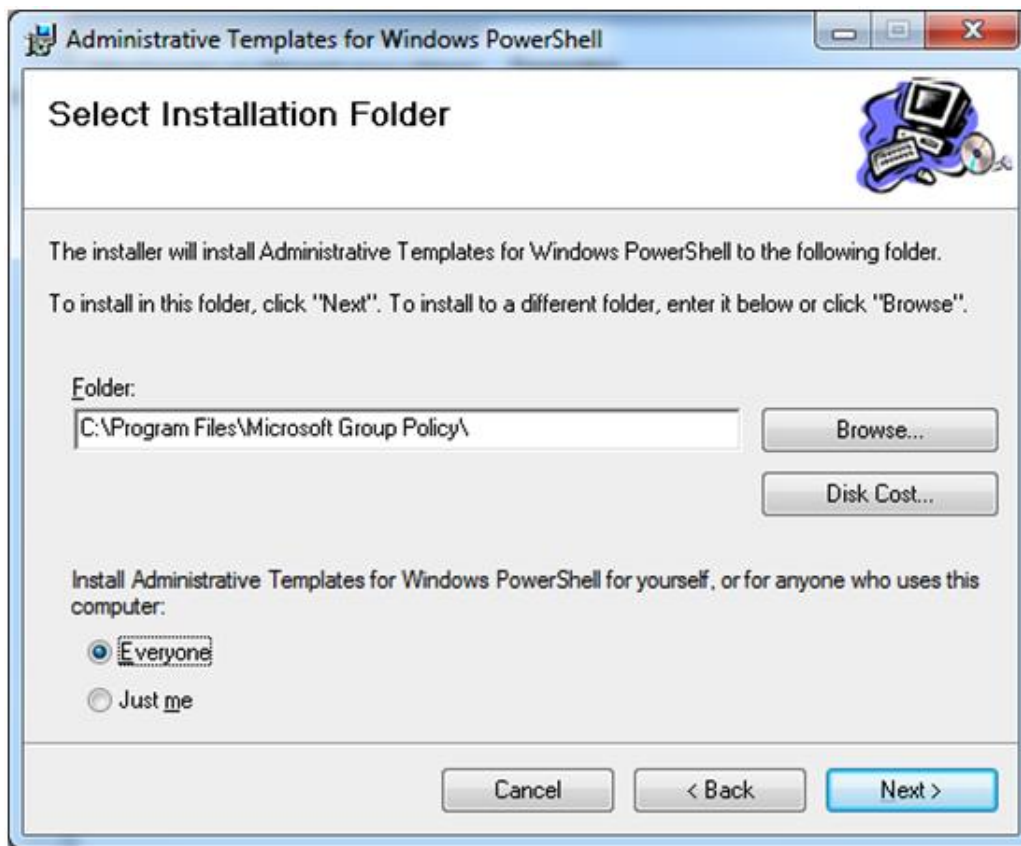
- Une fois téléchargé, cliquez sur **exécuter**, puis laissez-vous guider par le guide d'installation.



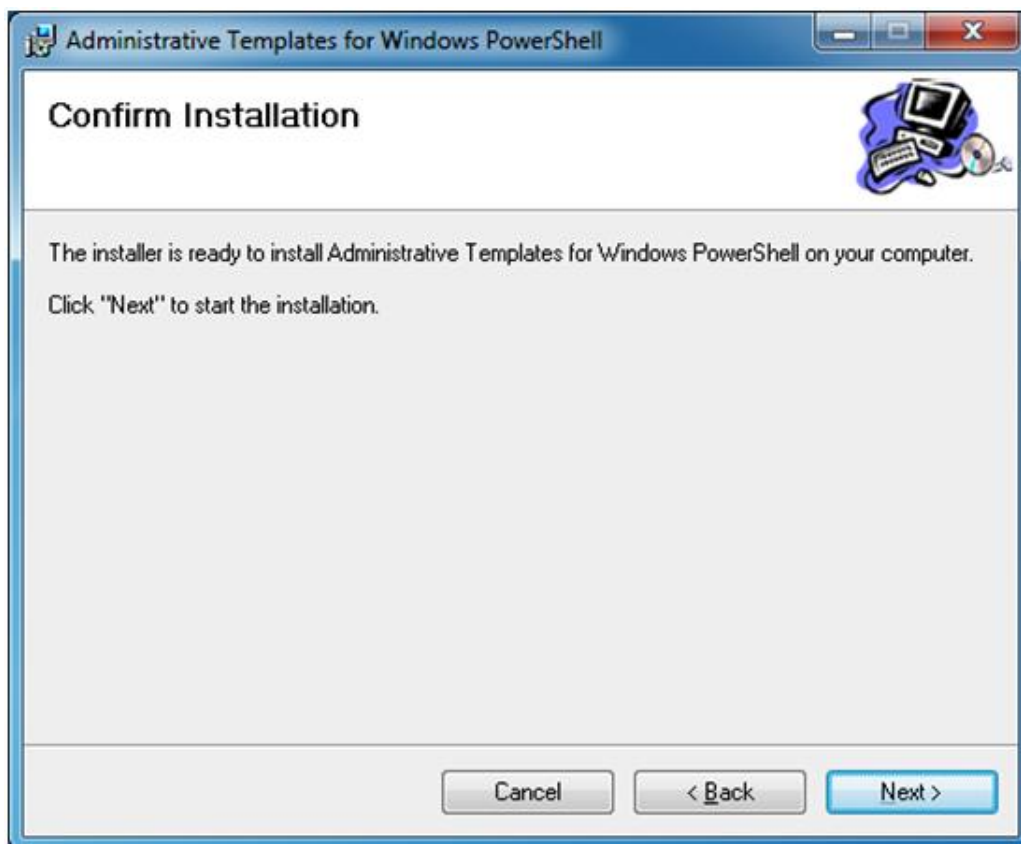
- Choisissez d'accepter la licence, puis cliquez sur **Next**.

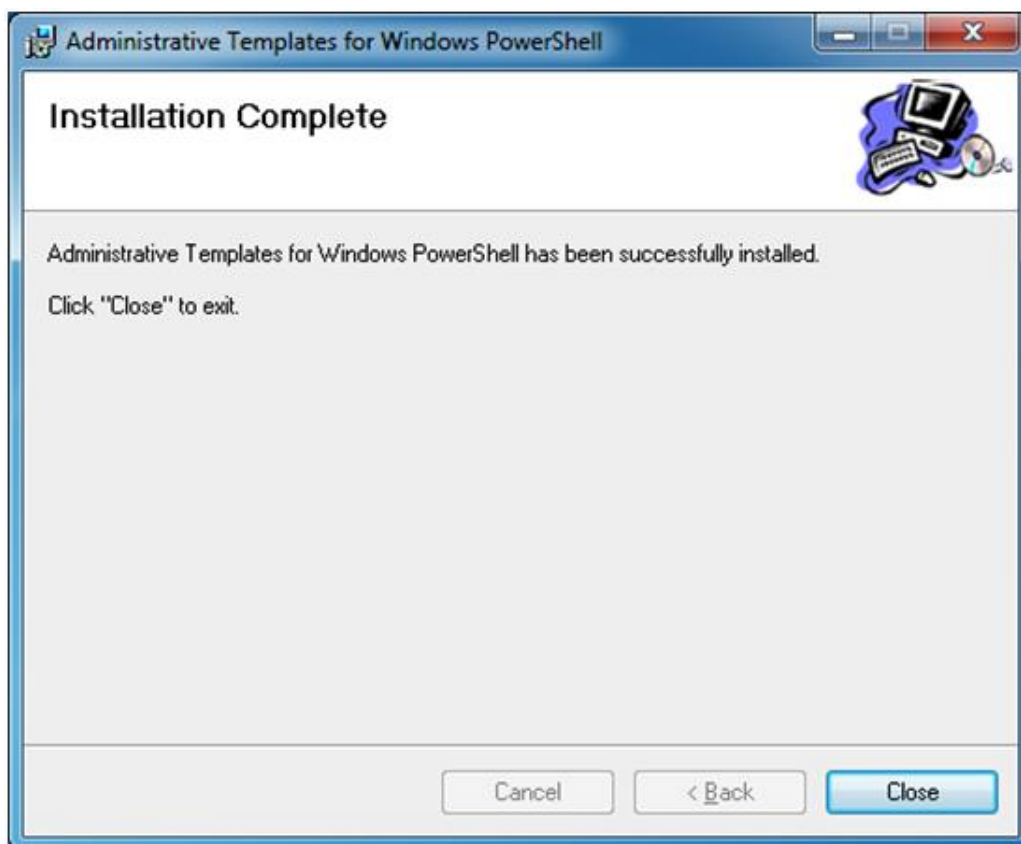
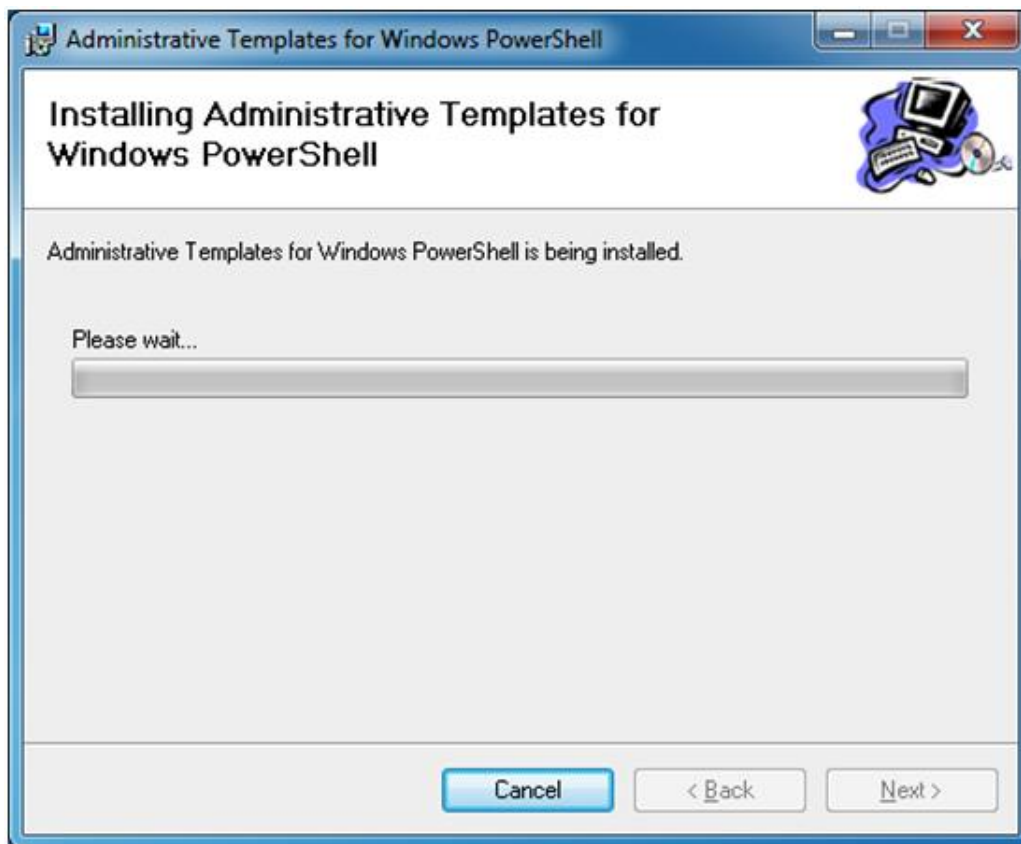


- Sélectionnez un répertoire d'installation pour le modèle.



- Terminez ensuite l'installation.





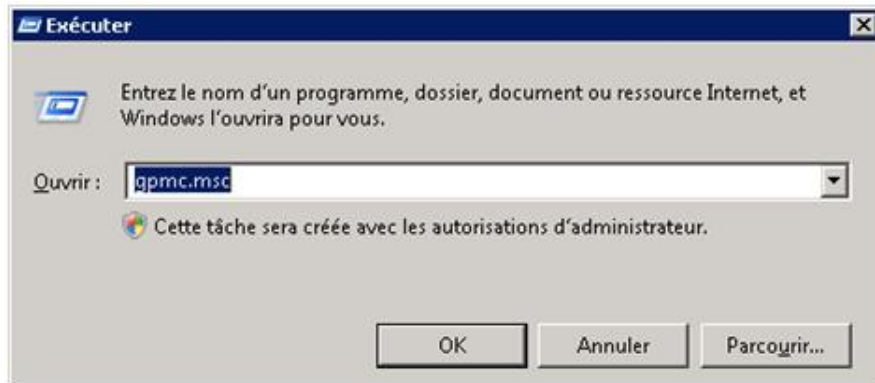
Le modèle d'administration est alors disponible sous forme d'un fichier .adm (PowerShellExecutionPolicy.adm) dans le répertoire d'installation défini plus haut.

2. Application de la stratégie d'exécution

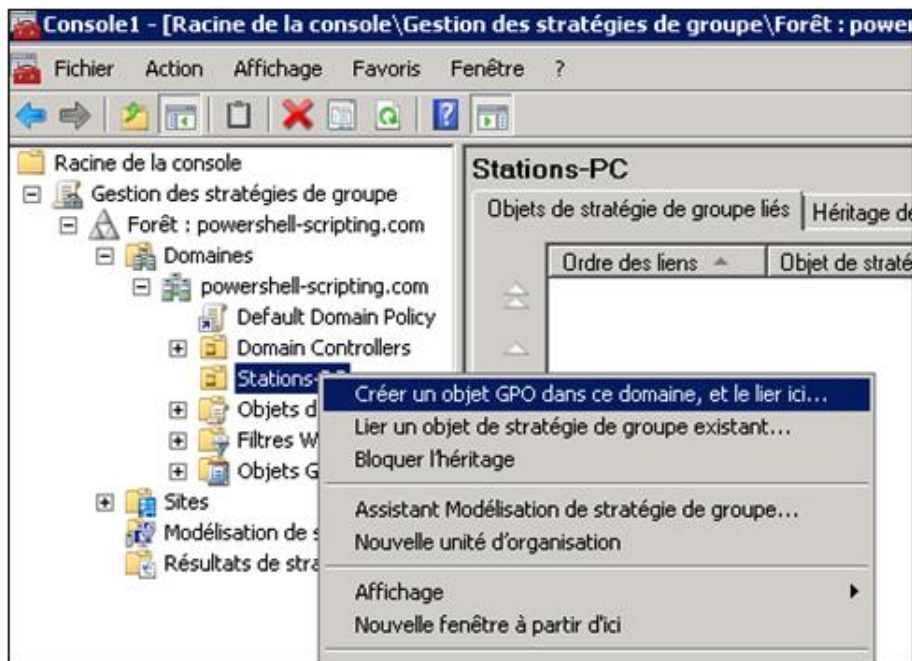
Lorsque le modèle d'administration est installé. La création et l'application de GPO peut être réalisée de différentes façons selon la version de Windows Server utilisée. Dans cet ouvrage, la gestion des stratégies de groupe est réalisée à travers la console GPMC (*Group Policy Management Console*) sur un serveur Windows 2008 R2.

- Bien que la Console de gestion des stratégies de groupe (GPMC.msc) soit fournie avec Windows Server 2008 R2, vous devez installer la Gestion des stratégies de groupe en tant que fonctionnalité via le Gestionnaire de serveur.

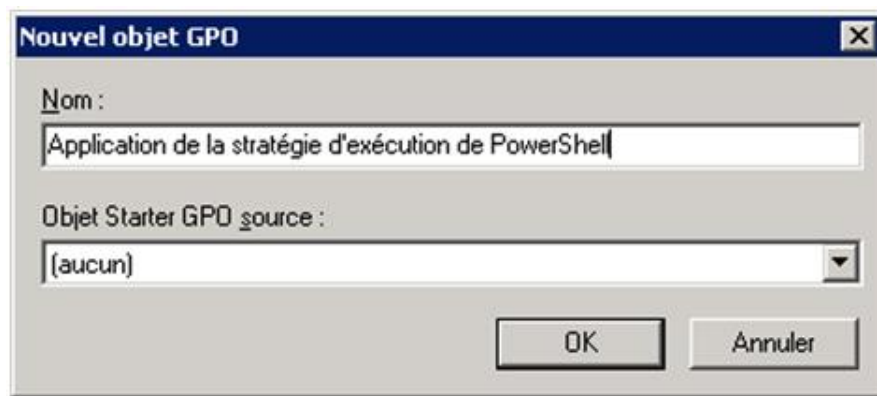
- Cliquez sur **Exécuter** puis saisissez **GPMC.msc**.



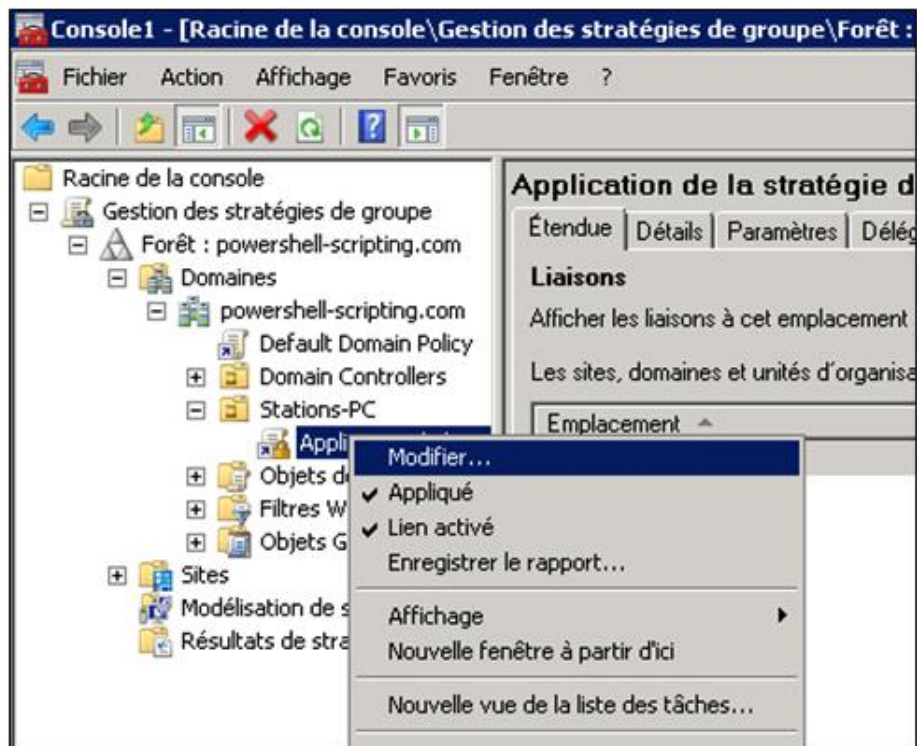
- Sélectionnez l'UO (Unité d'Organisation) souhaitée, faites un clic droit pour sélectionner **Créer un objet GPO dans ce domaine, et le lier ici...**



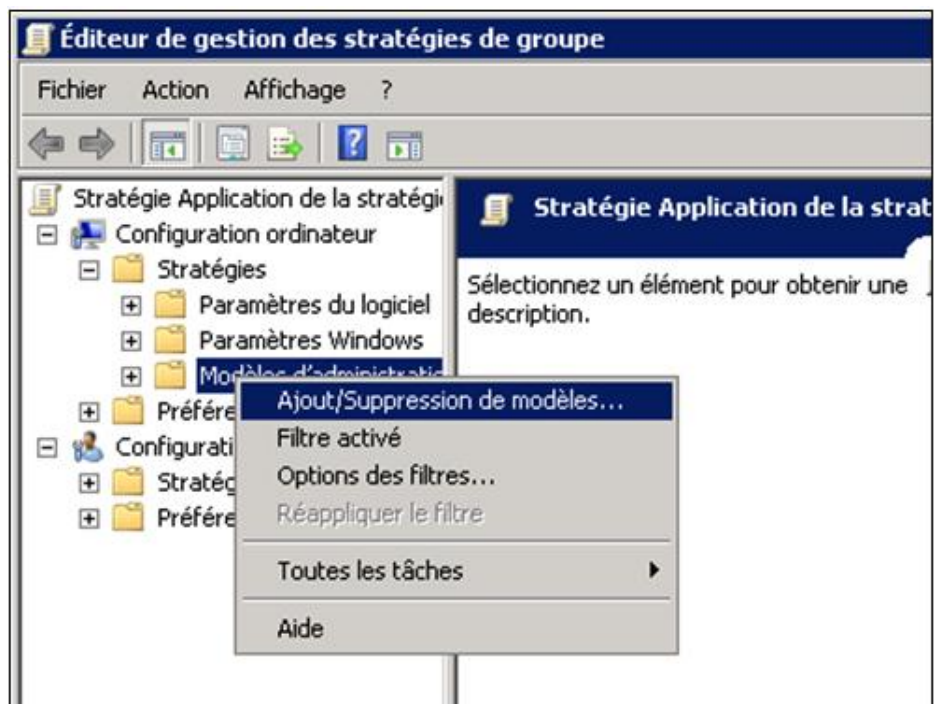
- Donnez un nom à la nouvelle GPO.



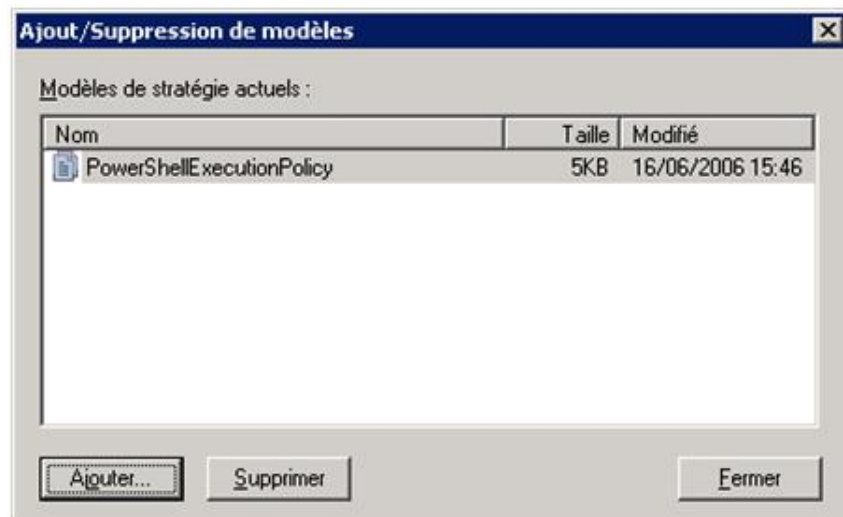
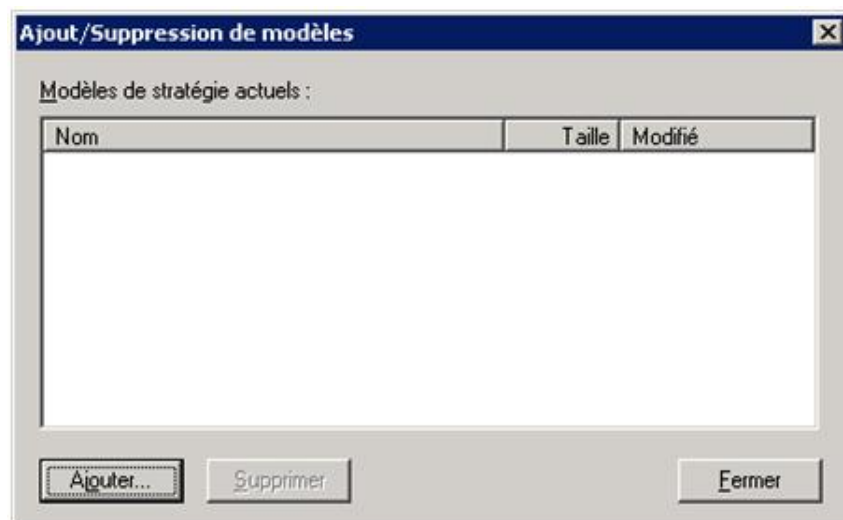
- Faites un clic droit, puis **Modifier** sur la GPO fraîchement créée.



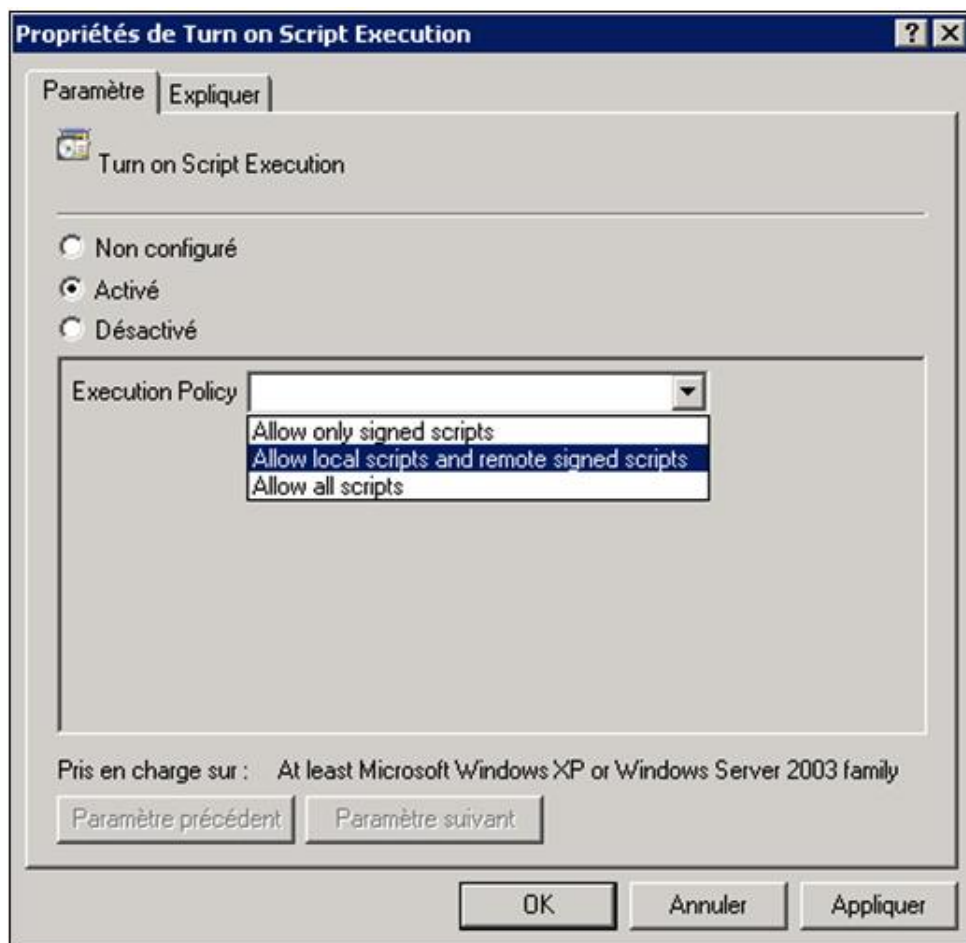
- La fenêtre d'édition de la stratégie s'ouvre. Faites alors un clic droit sur **Modèles d'administration** (Sous l'arborescence Configuration ordinateur - Stratégies), puis sélectionnez **Ajout/Suppression de modèles**.



- Cliquez sur **Ajouter**, puis sélectionnez le fichier **PowerShellExecutionPolicy.adm**.



- Double cliquez sur le paramètre **Turn on Script Execution** maintenant disponible sous l'arborescence **Configuration de l'ordinateur/Modèles d'administration/Modèles d'administration classiques (ADM)/Windows PowerShell**. Puis choisissez la stratégie d'exécution souhaitée.



- Enfin, fermez l'éditeur de GPO.

La stratégie d'exécution est à présent déployée sur toutes les machines appartenant à l'unité d'organisation souhaitée.



Pour appliquer immédiatement une GPO, exécutez la commande `Gpupdate /force`.

Introduction au .NET

Dans les chapitres précédents, nous avons quelques fois fait appel aux objets du Framework .NET sans vraiment avoir pris la peine d'expliquer leur nature et la façon de les utiliser.

Dans cette partie, nous allons donc vous expliquer pas à pas ce qu'est le Framework, ce qu'il contient, comment rechercher des objets .NET qui sont susceptibles de nous intéresser, comment les créer, et comment lister leurs membres.

Mais avant cela revenons sur le « pourquoi de l'utilisation des objets .NET ».

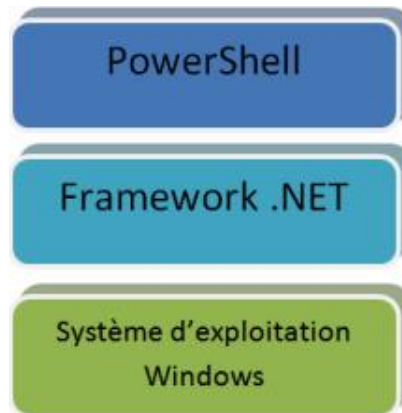
Si vous avez vous-même installé PowerShell, vous n'êtes pas sans savoir que ce dernier nécessite au préalable l'installation du Framework .NET. Et ce pour la raison toute simple que les concepteurs de PowerShell se sont appuyés sur les classes du Framework pour développer des outils en ligne de commande, plus communément appelées commandelettes.

Selon l'aveu des concepteurs eux-mêmes, il était prévu d'intégrer dans Windows PowerShell de nombreuses commandelettes pour pouvoir exploiter au maximum le système Windows. Mais le Framework .NET étant tellement vaste que les choses ne se sont pas exactement passées comme prévu. Il s'est avéré que l'ajout de nouvelles commandes prenait un temps trop important, et que par conséquent, pour ne pas retarder la sortie de PowerShell, ils ont finalement préféré faciliter l'accès aux classes du Framework, permettant ainsi de couvrir l'ensemble des besoins des utilisateurs.

Dans ce chapitre, nous parlerons indifféremment de classe .NET et de type .NET, qui désignent à peu près (pour ne pas dire exactement) la même chose.

Le Framework .NET

Connu de nombreux développeurs, le Framework .NET est un composant Windows apparu en version finale pour la première fois en 2002. Indispensable à l'installation de PowerShell, le Framework est désormais installé nativement sous Windows Vista, Windows 7, Windows Server 2008 R2 et disponible sous forme d'un composant additionnel pour Windows XP, Windows Server 2003 et Windows Server 2008. Destiné à faciliter le développement des applications informatiques, le Framework fournit une immense bibliothèque de classes, sur laquelle s'appuient certains langages comme le C#, le VB.NET, le J#, et bien évidemment PowerShell.



Architecture logicielle



Composition du Framework .NET

Sans rentrer dans des détails trop techniques qui ne seraient pas vraiment utiles pour le reste de la compréhension, sachez seulement que le Framework est composé de deux éléments principaux :

- Le CLR (*Common Language Runtime*), environnement d'exécution compatible avec tout langage de programmation respectant le CLS (*Common Language Specification*).
- La bibliothèque de classes, qui contient tous les types que l'on peut trouver dans le Framework .NET. Chaque classe étant répertoriée dans un espace de noms.

Utiliser des objets .NET avec PowerShell

Depuis le début de cet ouvrage, à quelques exceptions près, nous avons manipulé de nombreux objets qui nous étaient directement accessibles sans vraiment nous soucier de leurs origines. Mais ce qu'il faut savoir, c'est que l'utilisation de PowerShell ne s'arrête pas à l'utilisation de ces uniques types. En effet, PowerShell offre aussi la possibilité de manipuler d'autres types d'objet définis dans la bibliothèque de classe du Framework, et c'est ce que nous allons voir dans cette partie.

Avant toute chose, ce qu'il faut savoir, c'est qu'avec l'environnement Framework .NET, tout a un type. Jusqu'à maintenant, sans vraiment porter attention, nous avons manipulé de nombreux objets qui possèdent chacun un type bien particulier défini dans la bibliothèque du Framework. Prenons par exemple le cas de l'objet retourné par la commande **Get-Date**.

```
PS > $Date=Get-Date
PS > $Date.GetType()
```

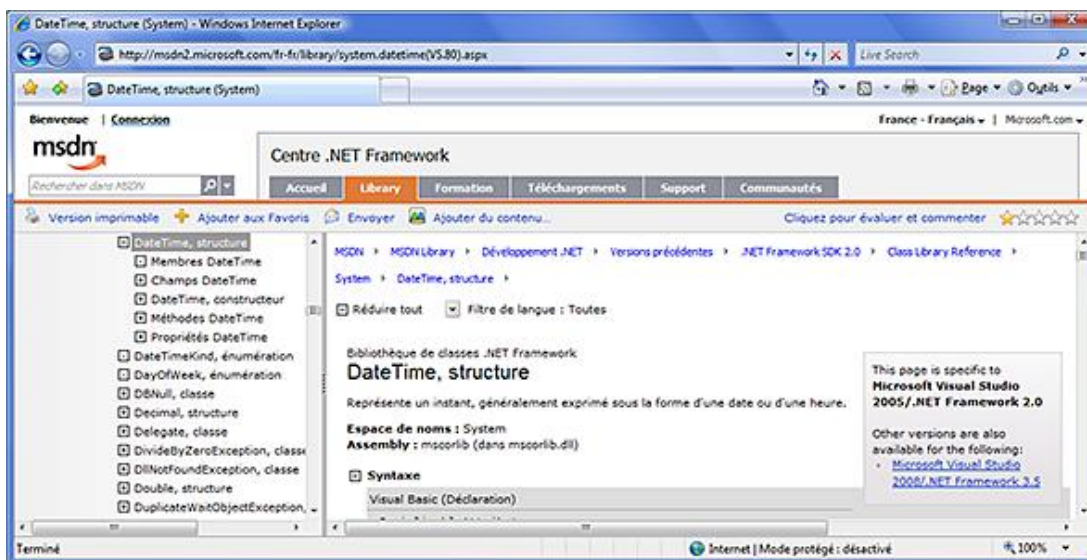
IsPublic	IsSerial	Name	BaseType
True	True	DateTime	System.ValueType

En appliquant la méthode `GetType` à cet objet, nous pouvons observer que le type utilisé est `DateTime` et que son espace de noms est « `System` ». Soit le nom complet : `System.DateTime`.



On appelle espace de noms (ou namespace en anglais), ce qui précède le nom d'une ou plusieurs classes .NET ou WMI. Ils sont utilisés dans le but d'organiser les objets et ainsi éviter de confondre des classes qui pourraient éventuellement porter le même nom.

Pour obtenir plus d'informations, sachez que tous les types (qu'il s'agisse de classes, de structures, d'énumérations, de délégués ou d'interfaces) définis dans la bibliothèque de classe Framework, sont détaillés sur le site de Microsoft MSDN.



En utilisant la méthode personnalisée `GetMsdnHelp` créée dans le chapitre Maîtrise du Shell, vous pourrez pour chaque objet, vous rendre directement sur la page du site MSDN en relation avec le type de votre objet.

À l'instar des types rencontrés jusque-là, chaque type .NET que vous rencontrerez possède un ou plusieurs membres qu'il est possible d'obtenir avec la commande **Get-Member**. Pour lister les méthodes et propriétés du type `DateTime` tapez simplement : `$date | Get-Member`.

```
PS > Get-date | Get-Member
```

```
TypeName: System.DateTime
```

Name	MemberType	Definition
-----	-----	-----
Add	Method	System.DateTime Add(TimeSpan value)
AddDays	Method	System.DateTime AddDays(Double value)

ToFileTimeUtc	Method	System.Int64 ToFileTimeUtc()
ToLocalTime	Method	System.DateTime ToLocalTime()
ToLongDateString	Method	System.String ToLongDateString()
ToLongTimeString	Method	System.String ToLongTimeString()
ToOADate	Method	System.Double ToOADate()

Parmi tous les membres que l'on peut trouver dans un type .NET, on va trouver ce qu'on appelle des membres statiques. Ces membres semblables aux autres diffèrent par le fait qu'ils doivent être appelés sans instancier au préalable la classe à laquelle ils se rapportent. Par exemple, le type `DateTime` dispose selon les informations données par le site MSDN de certains membres statiques comme *Now*, *Today*, *UtcNow*, etc.

Et pour utiliser une méthode, ou une propriété « statique » d'une classe du Framework, il suffit simplement d'utiliser la syntaxe suivante : [`<Espace de noms>.<Type .NET>`]:<Membre-statique>

Exemple :

```
PS > [System.DateTime]::Now
dimanche 4 octobre 2009 17:32:27
```



Pour faire référence à un type, on spécifie son nom entre crochets. Exemple : `[System.DateTime]`

Notez au passage, qu'il s'agit du même résultat retourné par la commandelette **Get-Date** :

```
PS > Get-Date
dimanche 4 octobre 2009 17:32:35
```



Il existe des classes contenant uniquement des membres statiques. Ces classes sont dites « statiques ». C'est-à-dire qu'elles ne peuvent pas être instanciées à l'aide de la commandelette **New-Object**.

Pour connaître les membres statiques contenus par un type, il existe deux solutions :

- Se rendre sur la page du site MSDN correspondant au type voulu, et observer tous les membres définis avec le petit logo  pour signifier « static ».
- Utiliser le paramètre **-static** avec la commandelette **Get-Member**.

Exemple :

```
PS > [System.DateTime] | Get-Member -static

TypeName: System.DateTime

Name      MemberType Definition
----      -
Compare    Method      static System.Int32 Compare(DateTime t1,
DaysInMonth Method      static System.Int32 DaysInMonth(Int32 year,
Equals      Method      static System.Boolean Equals
FromBinary  Method      static System.DateTime
FromBinary(Int64 dateData)...
```

1. Créer une instance de type (Objet)

Comme dans tout langage orienté objet, la création d'un objet n'est autre qu'une instanciation de type. Et avec PowerShell, l'instanciation de type, qu'il soit .NET ou COM (comme nous le verrons dans la partie suivante) est réalisée avec la commandelette **New-Object**.



Pour instancier des objets de type COM, la commandelette **New-Object** nécessite le paramètre **-ComObject**.

À chaque fois que l'on instancie un type par l'intermédiaire de la commandelette **New-Object**, on fait appel à un constructeur. Un constructeur est une méthode portant le même nom que le type en question, et qui permet généralement l'initialisation de variables. Chaque type possède au moins un constructeur. Et on parle de surcharge de constructeur lorsqu'il existe plusieurs constructeurs utilisant différents arguments. Pour connaître la liste des surcharges d'un constructeur, la solution la plus rapide est de se rendre sur le site MSDN, pour observer les caractéristiques du type étudié.



Comme nous allons le voir un peu plus loin dans cette partie, on parle de constructeur par défaut quand celui-ci ne nécessite aucun paramètre pour instancier le type. Mais prudence car, malgré son nom, tous les types ne possèdent pas un tel constructeur.

Dans le cadre d'une utilisation .NET, la commandelette **New-Object** possède deux paramètres (hors paramètres communs), dont voici le détail :

Paramètre	Description
-typeName	Spécifie le nom complet de la classe .NET, c'est-à-dire l'espace de noms plus la classe.
-argumentList	Spécifie une liste d'arguments à passer au constructeur de la classe .NET.



On peut omettre « System » dans la description des types se situant sous l'espace de noms « System », « System » étant l'espace de noms par défaut.

Exemple :

```
New-Object -typeName System.DateTime
```

est équivalent à :

```
New-Object -typeName DateTime
```

Exemple :

Création d'une date avec l'objet .NET DateTime.

Nous allons dans cet exemple, créer un objet de type DateTime. C'est-à-dire une instance du type System.DateTime. Puis vérifier sa valeur et son type (avec la méthode GetType).

```
PS > $var = New-Object -typeName DateTime
PS > $var

lundi 1 janvier 0001 00:00:00

PS > $var.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     DateTime                                     System.ValueType
```

Remarquez que nous avons fait appel au constructeur par défaut puisque nous n'avons fourni aucun paramètre. Par conséquent, notre objet a pour valeur la date du 1^{er} janvier de l'année 01 à 0h00.

Essayons maintenant de créer cet objet à l'aide du constructeur surchargé suivant :

DateTime (Int32, Int32, Int32) : ce constructeur initialise une nouvelle instance de la structure DateTime avec l'année, le mois et le jour spécifiés.

En utilisant ce constructeur, et en complétant correctement les valeurs comme ci-dessous, nous obtenons un objet date ayant pour valeur la date du 13 février de l'année 2008 à 0h00 :

```
PS > $var = New-Object -typeName DateTime -argumentList 2008, 2, 13
PS > $var

mercredi 13 février 2008 00:00:00
```



Pour tous les types définis par défaut dans PowerShell, l'utilisation de la commandelette **New-Object** n'est pas utile. Il suffit de spécifier le type entre crochets pour faire appel au constructeur.

Exemple :

```
PS > [System.DateTime]'2/12/2008'

mardi 12 février 2008 00:00:00
```

Ou

```
PS > [System.DateTime]$Date = '2/12/2008'
PS > $Date

mardi 12 février 2008 00:00:00
```

Exemple :

Création d'une chaîne avec l'objet .NET String.

Voici un exemple qui met en avant l'importance des constructeurs. Nous allons à présent tenter de créer un objet de type String. Pour ce faire, nous allons instancier la classe System.String à l'aide de la commande **New-Object**, mais sans préciser d'argument.

Voici le résultat :

```
PS > $Chaine = New-Object -typeName System.String

New-Object : Constructeur introuvable. Impossible de trouver
un constructeur approprié pour le type string.
```

PowerShell nous informe qu'il ne trouve pas de constructeur approprié, et que par conséquent il ne peut créer l'objet. Pour instancier cette classe, il est nécessaire de fournir un ou plusieurs arguments au constructeur.

Exemple :

```
PS > $Chaine = New-Object -typeName System.string -argumentList 'Bonjour'
PS > $Chaine
Bonjour
```

Notez que l'on peut aussi simplement se passer de **-typeName** et de **-argumentList** et écrire :

```
PS > $Chaine = New-Object System.String 'Bonjour'
```



Pour les habitués de la programmation objet, sachez qu'il est possible de spécifier la liste des arguments d'un constructeur entre parenthèses.

Exemple :

```
PS > $Chaine = New-Object -typeName System.String -argumentList 'Bonjour'
```

Est équivalent à :

```
PS > $Chaine = New-Object System.String('Bonjour')
```

Exemple :

Tentative de création d'une Windows Form (formulaire graphique).

Essayons maintenant d'instancier le type `System.Windows.Forms.Form` pour créer une *Windows Form*. Pour ce faire, utilisons une fois encore la commande **New-Object** :

```
PS > $var = New-Object -typeName 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.
```

Un message d'erreur de ce type est affiché à chaque fois que nous essayons d'instancier un type du Framework qui n'est pas chargé via une « assembly » dans PowerShell.

2. Les assemblies

Élément incontournable du Framework .NET, une assembly peut être considérée comme un ensemble de types .NET constituant une unité logique exécutable par le CLR (*Common Language Runtime*) du Framework. Cependant, pour la plupart des utilisateurs PowerShell que nous sommes, le terme de bibliothèque de type .NET suffira.

Même si elle pourrait s'y apparenter, dans le concept, une assembly n'est pas la même chose qu'une DLL (*Dynamic Link Library*), puisqu'une assembly est composée d'un exécutable et de plusieurs DLLs indissociables les uns des autres, ainsi qu'un manifeste garantissant les bonnes versions des DLLs chargées.

Élément indispensable au bon fonctionnement de PowerShell, certaines assemblies du Framework sont chargées dès le démarrage de PowerShell, de façon à garantir l'utilisation d'un certain nombre de types d'objets.

- Pour connaître les assemblies chargées par PowerShell, tapez la commande suivante :

```
PS > [System.AppDomain]::CurrentDomain.GetAssemblies()

GAC    Version    Location
---    -
True   v2.0.50727 C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorlib.
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Conso...
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.Management.Automati...
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\System\2.0.0.0__b77a5c5619...
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c...
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.Configuration.Insta...
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Comma...
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.ServiceProcess\2.0.
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Secur...
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Comma...
True   v2.0       C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Conso...
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.Management\2.0.0.0_
True   v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.DirectoryServices\2...
True   v2.0.50727 C:\Windows\assembly\GAC_32\System.Data\2.0.0.0__b77a5c5...
True   v2.0       C:\Windows\assembly\GAC_MSIL\System.Management.Automati...
True   v2.0.50727 C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorlib.
True   v2.0       C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Secur...
True   v2.0       C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Comma...
```



Pour connaître le nombre d'assemblies chargées, tapez la commande suivante :
`[System.AppDomain]::CurrentDomain.GetAssemblies().Count`

Chaque assembly retournée par la commande `[System.AppDomain]::CurrentDomain.GetAssemblies()` est de type *System.Reflection.Assembly* et possède donc une palette de méthodes, dont `GetExportedTypes` qui permet de lister tous les types contenus dans une assembly.

Exemple :

```
PS > $assemblies = [System.AppDomain]::CurrentDomain.GetAssemblies()
PS > $assemblies[0]
```

GAC	Version	Location
True	v2.0.50727	C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorlib.dll

```
PS > $assemblies[0].GetExportedTypes()
```

IsPublic	IsSerial	Name	BaseType
True	True	Boolean	System.ValueType
True	False	Buffer	System.Object
True	True	Byte	System.ValueType
True	True	CannotUnloadAppDomainException	System.SystemException
True	True	Char	System.ValueType
True	True	CharEnumerator	System.Object
True	False	Console	System.Object
True	True	ConsoleColor	System.Enum
...			



Pour connaître le détail de toutes les assemblies chargées, utilisez la commande suivante :
`[System.AppDomain]::CurrentDomain.GetAssemblies() | format-list *`

Voici le détail de la première assembly chargée :

```
CodeBase          : file:///C:/Windows/Microsoft.NET/Framework/
v2.0.50727/mscorlib.dll
EscapedCodeBase    : file:///C:/Windows/Microsoft.NET/Framework/
v2.0.50727/mscorlib.dll
FullName           : mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
EntryPoint         :
Evidence           : {<System.Security.Policy.Zone version="1">
<Zone>MyComputer</Zone>
</System.Security.Policy.Zone>
, <System.Security.Policy.Url version="1">
<Url>file:///C:/Windows/assembly/GAC_32/mscorlib/
2.0.0.0_b77a5c561934e089/mscorlib.dll</Url>
</System.Security.Policy.Url>
, <System.Security.Policy.GacInstalled version="1"/>
, mscorlib...}
ManifestModule     : CommonLanguageRuntimeLibrary
ReflectionOnly     : False
Location           : C:\Windows\Microsoft.NET\Framework\
v2.0.50727\mscorlib.dll
ImageRuntimeVersion : v2.0.50727
GlobalAssemblyCache : True
HostContext        : 0
```

3. Charger une assembly

Depuis le début de cet ouvrage (à l'exception de la création d'une Windows Form), nous avons utilisé que des types « intégrés » grâce aux assemblies chargées au démarrage de PowerShell. Mais bien d'autres types sont disponibles, et pour les importer, il faut charger les assemblies correspondantes. L'exemple le plus concret concerne l'utilisation des forms Windows. L'utilisation des objets graphiques n'est pas possible avec PowerShell sans prendre soin de charger l'assembly *System.Windows.Forms*.

PowerShell ne disposant pas d'une commande capable de charger des assemblies, nous allons une nouvelle fois faire appel à une classe .NET pour y parvenir. Utilisons la méthode `LoadWithPartialName` de la classe *System.Reflection.Assembly* pour charger l'assembly *System.Windows.Forms* :

```
PS > [System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
```

GAC	Version	Location
-----	---------	----------

```
---
True v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.Windows.Forms\2.0.0...
```

La méthode `LoadWithPartialName` n'est pas la seule permettant de charger une assembly : les méthodes `Load` ou `LoadFrom` donnent exactement le même résultat. La différence réside dans le fait que dans un cas nous utilisons le nom court de l'assembly et dans l'autre le nom long (ou nom fort). Un nom fort étant constitué de quatre parties : le nom de l'assembly, la version, la culture et le jeton public (version compressée de la clé publique de l'assembly).

Exemple :

```
PS > [System.Reflection.Assembly]::Load('System.Windows.Forms,
version=2.0.0.0, culture=neutral, '+ 'PublicKeyToken=b77a5c561934e089')
```

GAC	Version	Location
True	v2.0.50727	C:\Windows\assembly\GAC_MSIL\System.Windows.Forms\2.0.0...



Bien que l'utilisation de la méthode `LoadWithPartialName` soit plus simple d'utilisation, elle ne permet pas de vérifier que l'on charge la bonne version d'une assembly. C'est pourquoi cette méthode est signalée comme susceptible de disparaître dans les versions à venir du Framework.

4. Lister les types contenus dans les assemblies

Lorsque l'on prévoit d'utiliser un type particulier du Framework, il peut être intéressant de savoir si ce type est déjà chargé ou non, afin d'éviter un chargement d'assembly qui pourrait s'avérer inutile. Pour cela, il existe une technique qui consiste à créer une commande capable de récupérer tous les types disponibles avec les assemblies chargées en mémoire et faire ensuite un tri sur leur espace de noms.

Par exemple, pour obtenir le contenu de l'assembly `System.Windows.Forms`, les deux lignes de commandes suffisent :

```
PS > $FormAssembly = [System.AppDomain]::CurrentDomain.GetAssemblies() |
```

```
where {$_.Fullname -match 'System.windows.forms'}
```

```
PS > $FormAssembly.GetExportedTypes() | foreach{$_.Fullname}
```

```
...
```

```
System.Windows.Forms.CaptionButton
System.Windows.Forms.CharacterCasing
System.Windows.Forms.CheckBox
System.Windows.Forms.CheckBox+CheckBoxAccessibleObject
System.Windows.Forms.CheckBoxRenderer
System.Windows.Forms.ListControl
System.Windows.Forms.ListBox
System.Windows.Forms.ListBox+ObjectCollection
System.Windows.Forms.ListBox+IntegerCollection
System.Windows.Forms.ListBox+SelectedIndexCollection
System.Windows.Forms.ListBox+SelectedObjectCollection
System.Windows.Forms.CheckedListBox
System.Windows.Forms.CheckedListBox+ObjectCollection
System.Windows.Forms.CheckedListBox+CheckedIndexCollection
System.Windows.Forms.CheckedListBox+CheckedItemCollection
System.Windows.Forms.CheckState
System.Windows.Forms.Clipboard
```

```
...
```

Soit plus de mille types, parmi lesquels, certains comme : `ListBox`, `TextBox`, `Form`, qui sont bien connus de ceux qui ont un jour dû développer une petite interface graphique.

Bien que la démarche de recherche d'un type ne soit pas non plus fastidieuse, il est toujours intéressant d'automatiser ce genre de tâche. Pour cela, vous pouvez créer une fonction de recherche sur les types disponibles dans les assemblies chargées par PowerShell.


```
PS > function Get-TypeName ($nom = '.') {  
    [System.AppDomain]::CurrentDomain.GetAssemblies() |  
    Foreach-Object{$_ .GetExportedTypes() } |  
    Where-Object {$_ .Fullname -match $nom} | %{$_ .Fullname}}
```

Dans cette fonction, composée de trois pipelines, nous récupérons dans un premier temps, via la méthode `CurrentDomain.GetAssemblies`, toutes les assemblées chargées par PowerShell.

Le résultat passe ensuite par le premier pipeline pour finalement se voir appliquer la méthode `GetExportedTypes`, permettant de lister le contenu de chaque assembly.

Le reste de la fonction permet, par l'intermédiaire d'un `Where-Object`, de faire un tri sur les noms de types passés par le second pipeline et d'en afficher le nom complet.

Ainsi, en utilisant cette fonction, nous pouvons, d'une simple commande, retrouver tout type comportant le mot clé que vous aurez défini.

Exemple avec le mot clé `TextBox` :

```
PS > Get-TypeName TextBox  
  
System.Windows.Forms.TextBoxBase  
System.Windows.Forms.TextBox  
System.Windows.Forms.DataGridTextBox  
System.Windows.Forms.DataGridTextBoxColumn
```

La fonction nous renvoie tous les types comprenant dans leur nom complet le mot `TextBox`.



De façon à pouvoir réutiliser cette fonction ultérieurement, nous vous conseillons de l'inscrire dans votre profil.

Manipuler les objets .NET

L'utilisation des objets .NET donne à PowerShell une ouverture sur des milliers de classes prêtes à l'emploi. Par conséquent, manipuler les objets .NET, c'est permettre une plus grande flexibilité à vos scripts mais également d'en élever le niveau jusqu'à flirter avec la programmation objet.

Afin d'illustrer ces propos, nous allons dans cette troisième partie, expliquer étape par étape comment exploiter quelques objets à travers différentes situations qu'un administrateur système peut rencontrer, comme :

- l'envoi d'un mail,
- le Wake On Lan (réveil en ligne),
- la gestion des journaux d'événements de postes distants,
- la compression de fichiers.

1. Envoyer un e-mail

Dans cet exemple, nous allons détailler l'envoi d'un e-mail en nous appuyant sur les objets proposés par le Framework. Avant toutes choses, regardons quelles classes sont disponibles sous l'espace de noms System.Net.Mail. Pour cela, nous vous invitons à les lister grâce à la fonction Get-TypeName créée précédemment.

```
PS > Get-TypeName System.Net.Mail

System.Net.Mail.AttachmentBase
System.Net.Mail.AlternateView
System.Net.Mail.AlternateViewCollection
System.Net.Mail.Attachment
System.Net.Mail.AttachmentCollection
System.Net.Mail.LinkedResource
System.Net.Mail.LinkedResourceCollection
System.Net.Mail.MailAddress
System.Net.Mail.MailAddressCollection
System.Net.Mail.DeliveryNotificationOptions
System.Net.Mail.MailMessage
System.Net.Mail.MailPriority
System.Net.Mail.SendCompletedEventHandler
System.Net.Mail.SmtpDeliveryMethod
System.Net.Mail.SmtpClient
System.Net.Mail.SmtpException
System.Net.Mail.SmtpFailedRecipientException
System.Net.Mail.SmtpFailedRecipientsException
System.Net.Mail.SmtpAccess
System.Net.Mail.SmtpPermissionAttribute
System.Net.Mail.SmtpPermission
System.Net.Mail.SmtpStatusCode
```

Soit une vingtaine de classes au total. Bien que les noms soient assez explicites, rien ne vaut un petit coup d'œil sur le site MSDN pour en connaître la description.

Pour un envoi de mail « classique », c'est-à-dire sans accusé ni pièce jointes, nous nous intéresserons uniquement aux classes « MailMessage » (classe représentant un message électronique) et « SmtpClient » (classe qui permet l'envoi de courriers électroniques à l'aide du protocole SMTP (*Simple Mail Transfer Protocol*)).

La première étape consiste à créer notre objet « message » de type System.Net.Mail.MailMessage. Pour cela utilisons la commande suivante :

```
PS > $Message = New-Object System.Net.Mail.MailMessage
```

Une fois l'objet « message » créé, il faut ensuite le configurer, c'est-à-dire lui définir un certain nombre d'attributs comme son expéditeur, le destinataire, l'objet et le corps du message. Pour cela, observons les propriétés disponibles de l'objet avec la commande **Get-Member** :

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

Dans notre exemple, nous nous contenterons d'attribuer à notre objet que les paramètres essentiels :

```
$message.Body = 'Message envoyé avec PowerShell'
$message.From = 'robot@powershell-scripting.com'
$message.Subject = 'Mon premier message'
$message.To.Add('admin@powershell-scripting.com')
```

À ce stade, il ne nous reste qu'une seule chose à faire : envoyer le message via le protocole SMTP. Et pour cela trois commandes suffisent.

La première pour créer un objet de type `SmtpClient` :

```
PS > $client = New-Object System.Net.Mail.SmtpClient
```

La seconde pour connecter le client au serveur SMTP qui va permettre l'envoi du mail :

```
PS > $client.Set_host('SERVEUR2008')
```

Et la dernière pour envoyer définitivement le message :

```
PS > $client.Send($message)
```

Soit le script suivant en version complète :

Script : Envoi d'un e-mail

```
#Send-email.ps1
#Script permettant l'envoi d'un e-mail

$expediteur = 'expediteur@host.com'
$destinataire = 'destinataire@host.com'
$serveur = 'mail.host.com'
$sujet = 'Envoi de mail via powershell' + $(get-date)
$texte = 'ceci est le corps du message'

# Création de l'objet MailMessage
$message = New-Object System.Net.Mail.MailMessage

# Ajout des propriétés
$message.Body = $texte
$message.From = $expediteur
$message.Subject = $sujet
$message.To.Add($destinataire)

# Création de l'objet SmtpClient
$client = New-Object System.Net.Mail.SmtpClient

# Définition de la propriété concernant l'hôte
$client.Set_Host($serveur)

# Envoi du message avec la method Send
$client.Send($message)
```



Cet exemple d'envoi de mail concerne la version 1 de PowerShell. En effet, PowerShell v2 intègre désormais nativement la commande `Send-MailMessage` et cette dernière est bien plus complète que notre exemple.

2. Wake On Lan

Le Wake On Lan (WOL) est un procédé qui permet d'allumer un poste éteint via l'envoi sur le réseau Ethernet, d'une suite d'octets un peu particulière appelée « paquet magique ».



Aujourd'hui pratiquement toutes les cartes mères le supportent, néanmoins il se peut que le Wake On Lan soit désactivé par défaut dans le BIOS.

Le paquet magique permettant de déclencher le WOL est une suite de 102 octets dont les 6 premiers prennent la valeur hexadécimale FF, et les 96 suivants sont 16 fois la répétition de l'adresse MAC (*Media Access Control*) de la carte réseau de l'ordinateur distant. Pour créer ce paquet, nous utiliserons le tableau d'octets suivant :

```
PS > [byte[]]$Adresse_Mac = 0x00, 0x11, 0x43, 0x0E, 0x97, 0x4F
PS > [byte[]]$paquet = 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
PS > $paquet += $Adresse_Mac * 16
```

Une fois le paquet constitué, il faut maintenant l'envoyer via le réseau. Et pour ce faire, nous allons utiliser la classe `UdpClient` (sous l'espace de noms `System.Net.Sockets`) qui fournit les services réseaux nécessaires à l'envoi de datagrammes UDP (*User Datagram Protocol*) :

```
PS > $UdpClient = New-Object System.Net.Socket.UdpClient
```

C'est grâce à cette classe et plus particulièrement à la méthode `Connect` que l'on va pouvoir établir une connexion avec un hôte distant. Il suffira ensuite d'un simple appel à la méthode `Send` pour finaliser l'envoi du datagramme :

```
PS > $UdpClient.Connect(([System.Net.IPAddress]::Broadcast),1600)
PS > $UdpClient.Send($Paquet,$Paquet.Length)
```

Notez que lors de l'appel de la méthode `Connect`, nous utilisons à la fois, la propriété statique *Broadcast* qui retourne l'adresse IP de broadcast (255.255.255.255) de façon à garantir une diffusion générale du datagramme, ainsi que le numéro de port 1600.



Lors de l'envoi d'un datagramme, faites attention à ne pas choisir un port déjà utilisé par une autre application. Pour rappel, les ports utilisés par défaut sont les ports allant de 0 à 1023.

Voici notre script de Wake On Lan complet :

Script : Script de Wake On Lan

```
# WOL.ps1
# Script permettant d'allumer une machine distante

[byte[]]$Adresse_Mac = 0x00, 0x11, 0x43, 0x0E, 0x97, 0x4F
[byte[]]$paquet = 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
$paquet += $Adresse_Mac * 16
$UdpClient = New-Object System.Net.Sockets.UdpClient
$UdpClient.Connect(([System.Net.IPAddress]::Broadcast),1600)
$UdpClient.Send($Paquet,$Paquet.Length)
```

En conséquence, si le poste distant est bien connecté au réseau, et à la condition que sa carte mère soit bien configurée pour prendre en compte le WOL, vous aurez l'agréable surprise de réveiller un ordinateur en plein sommeil.

3. Gérer les journaux d'événements

Passons à présent à la gestion des journaux d'événements. Dans sa configuration de base, PowerShell intègre une commandelette du nom de `Get-EventLog` qui permet d'obtenir des informations à propos des journaux des événements de l'ordinateur local.

```
PS > Get-EventLog -list
```

Max(K)	Retain	OverflowAction	Entries	Name
10	240	0 OverwriteAsNeeded	23	Antivirus
20	480	0 OverwriteAsNeeded	1 008	Application
20	480	0 OverwriteAsNeeded	6 903	Système
15	360	0 OverwriteAsNeeded	242	Windows PowerShell

Seulement voilà, cette commandelette ne permet pas l'accès aux informations contenues dans les journaux d'événement de postes distants. Et pour remédier à ce qui peut s'apparenter à une carence, nous allons faire appel à la classe `System.Diagnostics.EventLog` du Framework .NET car celle-ci possède une méthode qui permet d'accéder aux journaux d'une machine distante.



Les événements de type sécurité ne sont accessibles qu'avec un compte disposant de privilèges.

La méthode est en l'occurrence la méthode statique `GetEventLogs`.

```
PS > $Evenements = [System.Diagnostics.EventLog]::GetEventLogs('SERVEUR2008')
PS > $Evenements
```

Max(K)	Retain	OverflowAction	Entries	Name
16	384	0 OverwriteAsNeeded	381	Application
	512	0 OverwriteAsNeeded	78	Service d'annuaire
	512	7 OverwriteOlder	18	Serveur DNS
	512	0 OverwriteAsNeeded	23	Service de réplication de fichiers
131	072	0 OverwriteAsNeeded	4 247	Sécurité
16	384	0 OverwriteAsNeeded	304	Système
15	360	0 OverwriteAsNeeded	64	Windows PowerShell

Cette méthode retourne un tableau d'objets, où chaque élément correspond à un journal d'événements particulier. Et pour connaître le contenu d'un journal, il suffit d'utiliser la propriété « `entries` ».

```
PS > $Evenements[0].Entries
```

Index	Time	Type	Source	EventID	Message
1	sept. 15 2...	Info	ESENT	100	svchost (632) Le moteur de la ...
2	sept. 15 2...	Info	ESENT	101	svchost (632) Le moteur de la ...
3	sept. 15 2...	Info	ESENT	100	svchost (632) Le moteur de la ...

L'utilisation des journaux d'événements via le Framework .NET nous permet donc de récupérer tous les événements d'un poste distant. Reste maintenant à déterminer quels éléments garder ou non.

Pour vous aider dans cette tâche qu'est le tri d'événements, l'exemple suivant vous montre comment déterminer en une ligne de commandes, tous les événements du journal application qui correspondent à une erreur d'application (ID événement 1000).

```
PS > [System.Diagnostics.EventLog]::GetEventLogs('SERVEUR2008') |
Where-Object {$_.Get_LogDisplayName() -eq 'Application'} |
Foreach($_.entries) | Where-Object{$_.EventID -eq 1000}
```

Index	Time	Type	Source	EventID	Message
43	sept. 10 2...	Info	LoadPerf	1000	La description de l'ID d'événement '1073742824'
61	sept. 10 2...	Info	LoadPerf	1000	La description de l'ID d'événement '1073742824'
140	sept. 10 2...	Info	LoadPerf	1000	La description de l'ID d'événement '1073742824'



L'exemple proposé ci-dessus n'est pas l'unique moyen d'accéder aux journaux d'événements d'un poste distant. Les requêtes WMI permettent également d'obtenir le même résultat, voire même d'optimiser les requêtes portant sur un grand nombre potentiel d'événements.

4. Compresser un fichier

Passons à un autre exemple. Regardons à présent comment compresser et décompresser un fichier sans passer par un outil tiers. Cette fonctionnalité qui a fait son apparition dans le Framework 2.0 utilise des classes situées sous l'espace de nom `System.IO.Compression`. Et la classe qui va particulièrement nous intéresser est la classe `GZipStream` qui fournit des méthodes et des propriétés permettant la compression et la décompression de flux de données.

Seulement voilà, en regardant de plus près le constructeur de cette classe, on s'aperçoit qu'il nécessite, non pas un fichier, mais un flux (`Stream`).

Pour lui fournir le flux dont il a besoin, il faut donc utiliser la classe `FileStream` disponible sous l'espace de nom `System.IO`.

Pour compresser un fichier, la première étape consiste à récupérer le flux d'informations du fichier (en fait, son contenu sous forme d'une suite d'octets) et à le copier dans un tableau d'octets. Pour cela la première action consiste à instancier la classe `FileStream` avec pour paramètres constructeur le chemin du fichier ainsi que la façon dont le système d'exploitation doit ouvrir le fichier :

```
PS> $Stream = New-Object System.IO.FileStream <Chemin_du_Fichier>, 'open'
```

Puis vient le moment de la création d'un buffer d'une taille suffisante pour y insérer le flux d'informations :

```
PS > $buffer = New-Object System.Byte[] $Stream.length
```

Appliquons le contenu de l'objet `$Stream` du premier au dernier octet du flux dans le buffer grâce à la méthode `Read`. Et libérons le fichier :

```
PS > $Stream.Read($buffer,0,$Stream.length)
PS > $Stream.Close()
```

Un fois le flux récupéré, il faut maintenant le compresser grâce à la classe `GZipStream`, et l'insérer dans un fichier. Cela passe d'abord par la création d'un flux, toujours avec l'objet `System.IO.FileStream` :

```
PS > $Stream = New-Object System.IO.FileStream <Nom_du_Fichier>, 'create'
```

Puis par la création d'un objet de type « flux compressé ». Pour cela, nous utilisons ici la classe `GZipStream` avec comme argument `$Stream` pour spécifier dans quel flux (`FileStream`) nous voulons y insérer les informations, et `Compress` pour choisir un flux de type compressé.

```
PS > $fichier_gzip = New-Object System.IO.Compression.
GZipStream($Stream, 'compress')
```

Il reste à écrire la totalité du flux compressé avec la méthode `Write` et ensuite libérer le fichier.

```
PS > $fichier_gzip.Write($buffer,0,$buffer.Length)
PS > $fichier_gzip.Close()
```

Maintenant que nous avons tous les éléments de la compression en main, créons une fonction qui sera intéressante de garder dans votre profil.

```
function Convert-ToGzip {
param([string]$fichier)
if(Test-Path $fichier)
{
    $Stream = New-Object System.IO.FileStream $fichier, 'open'
    $buffer = New-Object System.Byte[] $Stream.length
    $Stream.Read($buffer,0,$Stream.length)
    $Stream.Close()
    $nom_zip=$fichier + '.Gzip'
    $Stream = New-Object System.IO.FileStream $nom_zip, 'create'
    $fichierzip =
    New-Object System.IO.Compression.GZipStream($Stream, 'compress',0)
    $fichierzip.Write($buffer,0,$buffer.Length)
    $fichierzip.Close()
    Write-Host 'Fin de compression'
}
}
```

Regardons maintenant quel résultat nous pouvons attendre d'une telle compression. Pour le savoir, créons un fichier texte contenant le résultat de la commande **Get-Process**, et utilisons notre fonction `ConvertTo-Gzip` pour en obtenir également une version compressée :

```
PS > Get-Process | Out-File 'c:\Process.txt'
PS > Convert-ToGzip -fichier 'c:\Process.txt'
```

Et enfin, observons le résultat avec la commande **Get-ChildItem** :

```
PS > Get-ChildItem
```

Répertoire : C:\Scripts

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	23/09/2007 11:48	12598	Process.txt
-a---	23/09/2007 11:50	2151	Process.Gzip

Le résultat est sans appel, 12598 octets contre 2151 après compression.

Créer des interfaces graphiques

À l'heure du retour en force de la ligne de commandes, il est tout à fait légitime de se poser la question : « À quoi bon vouloir des interfaces graphiques alors que la philosophie de PowerShell est de se substituer à ces dernières ? ». En effet, si cela peut sembler paradoxal de prime abord, en prenant un peu de recul, il est facile de trouver de nombreuses bonnes raisons qui font que les interfaces graphiques sont finalement complémentaires aux scripts

Parmi les bonnes raisons, on peut trouver au moins les suivantes :

- Besoin de fournir un script à des utilisateurs. Ainsi, par le biais de l'interface graphique, des utilisateurs peu expérimentés en scripting peuvent interagir de façon conviviale.
- Faciliter l'utilisation d'un script qui nécessite de nombreux paramètres. Tous les scripts n'ont pas nécessairement besoin d'être exécutés uniquement en tant que tâche planifiée. De plus si l'on peut s'éviter l'apprentissage de tous les paramètres d'un script, pourquoi se priver d'une interface graphique qui présenterait ces derniers sous forme de cases à cocher ? On pourrait très bien imaginer qu'un script lancé sans paramètres affiche une interface graphique pour demander leur saisie. Ce même script alors lancé en ligne de commandes avec ses paramètres fonctionnerait comme tout script PowerShell classique. L'un n'empêche pas l'autre, bien au contraire !
- Disposer d'une interface graphique flexible. Grâce au fait que PowerShell soit un langage interprété, c'est-à-dire non compilé, tout le monde peut avoir accès au code source. Par conséquent, il est aisé de modifier le code pour obtenir l'interface graphique qui répond exactement au besoin du moment. Ceci n'est pas faisable avec des outils classiques réalisés en langages compilés tel que C, C# ou VB car généralement on ne dispose pas des sources.
- Faire progresser les utilisateurs dans leur connaissance de PowerShell. Par expérience, lorsque l'on donne un outil flexible à des utilisateurs un peu curieux, ces derniers finissent par se l'approprier et l'améliorent. Ce qui renforce assurément leur connaissance du langage.

1. Quelle technologie choisir ?

En fait tout dépend du niveau de fonctionnalités attendu par l'interface graphique, de la plate-forme sur laquelle votre script doit s'exécuter, et du temps dont vous disposez. Si vous souhaitez simplement vous assurer qu'une tâche particulière soit toujours faite parfaitement, alors le look de l'interface graphique a assez peu d'importance. Un cas typique est celui où en tant que responsable de l'exploitation, vous fournissez aux personnes qui déclarent les comptes dans l'Active Directory un script avec interface graphique qui, par exemple, force la mise en majuscule du nom et qui crée la home directory toujours au bon endroit et avec les bonnes permissions. Dans ce cas présent la technologie Windows Forms apportée par le Framework .NET 2.0 est généralement suffisante. En effet les *Windows Forms* possèdent le look Windows. On y trouve des boutons, des cases à cocher, des labels, des zones de saisie de texte, en bref le grand classique.

Si par contre, vous avez besoin de développer un script qui doit générer des graphiques 2D ou 3D, comme par exemple pour le suivi de l'espace disque de vos serveurs de fichiers, alors le résultat final revêt une grande importance. Dans ce cas, il sera généralement préférable de faire appel à la technologie **WPF** (*Windows Presentation Foundation*). De même que si vous voulez réaliser des interfaces graphiques extravagantes, qui sortent de l'ordinaire ; WPF sera un excellent candidat.

Un avantage qu'a WPF sur son frère Windows Forms pour la conception des interfaces graphiques est que l'on peut décrire ces dernières avec une grammaire basée sur le langage XML. Il s'agit du format XAML (prononcer « gzamel »). XAML, étant un langage descriptif, il facilite la création d'interfaces et apporte beaucoup de souplesse lors de la modification de ces dernières ; mais nous verrons cela par la suite...

À présent que vous en savez plus sur les technologies, voyons quels sont les pré-requis nécessaires pour leur mise en œuvre :

Windows Forms	WPF
Framework .NET 2.0 minimum	Framework .NET 3.0 minimum - 3.5 recommandé
PowerShell 1.0 et 2.0	PowerShell 2.0

À la vue de ces pré-requis, vous pouvez vous en douter, un script s'appuyant sur les Windows Forms sera plus portable qu'un script s'appuyant sur WPF. Ceci vient du fait que le couple Framework .NET 2.0/PowerShell 1.0 est maintenant presque un standard sur les PC en entreprise. Si vous prévoyez de faire fonctionner vos scripts utilisant WPF sur les plates-formes Windows 7 ou Windows Server 2008 R2, alors n'ayez pas d'inquiétude car les prérequis

sont installés par défaut. En revanche, toujours pour WPF, sous Vista il faudra installer PowerShell v2 et sous Windows XP a minima le Framework .NET 3.0 et PowerShell v2.

2. Windows Forms

a. Introduction aux Windows Forms

Microsoft Windows Forms, également appelé Winform est le nom donné aux interfaces graphiques apportées avec le Framework .NET. L'utilisation de ces Winforms, basées sur un ensemble de types disponibles dans l'assembly `System.Windows.Form`, permet de créer des interfaces au look Windows grâce à un accès aux éléments graphiques natifs de Windows.



Pour connaître les différents types graphiques dont l'espace de noms est `System.Windows.Form`, vous pouvez réutiliser la fonction `Get-TypeName` que nous avons développée précédemment. En utilisant la commande suivante, vous vous apercevrez qu'il existe plus d'un millier de types :

```
PS > (Get-TypeName System.Windows.Form).count
```

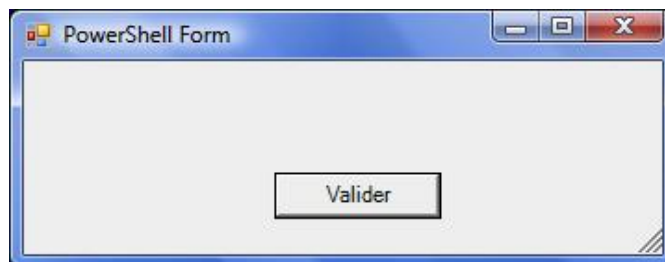
Attention toutefois, car ces types ne sont pas chargés au démarrage de PowerShell. Il vous faudra donc prendre soin de charger l'assembly `System.Windows.Forms` au préalable avec la commande :

```
[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
```

L'élément de base dans la construction d'une interface graphique Windows est appelé un contrôle, un contrôle peut aussi bien être un bouton, un texte, qu'une form, etc. Une form est une surface visuelle paramétrable sur laquelle vous pouvez venir greffer d'autres composants. Les forms bénéficient des mêmes techniques de développement que les interfaces du système d'exploitation Windows. Ce qui permet à chaque formulaire graphique créé, d'hériter des propriétés d'affichage et du thème de votre système d'exploitation. Par exemple, les forms développées sous Vista ou Windows 7 profiteront d'un effet de transparence des fenêtres (voir illustration ci-après).



Windows Forms selon les thèmes Windows 7



Form avec bouton

Chaque contrôle est un objet, et par conséquent, dispose d'un certain nombre de propriétés communes qui permettent d'ajuster la texture, la visibilité, la taille, la position, etc. La liste des propriétés et méthodes peut être obtenue en appliquant la commandelette **Get-Member** à l'objet.

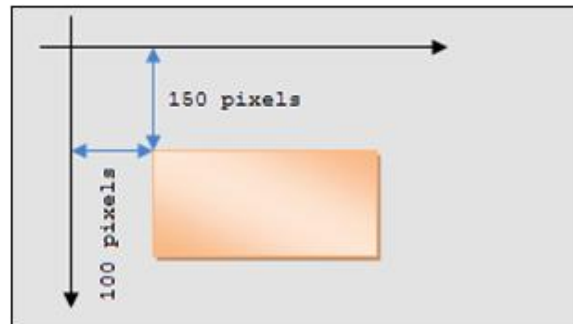
L'autre aspect intéressant des Winforms, est la gestion des événements (clic droit, clic gauche de souris, frappe clavier, etc.). Il en existe des dizaines et permettent d'ajouter un comportement à un contrôle. Comme par exemple, afficher du texte, fermer une form. Tout contrôle, qu'il soit une form ou un élément qui la compose, peut être soumis

à un ou plusieurs événements. Et chaque événement peut déclencher une ou plusieurs actions.

➤ PowerShell ne disposant pas d'un éditeur graphique pour la réalisation des Winforms, le positionnement des contrôles se fait de manière manuelle selon deux constantes : l'axe des ordonnées X et l'axe des abscisses Y, avec pour origine le coin supérieur gauche de son conteneur et pour unité le pixel.

Exemple :

Le positionnement d'un contrôle à 100 pixels sur l'axe des abscisses et 150 sur l'axe des ordonnées, donnera : location (100,150).



b. Création d'une form simple

Comme nous vous le disions précédemment, afin d'utiliser les types graphiques disponibles dans le Framework .NET, il est indispensable de charger l'assembly System.Windows.Forms qui contient toutes les classes permettant la création d'applications graphiques. Le chargement de l'assembly se fait de la manière suivante :

```
[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
```

Une fois les types graphiques disponibles, nous allons créer une form principale, sur laquelle nous pourrons greffer des composants graphiques, des menus, des contrôles, des boîtes de dialogue, etc.

```
# Création de la form principale  
$form = New-Object Windows.Forms.Form
```

En regardant les méthodes et propriétés de notre objet (soit plus de 500 !!!), nous allons chercher à personnaliser notre form. Comme par exemple lui ajouter un titre en utilisant la propriété « Text », et en lui attribuant une taille particulière :

```
#Affichage d'un titre  
$form.Text = 'PowerShell Form'  
  
# dimensionnement de la form  
$form.Size = New-Object System.Drawing.Size(360,140)
```

Notez que l'attribution d'une taille à une form nécessite de fournir à la propriété Size un objet de type System.Drawing.Size auquel nous attribuons deux variables sous forme (largeur, hauteur).

➤ Par défaut, la taille attribuée à une form est de 300 pixels en largeur et 300 en hauteur.

Ajoutons maintenant, un bouton à notre interface en créant un objet de type System.Windows.Forms.Button :

```
# Création bouton valider  
$bouton_quitter = New-Object System.Windows.Forms.Button  
$bouton_quitter.Text = 'Quitter'  
  
# Positionnement du bouton  
$bouton_quitter.Location = New-Object System.Drawing.Size(135,60)  
  
# Ajustement de la taille  
$bouton_quitter.Size = New-Object System.Drawing.Size(90,25)
```

Il faut désormais ajouter notre bouton à la form principale, pour cela utilisons la méthode Add appliquée au membre Controls de la form.

```
$form.Controls.Add($bouton_quitter)
```

Puis, pour finir, il nous faut bien évidemment afficher la form avec la méthode ShowDialog :

```
$form.ShowDialog()
```

La méthode Show permet également d'afficher la form. Mais cette dernière la fait aussitôt disparaître, puisqu'aucune boucle de message interne au système, n'est disponible pour bloquer la fenêtre.



En utilisant la méthode ShowDialog, vous créez une fenêtre enfant du processus PowerShell. Par contre, en utilisant la méthode statique Run de la classe System.Windows.Forms.Application, vous créez cette fois-ci une véritable application indépendante. Ainsi, lorsque vous fermerez la console PowerShell, votre form existera toujours. Ce qui n'est pas le cas avec la méthode ShowDialog.

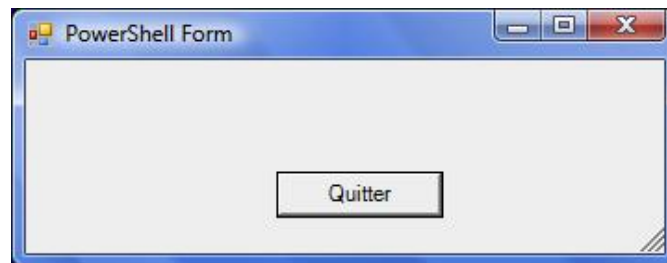
```
PS > $form = New-Object System.Windows.Form
PS > [System.Windows.Forms.Application]::Run($form)
```

En associant les bouts de code précédents, nous obtenons un script entier qui crée une interface graphique dont le résultat est donné dans la figure ci-dessous.

Script : Création d'une interface graphique

```
#Form_1.ps1
#Création d'une interface graphique

[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
$form = New-Object Windows.Forms.Form
$form.Text = 'PowerShell Form'
$form.Size = New-Object System.Drawing.Size(360,140)
$bouton_quitter = New-Object System.Windows.Forms.Button
$bouton_quitter.Text = 'Quitter'
$bouton_quitter.Location = New-Object System.Drawing.Size(135,60)
$bouton_quitter.Size = New-Object System.Drawing.Size(90,25)
$form.Controls.Add($bouton_quitter)
$form.ShowDialog()
```



Grâce à cet exemple, nous venons de créer une interface graphique avec PowerShell. Reste maintenant à ajouter d'autres fonctionnalités à cette form. C'est ce que nous allons voir dans la partie suivante.

c. Ajout de fonctionnalités à une form

Après avoir procédé à la création d'une form graphique élémentaire, nous allons maintenant progresser dans la création des interfaces graphiques avec l'insertion d'un menu ainsi que l'utilisation de la gestion d'événements et d'un Timer.

Les événements

Avec le Framework .NET, chaque composant graphique peut réagir à un ou plusieurs événements, pour peu qu'il soit correctement configuré. L'ajout d'un événement à un composant donné, s'effectue en appliquant la méthode Add_<nom_de l'événement>. Il suffit ensuite d'y insérer les actions à entreprendre entre les accolades à l'intérieur des parenthèses de la méthode.

Exemple :

Événement clic gauche (Add_Click).

```
$<Nom_du_bouton>.Add_Click(  
{  
    # Bloc d'instructions  
})
```



Lorsque vous utilisez des événements pour faire des modifications graphiques sur votre interface, il se peut que vous soyez dans l'obligation de rafraîchir la fenêtre pour que les modifications soient prises en compte.

Exemple :

Ajout d'événement à une form.

Dans cet exemple, nous allons nous appuyer sur la form développée dans la section précédente (Création d'une form simple) et ajouter un événement sur le bouton **Quitter** pour fermer la fenêtre quand un clic gauche y sera appliqué. Pour cela nous utiliserons la méthode Close définie dans l'événement Add_Click du bouton **Quitter**.

Script : Création d'une interface graphique avec bouton Quitter actif.

```
#Form_2.ps1  
#Création d'une interface graphique avec bouton Quitter actif  
  
[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')  
$form = New-Object Windows.Forms.Form  
$form.Text = 'PowerShell Form'  
$form.Size = New-Object System.Drawing.Size(360,160)  
$bouton_quitter = New-Object System.Windows.Forms.Button  
$bouton_quitter.Text = 'Quitter'  
$bouton_quitter.Location = New-Object System.Drawing.Size(135,80)  
$bouton_quitter.Size = New-Object System.Drawing.Size(90,25)  
  
# Ajout de l'événement 'clic gauche' au bouton  
$bouton_quitter.Add_Click(  
{  
    $form.Close()  
})  
  
$form.Controls.Add($bouton_quitter)  
$form.ShowDialog()
```

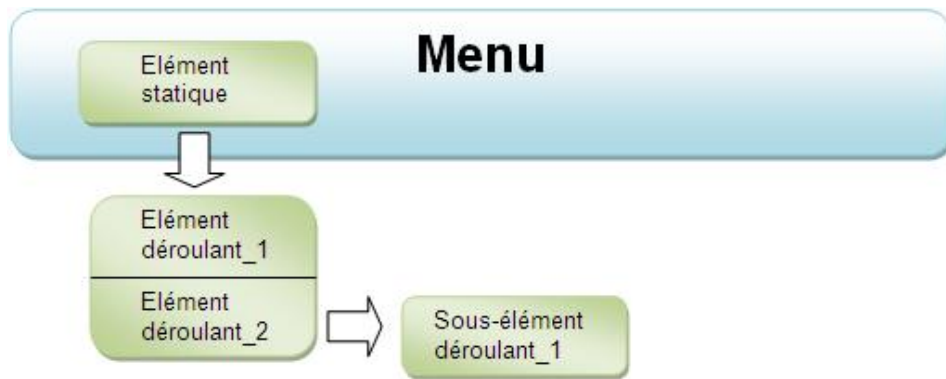
Résultat, la fenêtre se ferme lors d'un clic gauche sur le bouton **Quitter**.

Les menus

Pour étoffer encore un peu plus une interface, il est possible de créer une barre d'outils et de menus. Un composant de rangement sert à créer une arborescence de boutons sur lesquels il est également possible d'ajouter des événements.

La création d'un menu est réalisée avec l'instanciation de la classe MenuStrip (System.Windows.Forms.MenuStrip) qui a fait son apparition à partir du Framework 2.0.

À ce menu, peut venir se greffer des composants de type System.Windows.Forms.ToolStripMenuItem, pouvant eux aussi à leur tour contenir d'autres composants et ainsi former une arborescence de boutons.



L'ajout d'un élément statique au menu nécessite l'appel de la méthode Add se trouvant dans le membre Items appliquée au menu lui-même, alors que l'ajout d'un élément déroulant (ou sous élément) nécessite quant à lui la méthode Add se trouvant dans le membre DropDownItems appliquée à l'élément parent.

Exemple :

Correspondant à la création du menu ci-dessus.

```

# Création de l'objet Menu
$Menu = New-Object System.Windows.Forms.MenuStrip

# Déclaration des éléments
$elements =
New-Object System.Windows.Forms.ToolStripMenuItem('Element principal')
$element_1 =
New-Object System.Windows.Forms.ToolStripMenuItem('Element_1')
$element_2 =
New-Object System.Windows.Forms.ToolStripMenuItem('Element_2')
$element_3 =
New-Object System.Windows.Forms.ToolStripMenuItem('Sous_element_1')

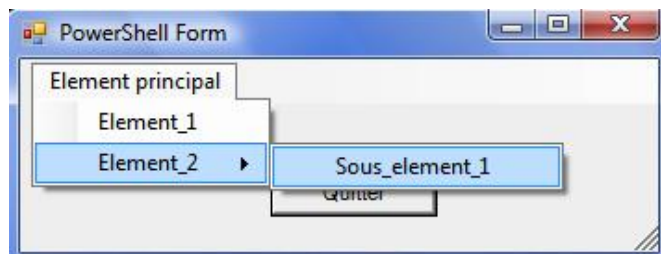
# Ajout des éléments
[void]$elements.DropDownItems.Add($element_1)
[void]$elements.DropDownItems.Add($element_2)
[void]$element_2.DropDownItems.Add($element_3)
[void]$Menu.Items.Add($elements)

# Ajout du menu à la form
$form.Controls.Add($Menu)
  
```



L'utilisation de [void] permet de ne pas afficher sur la console le résultat des commandes. Nous aurions pu faire l'équivalent en faisant une affectation de variable à la place de [void].

En ajoutant le bout de code précédant à notre petite interface, le résultat sera le suivant :



Modélisation des menus graphiques

Voici le script complet de notre interface graphique :

```

#Form_3.ps1
#Création d'une interface graphique
#avec bouton quitter actif et un menu
  
```

```

[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
$form = New-Object Windows.Forms.Form
$form.Text = 'PowerShell Form'
$form.Size = New-Object System.Drawing.Size(360,160)
$bouton_quitter = New-Object System.Windows.Forms.Button
$bouton_quitter.Text = 'Quitter'
$bouton_quitter.Location = New-Object System.Drawing.Size(135,80)
$bouton_quitter.Size = New-Object System.Drawing.Size(90,25)

# Ajout de l'événement 'clic gauche' au bouton
$bouton_quitter.Add_Click(
{
    $form.Close()
})

$form.Controls.Add($bouton_quitter)

# Création de l'objet Menu
$menu = New-Object System.Windows.Forms.MenuStrip

# Déclaration des éléments
$selements = New-Object System.Windows.Forms.ToolStripMenuItem('Element principal')
$selement_1 = New-Object System.Windows.Forms.ToolStripMenuItem('Element_1')
$selement_2 = New-Object System.Windows.Forms.ToolStripMenuItem('Element_2')
$selement_3 = New-Object System.Windows.Forms.ToolStripMenuItem('Sous_element_1')

# Ajout des éléments
[void]$selements.DropDownItems.Add($selement_1)
[void]$selements.DropDownItems.Add($selement_2)
[void]$selement_2.DropDownItems.Add($selement_3)
[void]$menu.Items.Add($selements)

# Ajout du menu à la form
$form.Controls.Add($menu)

# Affichage de la form
$form.ShowDialog()

```

Timer

L'implémentation d'un Timer dans une Winform permet le déclenchement automatique d'événements à intervalle de temps défini. Issu de la classe `System.Windows.Forms.Timer`, le Timer est un composant invisible dans une form dont l'intervalle de temps entre chaque exécution est fixé en millisecondes par la propriété `Interval`. La déclaration des instructions à effectuer à chaque déclenchement s'effectue en appliquant la méthode `Add_Tick`. Il suffit ensuite, comme pour les événements, d'y insérer les actions à entreprendre entre les accolades à l'intérieur des parenthèses de la méthode comme ci-dessous.

```

# création de l'objet
$timer = New-Object System.Windows.Forms.Timer

# Définition de l'intervalle à 1 seconde
$timer.Interval = 1000

$timer.Add_Tick({
    <Bloc d'instructions>
})

```

Avec les commandes ci-dessus, nous venons de configurer notre Timer. Mais il reste à déterminer quand celui-ci va démarrer et s'arrêter. Pour cela, il suffit d'appliquer les méthodes `Start` et `Stop` pour respectivement démarrer et arrêter le Timer.



Vous pouvez également démarrer/arrêter le Timer en affectant la valeur `True/False` à la propriété `Enabled` de l'objet Timer.

Afin de mieux assimiler comment utiliser un Timer, voici un exemple mettant en scène l'interface développée dans la partie « développement d'une form simple ». Nous allons cette fois y ajouter deux nouveaux contrôles :

- Un Label qui va afficher via la commande **Get-Date**, la date et l'heure actuelle.
- Un Timer, qui va chaque seconde, rafraîchir la valeur contenue dans le Label.

Script : Création d'une interface avec Timer incorporé.

```
#Form_4.ps1
# Création d'une interface avec Timer incorporé
[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
$form = New-Object Windows.Forms.Form

# Création de l'objet Timer
$timer = New-Object System.Windows.Forms.Timer
$form.Text = 'PowerShell Form'
$form.Size = New-Object System.Drawing.Size(360,160)
$bouton_quitter = New-Object System.Windows.Forms.Button
$bouton_quitter.Text = 'Quitter'
$bouton_quitter.Location = New-Object System.Drawing.Size(135,80)
$bouton_quitter.Size = New-Object System.Drawing.Size(90,25)
$bouton_quitter.Add_Click( {$form.Close()} )

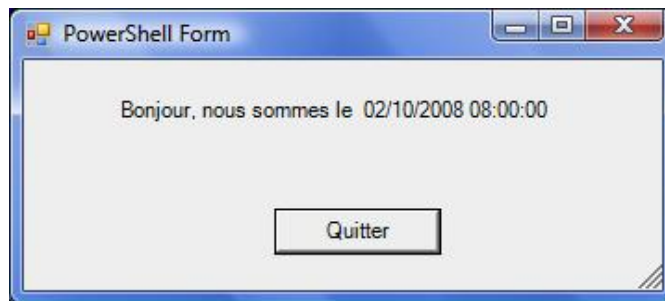
# Création d'un Label
$label1 = New-Object System.Windows.Forms.label
$label1.Location = New-Object System.Drawing.Point(50,20)
$label1.AutoSize = $true

# Création de l'objet Timer
$timer = New-Object System.Windows.Forms.Timer

$timer.Interval = 1000 # Définition de l'intervalle à 1 seconde
$timer.Add_Tick({
    $label1.Text = "Bonjour, nous sommes le $(Get-Date)"
})

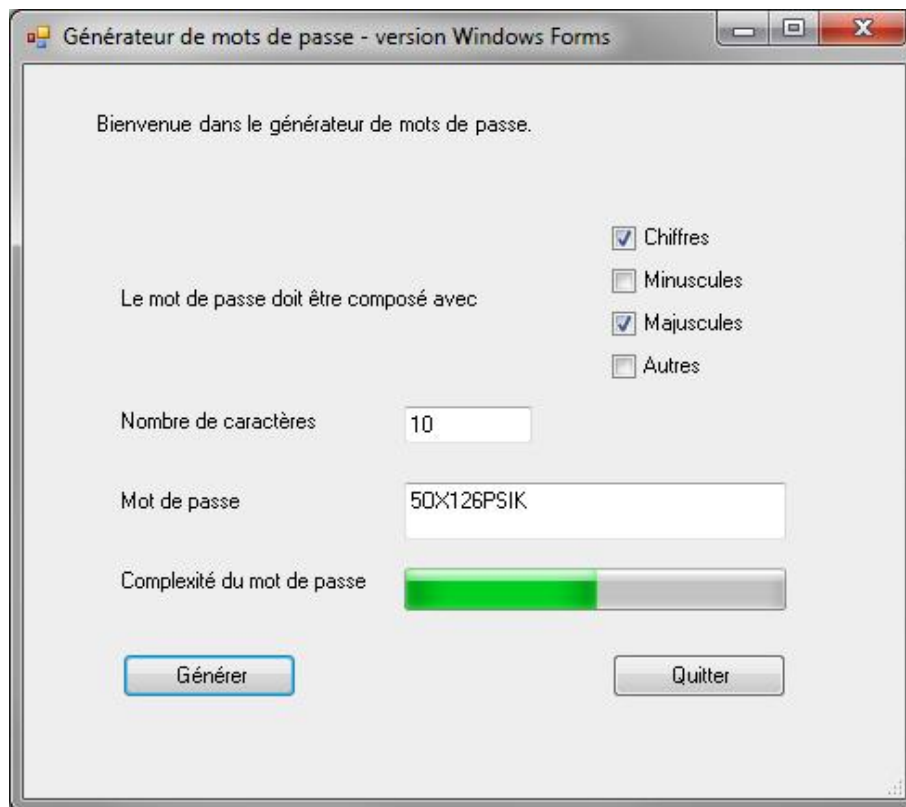
$timer.Start()
$form.Controls.Add($bouton_quitter)
$form.Controls.Add($label1)
$form.ShowDialog()
```

Résultat, notre interface affiche chaque seconde la date et l'heure actuelle.



Exemple : générateur de mots de passe complexes

Pour conclure sur la création d'interfaces graphiques, et de façon à mieux vous montrer les capacités graphiques de PowerShell grâce au Framework .NET, nous vous proposons un script reprenant quelques-uns des éléments décrits dans ce chapitre, comme l'insertion de composants à une form, tels que des cases à cocher, des zones de texte, etc. Le script que nous vous présentons crée une interface (cf. figure ci-après) qui vous permet de générer des mots de passe plus ou moins complexes via différents critères, comme sa composition et sa longueur.



Interface du générateur de mot de passe

Le script est le suivant :

Script : Script de génération de mots de passe.

```
# WinForms-pwdgenerator.ps1

[void][Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')

$textBox_resultat = New-Object System.Windows.Forms.TextBox
$progressBar = New-Object System.Windows.Forms.ProgressBar
$button_generer = New-Object System.Windows.Forms.Button
$checkBox_chiffres = New-Object System.Windows.Forms.CheckBox
$checkBox_minuscules = New-Object System.Windows.Forms.CheckBox
$checkBox_majuscules = New-Object System.Windows.Forms.CheckBox
$button_quitter = New-Object System.Windows.Forms.Button
$label1 = New-Object System.Windows.Forms.Label
$checkBox_autres = New-Object System.Windows.Forms.CheckBox
$label2 = New-Object System.Windows.Forms.Label
$label3 = New-Object System.Windows.Forms.Label
$textBox_Nb_caracteres = New-Object System.Windows.Forms.TextBox
$label4 = New-Object System.Windows.Forms.Label
$label_principal = New-Object System.Windows.Forms.Label
#
# textBox_resultat
#
$textBox_resultat.Location = New-Object System.Drawing.Point(205, 225)
$textBox_resultat.Multiline = $true
$textBox_resultat.Name = 'textBox_resultat'
$textBox_resultat.Size = New-Object System.Drawing.Size(206, 31)
$textBox_resultat.TabIndex = 2
#
# progressBar
#
$progressBar.Location = New-Object System.Drawing.Point(205, 271)
$progressBar.Name = 'progressBar'
$progressBar.Size = New-Object System.Drawing.Size(206, 23)
```



```

$progressBar.TabIndex = 3
$progressBar.set_forecolor('darkblue')
#
# button_generer
#
$button_generer.Location = New-Object System.Drawing.Point(53, 317)
$button_generer.Name = 'button_generer'
$button_generer.Size = New-Object System.Drawing.Size(94, 24)
$button_generer.TabIndex = 4
$button_generer.Text = 'G n rer'
$button_generer.UseVisualStyleBackColor = $true

$button_generer.Add_Click({
    [int]$len = $textBox_Nb_caracteres.get_text()
    $textBox_resultat.Text = ''
    $complex = 0
    $progressBar.Value = 0
    [string]$chars = ''

    if ($checkBox_chiffres.Checked)
        {$chars += '0123456789';$complex += 1}
    if ($checkBox_majuscules.Checked)
        {$chars += 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';$complex += 1}
    if ($checkBox_minuscules.Checked)
        {$chars += 'abcdefghijklmnopqrstuvwxyz';$complex += 1}
    if ($checkBox_autres.Checked)
        {$chars += '!@#%$';$complex += 1}

    if($chars -ne ''){
        $bytes = New-Object System.Byte[] $len
        $rnd =
        New-Object System.Security.Cryptography.RNGCryptoServiceProvider
        $rnd.GetBytes($bytes)
        $result = ''
        for( $i=0; $i -lt $len; $i++ )
        {
            $result += $chars[ $bytes[$i] % $chars.Length ]
        }
        $complex *= $(2.57*$len)
        if($complex -gt 100){ $complex = 100 }
        $progressBar.Value = $complex
        $textBox_resultat.Text = $result
    }
})
#
# checkBox_chiffres
#
$checkBox_chiffres.AutoSize = $true
$checkBox_chiffres.Location = New-Object System.Drawing.Point(317, 85)
$checkBox_chiffres.Name = 'checkBox_chiffres'
$checkBox_chiffres.Size = New-Object System.Drawing.Size(61, 17)
$checkBox_chiffres.TabIndex = 5
$checkBox_chiffres.Text = 'Chiffres'
$checkBox_chiffres.UseVisualStyleBackColor = $true
#
# checkBox_minuscules
#
$checkBox_minuscules.AutoSize = $true
$checkBox_minuscules.Location = New-Object System.Drawing.Point(317, 108)
$checkBox_minuscules.Name = 'checkBox_minuscules'
$checkBox_minuscules.Size = New-Object System.Drawing.Size(79, 17)
$checkBox_minuscules.TabIndex = 6
$checkBox_minuscules.Text = 'Minuscules'
$checkBox_minuscules.UseVisualStyleBackColor = $true
#
# checkBox_majuscules
#
$checkBox_majuscules.AutoSize = $true
$checkBox_majuscules.Location = New-Object System.Drawing.Point(317, 131)
$checkBox_majuscules.Name = 'checkBox_majuscules'

```

```

$checkBox_majuscules.Size = New-Object System.Drawing.Size(79, 17)
$checkBox_majuscules.TabIndex = 7
$checkBox_majuscules.Text = 'Majuscules'
$checkBox_majuscules.UseVisualStyleBackColor = $true
#
# button_quitter
#
$button_quitter.Location = New-Object System.Drawing.Point(317, 317)
$button_quitter.Name = 'button_quitter'
$button_quitter.Size = New-Object System.Drawing.Size(94, 24)
$button_quitter.TabIndex = 8
$button_quitter.Text = 'Quitter'
$button_quitter.UseVisualStyleBackColor = $true
$button_quitter.Add_Click({$Form1.Close()})
#
# label1
#
$label1.AutoSize = $true
$label1.Location = New-Object System.Drawing.Point(50, 271)
$label1.Name = 'label1'
$label1.Size = New-Object System.Drawing.Size(139, 13)
$label1.TabIndex = 9
$label1.Text = 'Complexité du mot de passe'
#
# checkBox_autres
#
$checkBox_autres.AutoSize = $true
$checkBox_autres.Location = New-Object System.Drawing.Point(317, 154)
$checkBox_autres.Name = 'checkBox_autres'
$checkBox_autres.Size = New-Object System.Drawing.Size(56, 17)
$checkBox_autres.TabIndex = 10
$checkBox_autres.Text = 'Autres'
$checkBox_autres.UseVisualStyleBackColor = $true
#
# label2
#
$label2.AutoSize = $true
$label2.Location = New-Object System.Drawing.Point(50, 119)
$label2.Name = 'label2'
$label2.Size = New-Object System.Drawing.Size(227, 15)
$label2.TabIndex = 11
$label2.Text = 'Le mot de passe doit être composé avec'
#
# label3
#
$label3.AutoSize = $true
$label3.Location = New-Object System.Drawing.Point(50, 185)
$label3.Name = 'label3'
$label3.Size = New-Object System.Drawing.Size(129, 15)
$label3.TabIndex = 12
$label3.Text = 'Nombre de caractères'
#
# textBox_Nb_caracteres
#
$textBox_Nb_caracteres.Location = New-Object System.Drawing.Point(205, 184)
$textBox_Nb_caracteres.Name = 'textBox_Nb_caracteres'
$textBox_Nb_caracteres.Size = New-Object System.Drawing.Size(69, 20)
$textBox_Nb_caracteres.TabIndex = 13
$textBox_Nb_caracteres.Text = '10'
#
# label4
#
$label4.AutoSize = $true
$label4.Location = New-Object System.Drawing.Point(50, 228)
$label4.Name = 'label4'
$label4.Size = New-Object System.Drawing.Size(71, 13)
$label4.TabIndex = 14
$label4.Text = 'Mot de passe'
#
# label_principal

```

```
#
$label_principal.AutoSize = $true
$label_principal.Location = New-Object System.Drawing.Point(37, 25)
$label_principal.Name = 'label_principal'
$label_principal.Size = New-Object System.Drawing.Size(355, 20)
$label_principal.TabIndex = 15
$label_principal.Text = 'Bienvenue dans le générateur de mots de passe.'
#
$Form1 = New-Object System.Windows.Forms.form
# Form1
#
$Form1.ClientSize = New-Object System.Drawing.Size(475, 395)
$Form1.Controls.Add($label_principal)
$Form1.Controls.Add($label4)
$Form1.Controls.Add($textBox_Nb_caracteres)
$Form1.Controls.Add($label3)
$Form1.Controls.Add($label2)
$Form1.Controls.Add($checkBox_autres)
$Form1.Controls.Add($label1)
$Form1.Controls.Add($button_quitter)
$Form1.Controls.Add($checkBox_majuscules)
$Form1.Controls.Add($checkBox_minuscules)
$Form1.Controls.Add($checkBox_chiffres)
$Form1.Controls.Add($button_generer)
$Form1.Controls.Add($progressBar)
$Form1.Controls.Add($textBox_resultat)
$Form1.Name = 'Form1'
$Form1.Text = 'Générateur de mots de passe - version Windows Forms'
$Form1.ShowDialog()
```

d. Le convertisseur de forms

PowerShell ne disposant pas d'éditeur graphique permettant de générer des interfaces, tout doit se faire en ligne de commandes. Et ceci est particulièrement long et fastidieux, surtout au moment de soigner la position des composants.

Pour répondre à ce besoin, vous trouverez sur notre site Internet www.powershell-scripting.com un script répondant au nom de CSForm2PS.ps1 capable de transcrire une form graphique développée avec le logiciel Visual C# en un script PowerShell.



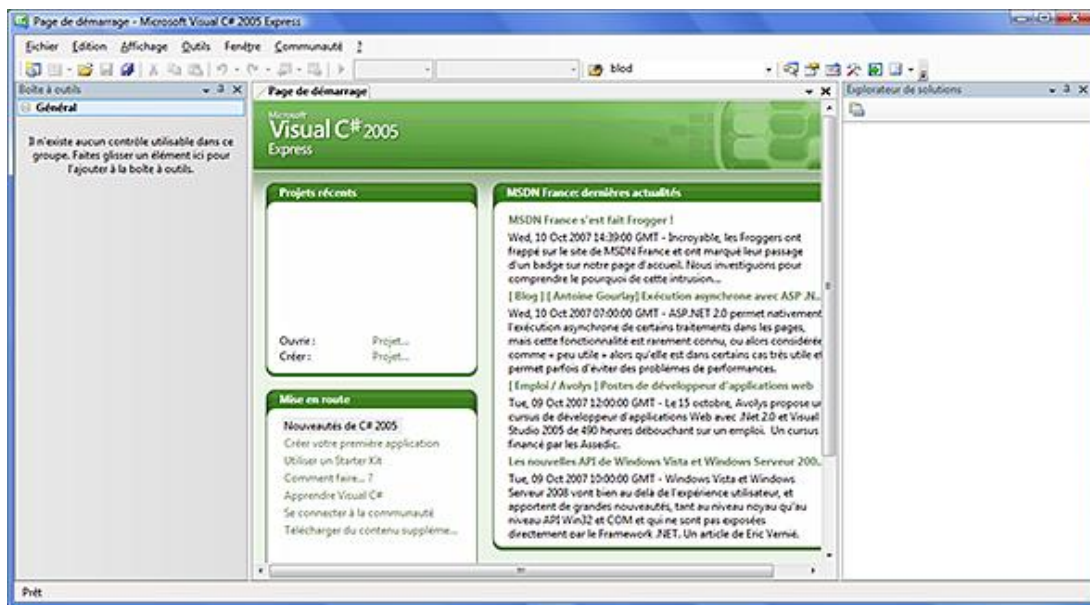
Ce script convertisseur de forms n'est pas un éditeur graphique, mais permet de passer du C# au PowerShell par transformation des lignes de code. L'utilisation du convertisseur se limite à la conversion des composants graphiques. Il ne prend pas en charge les événements ni le paramétrage spécifique des composants.

Simple et rapide, vous ne perdrez plus un temps conséquent à mesurer la taille et l'emplacement de chaque composant. Maintenant quelques clics de souris sous Visual C# suffiront.



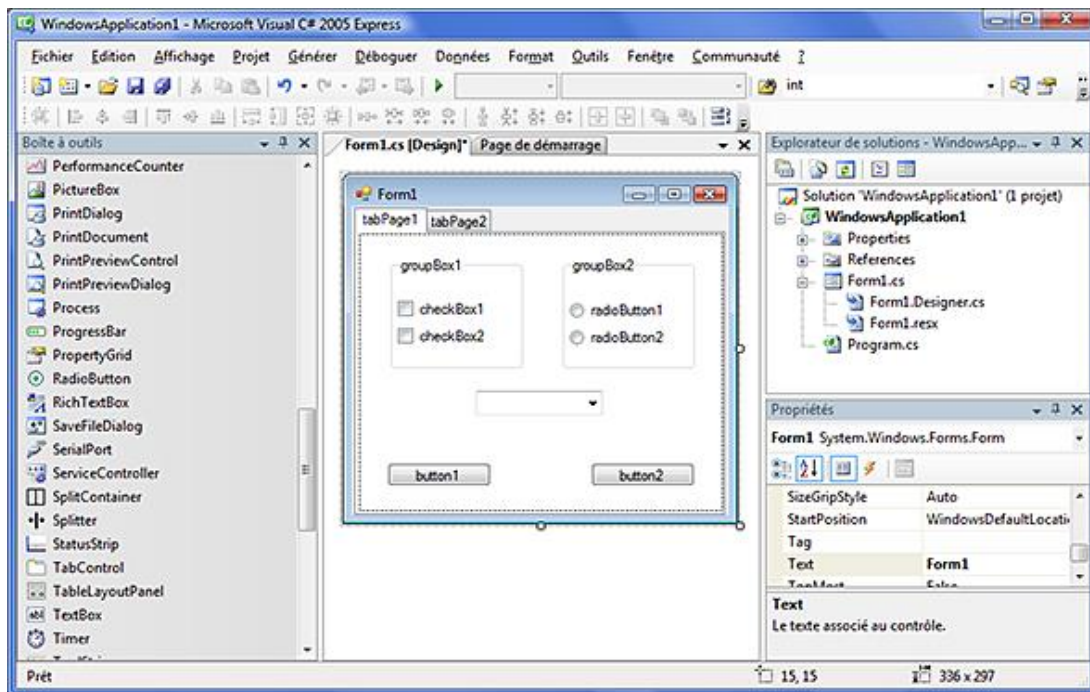
Si vous ne souhaitez pas acquérir une licence pour le logiciel Visual C#, sachez que ce dernier est téléchargeable gratuitement dans sa version Express. Version téléchargeable sur Internet et qui suffit amplement pour la réalisation de forms que vous pourrez ensuite convertir en PowerShell.

- Pour créer une form graphique, lancez Visual C#, et choisissez de créer un nouveau projet en cliquant sur **Fichier** puis **Nouveau projet**.



Fenêtre de démarrage de l'application Visual C#

Après avoir choisi le modèle **Application Windows**, il vous est possible de modifier la form qui est créée par défaut et de lui ajouter d'autres composants disponibles dans la barre d'outils, voir figure ci-après.



Création d'une interface sous Visual C#



Attention, le convertisseur de formulaire ne permet pas de traiter tous les types de composants. Pour plus d'informations sur les composants acceptés, rendez-vous sur le site www.powershell-scripting.com.

Après avoir défini l'interface selon vos besoins, enregistrez le projet sous le nom de votre choix et récupérez le fichier `<nom du formulaire>.Designer.cs` (par défaut, ce fichier se nomme `form1.Designer.cs`).

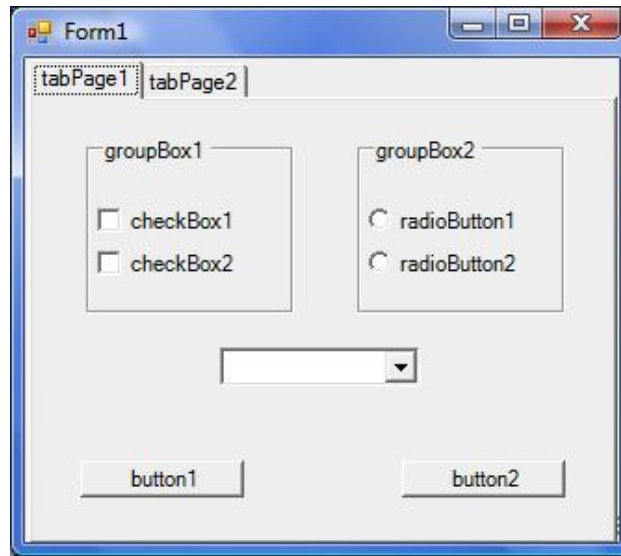
Il ne reste plus qu'à utiliser le script `CSForm2PS.ps1` avec les paramètres `-source` et `-dest` qui spécifient respectivement le nom du fichier créé avec Visual C#, et le nom du script PowerShell que vous souhaitez générer.

Exemple :

```
PS > ./CSForm2PS.ps1 -source <Fichier *.designer.cs> -dest <Fichier.ps1>
```

Le fichier *.designer.cs est ainsi transformé pour devenir un script PowerShell.

Exemple d'un résultat graphique obtenu en exécutant un script généré par **CSForm2PS.ps1** :

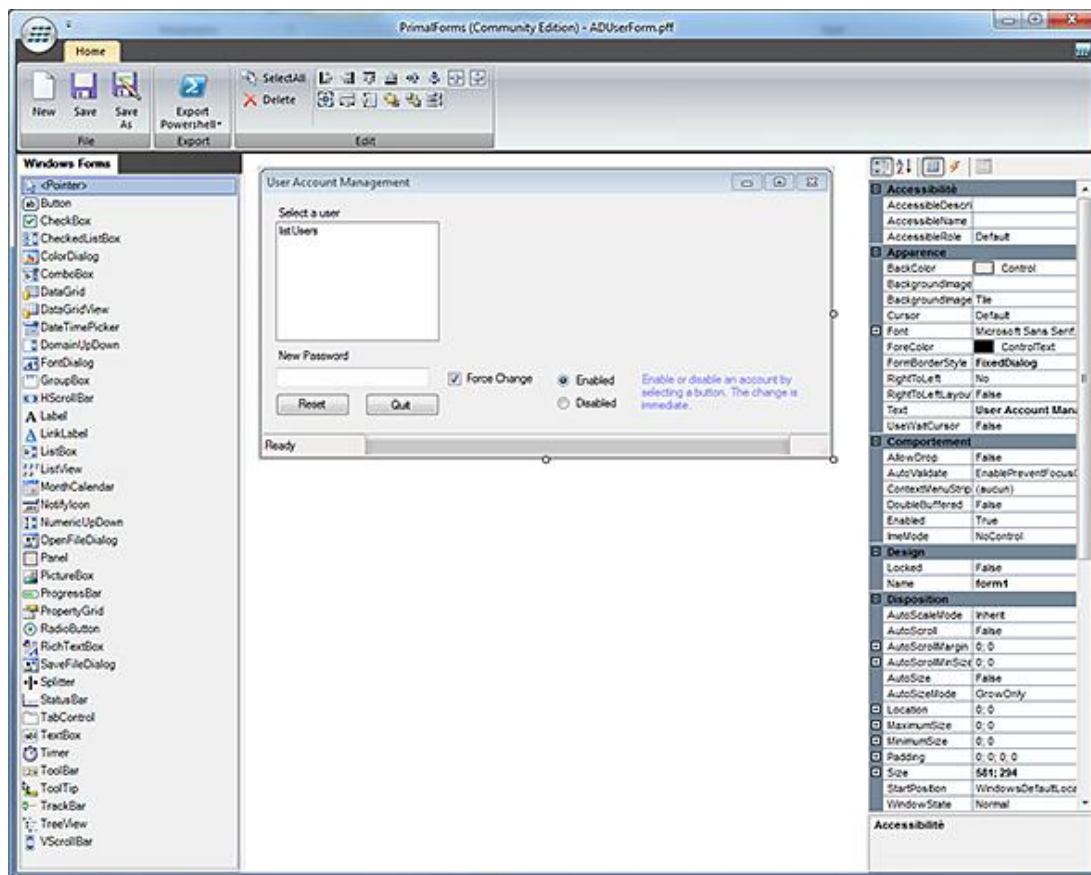


Le convertisseur de forms était l'outil ultime pour la création d'interfaces de type Windows Forms jusqu'à ce que l'outil **Primal Forms** vienne le détrôner. Ceci étant, il valait tout de même la peine de mentionner son existence, ne serais-ce que pour mettre en avant la prouesse technique qui consiste à convertir à la volée un code C# en un code PowerShell. Preuve que finalement ces deux langages ne sont pas si éloignés que cela...

e. Sapien Primal Forms Community Edition

PrimalForms Community Edition de la société Sapien Technologies (<http://www.primaltools.com/downloads/>) est un outil gratuit qui permet de créer de superbes interfaces Windows Forms en un temps record !

Avec PrimalForms, vous avez accès à la liste des composants graphiques (tels que les listes déroulantes, les zones de saisie de texte, les cases à cocher, etc.) et il suffit, à l'instar d'un Visual Studio, de *glisser déplacer* des composants sur le formulaire pour les voir apparaître. Une fois l'interface créée, il n'y a plus qu'à l'exporter soit dans le Presse-papiers, soit directement dans un fichier PS1.



Primal Forms Community Edition

Comme vous pouvez vous en rendre compte, un tel outil nous évite d'avoir à écrire de nombreuses lignes de script. C'est une aide inestimable en termes de gain de temps lors de la définition des interfaces.

Ensuite, une fois le script PowerShell généré il faut le modifier pour ajouter les événements tels que les actions à réaliser lors d'un clic sur l'un des boutons de l'interface.

Voici un extrait du script exporté :

```
#-----
#Generated Event Script Blocks
#-----
#Provide Custom Code for events specified in PrimalForms.
...
$handler_button1_Click=
{
    #TODO: Place custom script here
}
$handler_listBox1_SelectedIndexChanged=
{
    #TODO: Place custom script here
}
#-----
#region Generated Form Code
$button1.TabIndex = 2

$button1.Name = "button1"
$System_Drawing_Size = New-Object System.Drawing.Size
$System_Drawing_Size.Width = 75
$System_Drawing_Size.Height = 23
$button1.Size = $System_Drawing_Size
$button1.UseVisualStyleBackColor = $True
$button1.Text = "Commander"
$System_Drawing_Point = New-Object System.Drawing.Point
$System_Drawing_Point.X = 45
$System_Drawing_Point.Y = 245
$button1.Location = $System_Drawing_Point
```

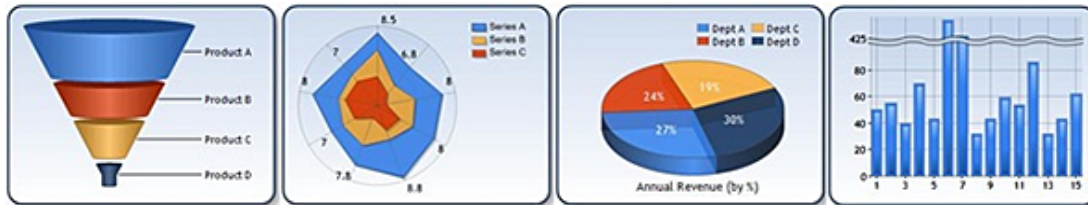
```
$button1.DataBindings.DefaultDataSourceUpdateMode = 0
$button1.add_Click($handler_button1_Click)

$form.Controls.Add($button1)
```

Alors que **PrimalForms Community Edition** est gratuit, sachez qu'il existe une version payante nommée sobrement **PrimalForms 2009**. La version payante donne accès à de nouvelles fonctionnalités telles que : un éditeur de scripts avec coloration syntaxique, l'aide en ligne contextuelle (à la *IntelliSense*), un explorateur d'objets .NET, et un outil incroyable qui fait de **PrimalForms** un produit unique : un packageur.

Le packageur permet de transformer n'importe quel script PowerShell en un exécutable. Il est même possible d'indiquer un compte et un mot de passe d'un utilisateur privilégié pour exécuter des scripts avec des droits différents de l'utilisateur. Seule contrainte : avoir PowerShell installé sur les postes qui font appel aux scripts transformés en exécutables.

f. Création de graphiques avec les MS Charts Controls



MS Charts Controls for .NET Framework 3.5 en action

MS Charts Controls for .NET Framework 3.5 est une bibliothèque d'objets graphiques, fournie gratuitement par Microsoft, conçue pour les technologies **Windows Forms** et **ASP.NET**. Ces objets, appelés contrôles, sont très agréables à l'œil et sont par conséquent très bien adaptés pour présenter des résultats.

Le seul pré-requis nécessaire est le Framework .NET 3.5 SP1, et bien sur PowerShell (v1 ou v2).

Voici le lien où télécharger cette bibliothèque (1.8 Mo) : <http://www.microsoft.com/downloads/details.aspx?displaylang=fr&FamilyID=130f7986-bf49-4fe5-9ca8-910ae6ea442c>

Microsoft Chart Controls for Microsoft .NET Framework 3.5 installera de nouvelles assemblées qui contiennent les contrôles graphiques ASP.NET et Windows Forms. L'installation est on ne peut plus simple : elle se résume à cliquer sur **Suivant**, **Suivant** et **Terminer**.

Prenons un exemple pour tenter d'afficher sous forme de barregraphe 3D les cinq processus les plus consommateurs de mémoire.

Pour commencer, nousinstancions un contrôle graphique de type Chart ou plus précisément de type System.Windows.Forms.DataVisualization.Charting.Chart. Et nous lui donnons une certaine dimension, comme ceci :

```
# Creation de l'objet Chart
$Chart = New-Object System.Windows.Forms.DataVisualization.Charting.Chart
$Chart.Width = 500
$Chart.Height = 400
$Chart.Left = 40
$Chart.Top = 30
```

Nous pouvons aussi en profiter pour donner un titre à notre graphique :

```
[void]$Chart.Titles.Add('Top 5 des processus les plus consommateurs de mémoire')
```

Nous définissons ensuite la propriété ChartAreas du graphique comme ceci :

```
$ChartArea =
New-Object System.Windows.Forms.DataVisualization.Charting.ChartArea
$ChartArea.AxisX.Title = 'Processus (PID)'
$ChartArea.AxisY.Title = 'Mémoire paginée utilisée (en Mo)'
$Chart.ChartAreas.Add($ChartArea)
```

Grâce à cette propriété nous définissons des légendes sur les axes X et Y.

Ensuite arrive la génération des données. L'objectif pour arriver à nos fins est de passer deux variables de type tableau à notre objet graphique. Sur l'axe X nous passerons le tableau qui contient les noms des processus et sur l'axe Y le tableau de valeurs associées.

Pour ce faire nous commençons par collecter les données :

```
$Processes = Get-Process | Sort-Object -Property PM | Select-Object -Last 5
```

On récupère ainsi 5 objets de type processus triés par valeur croissante de la propriété PM (*Paged Memory* - mémoire paginée). Si l'on affichait le contenu de la variable \$Processes, voici ce que l'on pourrait obtenir :

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
1255	60	83040	44456	301	62,37	5256	wlmail
213	12	111316	204448	327	317,52	3908	dwm
617	29	116308	186784	427	374,57	7776	powershell_ise
807	34	138816	84240	1076	321,71	4712	iexplore
1203	39	203736	92868	471	56,71	3952	explorer

Dans la variable \$ProcNames nous mettons le nom des processus ainsi que leur numéro d'identifiant entre parenthèses et nous convertissons le tout en tableau grâce à l'arobase :

```
$ProcNames = @(foreach($Proc in $Processes){$Proc.Name + '(' + $Proc.ID + ')'})
```

La ligne de script ci-dessus est la forme condensée de :

```
$ProcNames = @(
    foreach($Proc in $Processes)
    {
        $Proc.Name + '(' + $Proc.ID + ')'
    }
)
```

\$ProcNames contient à présent les valeurs suivantes :

```
wlmail(5256)
dwm(3908)
powershell_ise(7776)
iexplore(4712)
explorer(3952)
```

Dans la variable \$PM nous stockons les valeurs correspondantes aux propriétés PM de nos processus et convertissons le résultat en tableau :

```
$PM = @(foreach($Proc in $Processes){$Proc.PM/1MB})
```

Afin d'obtenir un résultat plus compréhensible nous convertissons les valeurs en Mo grâce au quantificateur d'octets MB. Voici le contenu de la variable \$PM :

```
81,09375
108,70703125
113,58203125
135,5625
198,9609375
```

Nous passons les tableaux de valeur à la propriété Series sous la forme suivante :

```
[void]$Chart.Series.Add('Data')
$Chart.Series['Data'].Points.DataBindXY($ProcNames, $PM)
```

Nous définissons l'apparence du graphique (colonnes 3D) :

```
# Définition du type de graphique (colonnes)
$Chart.Series['Data'].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Column

# Définition du style de graphique (cylindres 3D)
$Chart.Series['Data']['DrawingStyle'] = 'Cylinder'
```

En profitons pour donner un peu de couleurs aux valeurs mini et maxi :

```
# Trouve les valeurs mini et maxi et application des couleurs
```



```
$maxValuePoint = $Chart.Series['Data'].Points.FindMaxByValue()
$maxValuePoint.Color = [System.Drawing.Color]::Red
$minValuePoint = $Chart.Series['Data'].Points.FindMinByValue()
$minValuePoint.Color = [System.Drawing.Color]::Green
```

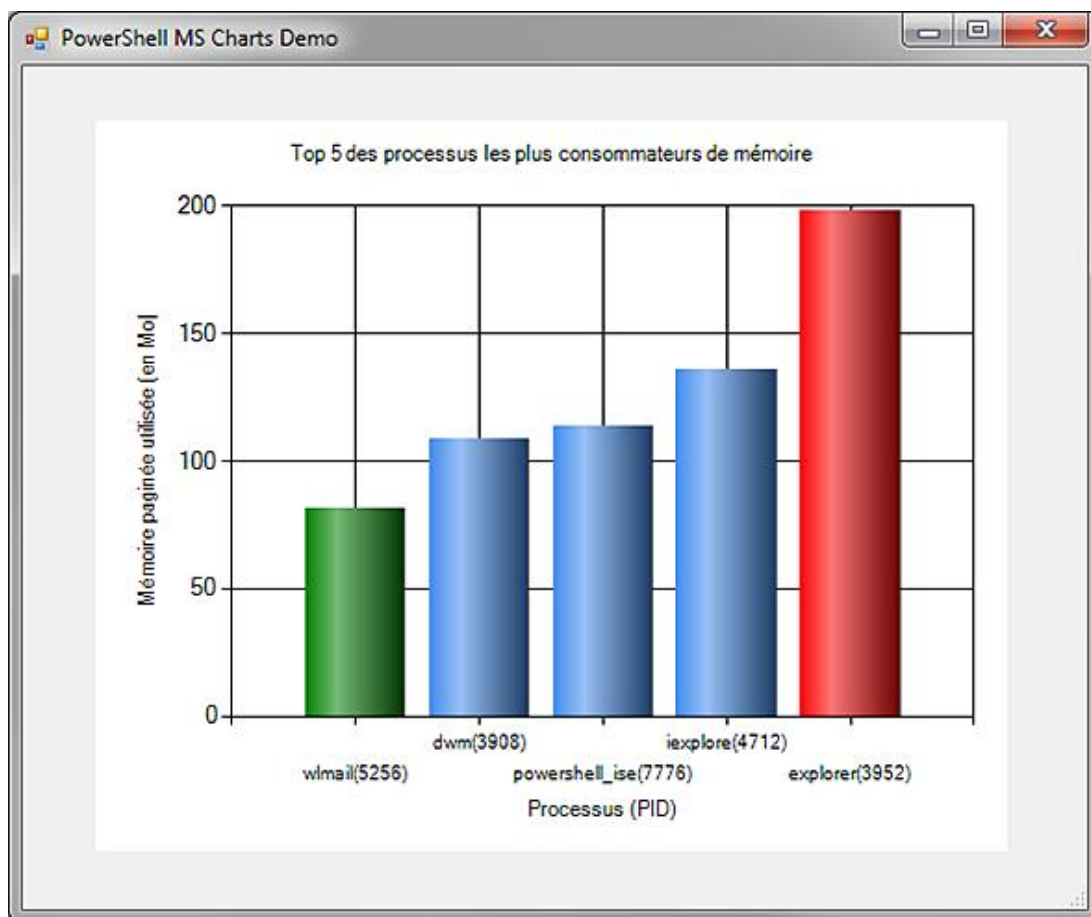
Et enfin créons la form, avec un nom, sa taille et lui affectons l'objet correspondant à notre graphique :

```
# Création du formulaire Windows Forms
$Form = New-Object Windows.Forms.Form
$Form.Text = 'PowerShell MS Charts Demo'
$Form.Width = 600
$Form.Height = 500
$Form.Controls.Add($Chart)
```

Puis pour terminer affichons le chef d'œuvre :

```
# Affichage du formulaire
$Form.Add_Shown({$Form.Activate()})
$Form.ShowDialog()
```

Et nous obtenons le résultat suivant :



Barregraphe 3D avec les MS Charts Controls

Voici le script dans son intégralité :

```
# Barregraphe 3D avec les MS Charts Controls
# Chargement des assemblies
[void][Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
[void][Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms.
DataVisualization')

# Creation de l'objet Chart
$Chart = New-Object System.Windows.Forms.DataVisualization.Charting.Chart
$Chart.Width = 500
$Chart.Height = 400
```

```

$Chart.Left = 40
$Chart.Top = 30
# Ajout d'un titre
[void]$Chart.Titles.Add('Top 5 des processus les plus consommateurs
de mémoire')

# Création d'une zone de dessin et ajout de l'objet Chart à cette zone
$ChartArea = New-Object System.Windows.Forms.DataVisualization.Charting.
ChartArea
$ChartArea.AxisX.Title = 'Processus (PID)'
$ChartArea.AxisY.Title = 'Mémoire paginée utilisée (en Mo)'
$Chart.ChartAreas.Add($ChartArea)

# Ajout des données
$Processes = Get-Process | Sort-Object -Property PM | Select-Object -Last 5
$ProcNames = @(foreach($Proc in $Processes){$Proc.Name + '(' + $Proc.ID + ')'})
$PM = @(foreach($Proc in $Processes){$Proc.PM/1MB})
[void]$Chart.Series.Add('Data')
$Chart.Series['Data'].Points.DataBindXY($ProcNames, $PM)

# Définition du type de graphique (colonnes)
$Chart.Series['Data'].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Column

# Définition du style de graphique (cylindres 3D)
$Chart.Series['Data']['DrawingStyle'] = 'Cylinder'

# Trouve les valeurs mini et maxi et application des couleurs
$maxValuePoint = $Chart.Series['Data'].Points.FindMaxByValue()
$maxValuePoint.Color = [System.Drawing.Color]::Red
$minValuePoint = $Chart.Series['Data'].Points.FindMinByValue()
$minValuePoint.Color = [System.Drawing.Color]::Green

# Création du formulaire Windows Forms
$Form = New-Object Windows.Forms.Form
$Form.Text = 'PowerShell MS Charts Demo'
$Form.Width = 600
$Form.Height = 500
$Form.controls.Add($Chart)

# Affichage du formulaire
$Form.Add_Shown({$Form.Activate()})
$Form.ShowDialog()

```

Finalement le résultat serait peut-être mieux compris s'il était présenté sous forme de camembert.

Remplacez les lignes suivantes :

```

$Chart.Series['Data'].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Column
$Chart.Series['Data']['DrawingStyle'] = 'Cylinder'

```

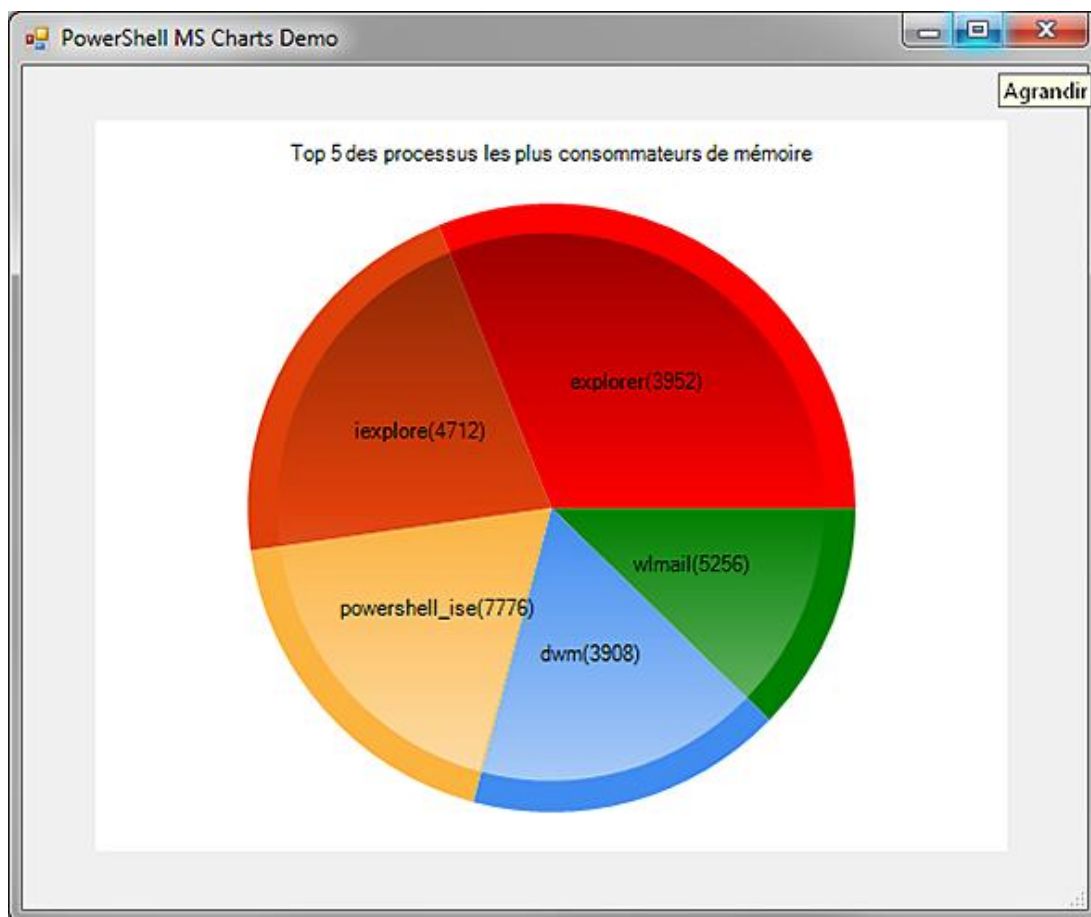
Par celles-ci :

```

$Chart.Series['Data'].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::pie
$Chart.Series['Data']['PieLabelStyle'] = 'inside'
$Chart.Series['Data']['PieDrawingStyle'] = 'concave'

```

Et voilà !



Camembert 3D avec les MS Charts Controls

g. Bibliothèque LibraryChart

La bibliothèque **LibraryChart**, créée par *Chad Miller*, que l'on peut trouver à cette adresse : <http://poshcode.org/1330> a pour objectif de simplifier l'utilisation des **MS Charts Controls**. **LibraryChart** propose les fonctionnalités suivantes :

- Envoi du résultat d'une ou plusieurs commande(s) PowerShell via le pipeline directement dans un graphique,
- Affichage des graphiques dans une **Windows Form**,
- Sauvegarde des graphiques en tant qu'image,
- Choix varié de graphiques : barres horizontales, colonnes, camembert, etc.
- Mise à jour des graphiques en temps réel grâce à l'utilisation de blocs de scripts ; ces derniers s'exécutant à un intervalle de temps donné,
- Compatibilité PowerShell v1

Une fois la librairie téléchargée, il suffit de la charger comme ceci :

```
PS >. ./libraryChart.ps1
```

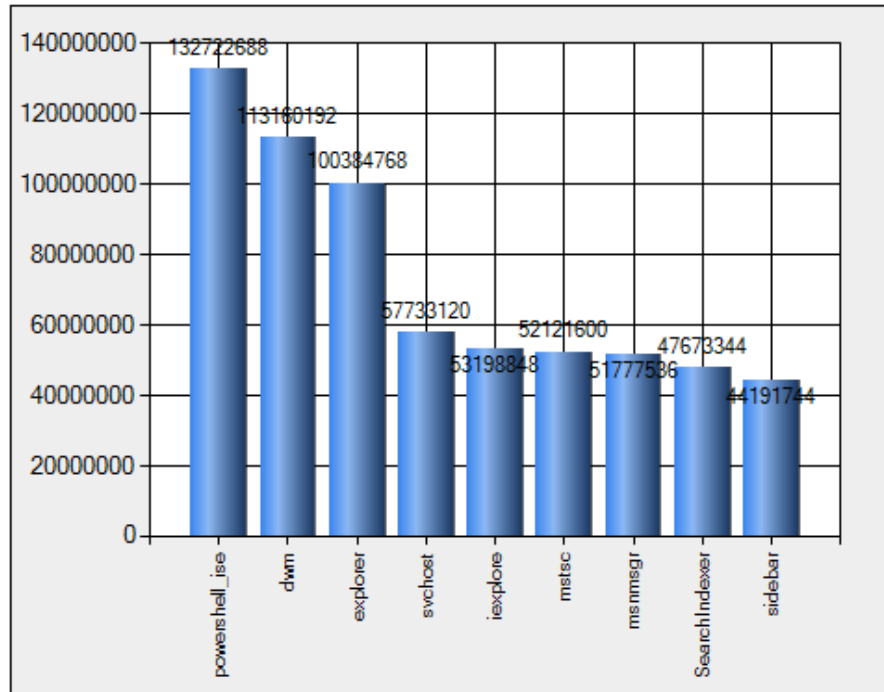
N'oubliez pas de bien mettre un point suivi d'un espace avant l'appel du script (DotSourcing - voir chapitre Fondamentaux) sinon la librairie ne sera pas importée dans la session de script (l'étendue) courante. Par conséquent les fonctions de la librairie ne seront pas chargées en mémoire.

Une fois ceci fait, nous pouvons commencer à créer des graphiques très simplement.

Exemple :

Affichage en mode colonnes des 10 processus occupant le plus de mémoire paginée.

```
PS > Get-Process | Sort-Object -Property PM | Select-Object -Last 10 |  
Out-Chart -x Name -y PM
```

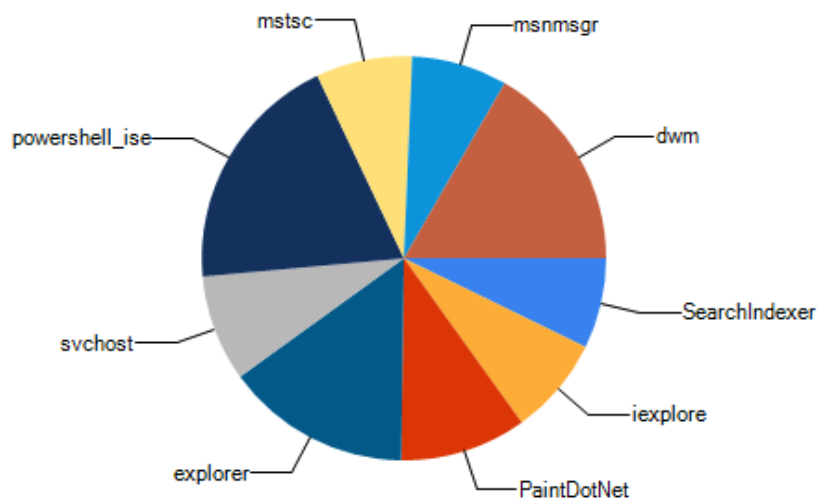


Barregraphe avec les MS Charts Controls et LibraryChart

Exemple :

Affichage en mode camembert des 10 processus occupant le plus de mémoire paginée.

```
PS > Get-Process | Sort-Object -Property PM | Select-Object -Last 10 |  
Out-Chart -x Name -y PM -chartType pie
```



Camembert avec les MS Charts Controls et LibraryChart

Avouez que le rendement lignes de script/résultat obtenu est plus que rentable... Si les effets 3D vous manquent ou d'autres fonctionnalités, dans ce cas il vous suffira de modifier le script PowerShell correspondant à la librairie.

3. Windows Presentation Foundation

Les interfaces graphiques **WPF** représentent, des points de vue de Microsoft et de la communauté des développeurs, la voie de l'avenir. Microsoft est clairement en train d'investir (depuis 2006) dans **WPF** et non plus dans **Windows Forms**, en tant que future plate-forme de présentation.

Il y a de nombreuses raisons à cela ; tout d'abord l'aspect vectoriel de **WPF** fait que les interfaces graphiques **WPF** sont indépendantes de la résolution des écrans. Ainsi une interface créée sur un écran 14 pouces, aura le même rendu que sur un écran 50 pouces. En outre, **WPF** étant une technologie nouvelle, elle sait tirer partie de l'accélération matérielle en s'appuyant sur l'API Direct 3D. Enfin, les éléments qui composent **WPF** sont d'une incroyable richesse et d'une grande flexibilité d'utilisation. Par exemple, même si cela n'est pas forcément recommandé, il serait aisé de créer une boîte déroulante remplie d'animations 2D ou de clips vidéo.

Mais **WPF** a également l'immense avantage par rapport aux **Windows Forms** de disposer du langage XAML (prononcer « *gzamel* »). Grâce à XAML, l'interface graphique va à présent non plus être construite en PowerShell via l'appel à des méthodes .NET (même si cela reste toujours possible), mais décrite avec une grammaire XML. Ceci est un point très important pour la maintenance des scripts avec interface graphique. Alors qu'avec les **Windows Forms** l'interface fait partie intégrante du script, il n'est par conséquent pas facile de la modifier. En effet, la logique du script et la définition de l'interface étant mélangées. À présent, avec **WPF** l'interface graphique est stockée dans un fichier externe, ce qui permettra de pouvoir la modifier ultérieurement avec les outils adéquats sans avoir à toucher au script PowerShell qui y fait appel.

On peut néanmoins, si on le désire, intégrer le code XAML dans un script PowerShell en le stockant dans une variable de type *Here String*. Ce qui dans certains cas de figure peut être pratique afin d'éviter des dépendances entre fichiers pour avoir un script PowerShell qui se suffit à lui-même.

a. Avant de commencer...

Avant de se lancer à corps et à cri dans le scripting avec **WPF**, vous devez savoir qu'il y a quelques pré-requis. Le premier, vous le connaissez déjà, est qu'il faut utiliser au minimum la version 2 de PowerShell. Le second, est qu'il faut soit :

- Exécuter le script dans PowerShell ISE (*Integrated Scripting Environment*), l'éditeur graphique PowerShell,
- Exécuter le script dans la console PowerShell, mais en ayant pris soin de lancer cette dernière avec le paramètre -STA, soit : PowerShell.exe -STA

Si vous avez opté pour la seconde méthode, alors il vous faudra toujours commencer vos scripts par le chargement des assemblées PresentationFramework, PresentationCore, et WindowsBase comme ci-après :

```
Add-Type -assemblyName PresentationFramework
Add-Type -assemblyName PresentationCore
Add-Type -assemblyName WindowsBase
```

Notez que si vous utilisez PowerShell ISE le chargement des assemblées n'est pas nécessaire car c'est déjà fait pour vous ; PowerShell ISE étant elle-même une application **WPF**. Dans les exemples qui suivront, afin de limiter le nombre de lignes de script, nous omettrons de charger les assemblées. N'oubliez donc pas d'ajouter ces quelques lignes au début de vos scripts si vous les lancez à partir de la console PowerShell classique.

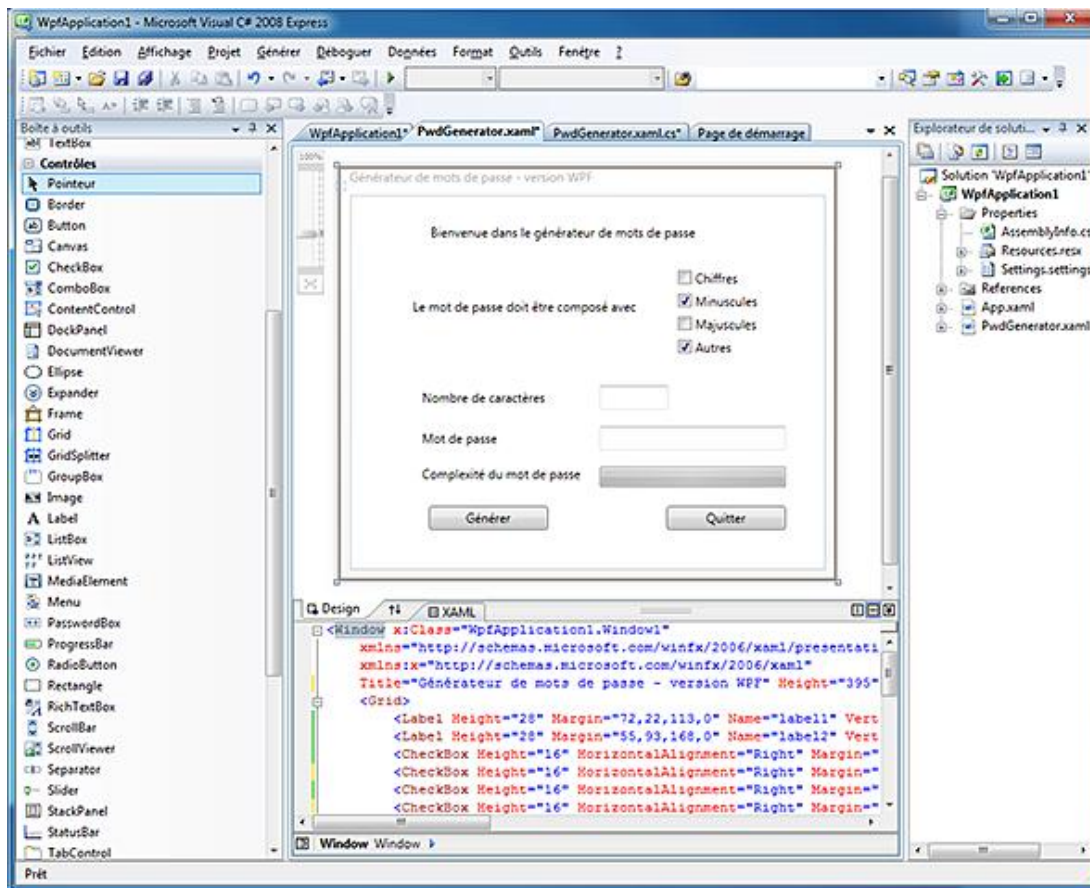
b. Création assistée d'interfaces graphiques WPF

WPF, par rapport à PowerShell est une technologie récente, en effet **WPF** n'est utilisable que depuis la version 2 de PowerShell. Par conséquent, il n'existe pas encore d'outil intégré tel que **PrimalForms** pour bâtir des interfaces graphiques. Cela fait donc de nous, en quelque sorte, des aventuriers... Et en tant que tel, nous allons devoir ruser un peu.

Premièrement, même si cela peut rebuter quelques administrateurs systèmes, nous allons devoir utiliser un outil pour concevoir notre interface graphique. Et pour ce faire, notre choix s'est porté sur **WPF Designer** inclus dans la suite de développement **Visual Studio 2008**. **Visual Studio 2008** se décline en différentes versions ; pour notre usage **Visual C# 2008 Express Edition** (disponible à l'adresse <http://www.microsoft.com/express/vcsharp>) fera parfaitement l'affaire ; il s'agit de la version gratuite.

Une fois **Visual C# 2008 Express Edition** installé, lancez-le, allez dans le menu Fichier/Nouveau projet... et sélectionnez Application WPF. À présent, vous venez de lancer le **WPF Designer** ; et de façon tout à fait classique vous n'avez plus qu'à construire votre interface en glissant déposant des éléments graphiques.

Par exemple, essayons de refaire le générateur de mots de passe qui nous a servi d'exemple pour illustrer les formulaires Windows Forms :



Conception d'une interface graphique WPF dans WPF Designer de Visual Studio 2008.

Vous pouvez voir dans le volet inférieur de la fenêtre, le code XAML correspondant à notre interface. Une fois votre interface terminée, il suffit de copier/coller le contenu de ce volet dans notre script PowerShell et la partie sera déjà à moitié gagnée !

Vous pouvez aussi sauvegarder votre travail et aller chercher dans le répertoire de sauvegarde le fichier portant l'extension .xaml pour copier son contenu.

Vous devriez obtenir un code XAML ressemblant à celui-ci :

```
<Window x:Class="WpfApplication1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Générateur de mots de passe - version WPF" Height="395"
Width="475">
  <Grid>
    <Label Height="28" Margin="72,22,113,0" Name="label1"
VerticalAlignment="Top">Bienvenue dans le générateur de mots
de passe</Label>
    <Label Height="28" Margin="55,93,168,0" Name="label2"
VerticalAlignment="Top">Le mot de passe doit être composé avec</Label>
    <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,71,20,0" Name="checkBox1" VerticalAlignment="Top"
Width="120" IsChecked="True">Chiffres</CheckBox>
    <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,137,20,0" Name="checkBox4" VerticalAlignment="Top"
Width="120">Autres</CheckBox>
    <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,115,20,0" Name="checkBox3" VerticalAlignment="Top"
Width="120" IsChecked="True">Majuscules</CheckBox>
    <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,93,20,0" Name="checkBox2" VerticalAlignment="Top"
Width="120">Minuscules</CheckBox>
    <Label Height="28" HorizontalAlignment="Left"
Margin="64,0,0,149" Name="label3" VerticalAlignment="Bottom"
Width="149">Nombre de caractères</Label>
```

```

        <Label Height="28" HorizontalAlignment="Left"
Margin="64,0,0,110" Name="label4" VerticalAlignment="Bottom"
Width="149">Mot de passe</Label>
        <Label Height="28" Margin="0,0,221,76" Name="label5"
VerticalAlignment="Bottom" HorizontalAlignment="Right"
Width="168">Complexité du mot de passe</Label>
        <TextBox Height="23" HorizontalAlignment="Right"
Margin="0,0,149,152" Name="textBox1" VerticalAlignment="Bottom"
Width="66" />
        <TextBox Height="23" HorizontalAlignment="Right"
Margin="0,0,37,113" Name="textBox2" VerticalAlignment="Bottom"
Width="178" />
        <ProgressBar Height="20" HorizontalAlignment="Right"
Margin="0,0,37,78" Name="progressBar1" VerticalAlignment="Bottom"
Width="178" />
        <Button Height="23" HorizontalAlignment="Left" Margin="75,0,0,37"
Name="button1" VerticalAlignment="Bottom" Width="114">Générer</Button>
        <Button Height="23" HorizontalAlignment="Right" Margin="0,0,37,37"
Name="button2" VerticalAlignment="Bottom" Width="114">Quitter</Button>
    </Grid>
</Window>

```

Pour avoir un code XAML utilisable avec PowerShell, enlevez simplement sur la première ligne « x:Class="WpfApplication1.Window1" ».



Veuillez noter la déclaration des espaces de noms XML grâce à la syntaxe suivante : même si cette syntaxe peut sembler un peu barbare, elle est de rigueur avec la norme XML. Ces deux lignes sont nécessaires car elles indiquent que le fichier se conforme à la grammaire XAML.

```

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

```

À présent pour utiliser notre interface, il nous faut créer une *Here String* qui contiendra le code XAML, puis convertir cette dernière en un objet XML. Pour ce faire seules quatre lignes de PowerShell sont nécessaires. Observez l'exemple ci-dessous :

```

[xml]$xaml = @'

Insérer ici le code XAML

'@
$reader=New-Object System.Xml.XmlNodeReader $xaml
$Form=[Windows.Markup.XamlReader]::Load($reader)

$Form.ShowDialog() | Out-Null

```

Avec ce script nous pouvons d'ores et déjà afficher notre interface. Ceci étant elle ne sera pas fonctionnelle tant que nous n'aurons pas scripté les actions à réaliser, telles que le clic sur les boutons.

Nous pourrions aussi très simplement externaliser dans un fichier texte toute la partie XAML, la charger avec la commande `Get-Content`, puis la convertir en XML comme ceci :

```

[XML]$xaml = Get-Content 'C:\scripts\generateurMDP.xaml'
$reader=New-Object System.Xml.XmlNodeReader $xaml
$Form=[Windows.Markup.XamlReader]::Load($reader)

$Form.ShowDialog() | Out-Null

```

c. Gestion des événements

Tout ceci est bien joli, nous disposons d'une belle interface graphique mais nous n'avons défini aucune action. Pour ce faire, il va falloir gérer les événements comme avec les **Windows Forms**. Nous allons donc ajouter quelques lignes de scripts supplémentaires.

Premièrement nous devons faire des recherches afin de nous connecter aux objets du formulaire. Faites bien attention car avec **WPF** les noms des objets sont sensibles à la casse, contrairement aux commandes et variables PowerShell.

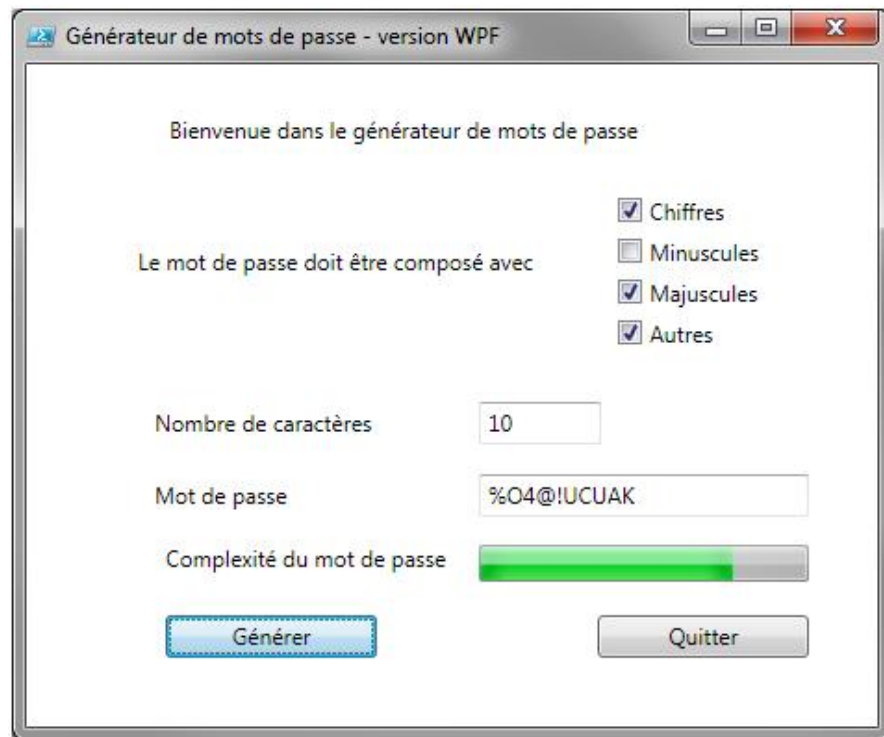
Deuxièmement, une fois connectés par exemple à un bouton, il faut ajouter à ce dernier un gestionnaire d'événement tel que « add_click ». Celui-ci comme son nom le laisse supposer, réagit en cas de clic de la souris. Dans notre exemple, en cas de clic sur le bouton **Générer**, nous lançons la génération du mot de passe. Et en cas de clic sur le bouton **Quitter**, alors le formulaire se fermera.

Exemple :

```
$btnQuitter = $form.FindName('button2')
$btnQuitter.add_click({ $Form.close() })
```

Ces quelques lignes de script sont à insérer juste avant d'afficher le formulaire, autrement dit juste avant d'appeler la méthode ShowDialog().

Et voilà notre générateur de mot de passe en version **WPF** ; vous constaterez qu'il n'y a pratiquement aucune différence visible en termes de graphique par rapport à la version réalisée avec les **Windows Forms** :



Générateur de mots de passe en version WPF

Voici le script complet du générateur de mot de passe remanié :

```
# WPF_PWDGenerator.ps1
[xml]$XAML = @"
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Générateur de mots de passe - version WPF" Height="395"
  Width="475">
  <Grid>
    <Label Height="28" Margin="72,22,113,0" Name="label1"
  VerticalAlignment="Top">Bienvenue dans le générateur de mots de
  passe</Label>
    <Label Height="28" Margin="55,93,168,0" Name="label2"
  VerticalAlignment="Top">Le mot de passe doit être composé avec</Label>
    <CheckBox Height="16" HorizontalAlignment="Right"
  Margin="0,71,20,0" Name="checkBox1" VerticalAlignment="Top"
  Width="120" IsChecked="True">Chiffres</CheckBox>
    <CheckBox Height="16" HorizontalAlignment="Right"
  Margin="0,137,20,0" Name="checkBox4" VerticalAlignment="Top"
  Width="120">Autres</CheckBox>
    <CheckBox Height="16" HorizontalAlignment="Right"
  Margin="0,115,20,0" Name="checkBox3" VerticalAlignment="Top" Width="120"
  IsChecked="True">Majuscules</CheckBox>
```



```

        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,93,20,0" Name="checkBox2" VerticalAlignment="Top"
Width="120">Minuscules</CheckBox>
        <Label Height="28" HorizontalAlignment="Left" Margin="64,0,0,149"
Name="label3" VerticalAlignment="Bottom" Width="149">Nombre
de caractères</Label>
        <Label Height="28" HorizontalAlignment="Left" Margin="64,0,0,110"
Name="label4" VerticalAlignment="Bottom" Width="149">Mot de passe</Label>
        <Label Height="28" Margin="0,0,221,76" Name="label5"
VerticalAlignment="Bottom" HorizontalAlignment="Right"
Width="168">Complexité du mot de passe</Label>
        <TextBox Height="23" HorizontalAlignment="Right"
Margin="0,0,149,152" Name="textBox1" VerticalAlignment="Bottom" Width="66" />
        <TextBox Height="23" HorizontalAlignment="Right"
Margin="0,0,37,113" Name="textBox2" VerticalAlignment="Bottom" Width="178" />
        <ProgressBar Height="20" HorizontalAlignment="Right"
Margin="0,0,37,78" Name="progressBar1" VerticalAlignment="Bottom"
Width="178" />
        <Button Height="23" HorizontalAlignment="Left" Margin="75,0,0,37"
Name="button1" VerticalAlignment="Bottom" Width="114">Générer</Button>
        <Button Height="23" HorizontalAlignment="Right" Margin="0,0,37,37"
Name="button2" VerticalAlignment="Bottom" Width="114">Quitter</Button>
    </Grid>
</Window>
'@

$reader=New-Object System.Xml.XmlNodeReader $xml
$Form=[Windows.Markup.XamlReader]::Load($reader)

# Attention: les noms d'objets recherchés sont sensibles à la casse
$textBox_Nb_caracteres = $form.FindName('textBox1')
$textBox_resultat = $form.FindName('textBox2')
$checkBox_chiffres = $form.FindName('checkBox1')
$checkBox_minuscules = $form.FindName('checkBox2')
$checkBox_majuscules = $form.FindName('checkBox3')
$checkBox_autres = $form.FindName('checkBox4')
$btnGenerer = $form.FindName('button1')
$btnQuitter = $form.FindName('button2')
$progressBar = $form.FindName('progressBar1')

$btnGenerer.add_click({
    [int]$len = $textBox_Nb_caracteres.get_text()
    $textBox_resultat.Text = ''
    $complex = 0
    $progressBar.Value = 0
    [string]$chars = ''

    if ($checkBox_chiffres.IsChecked)
        {$chars += '0123456789';$complex += 1}
    if ($checkBox_majuscules.IsChecked)
        {$chars += 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';$complex += 1}
    if ($checkBox_minuscules.IsChecked)
        {$chars += 'abcdefghijklmnopqrstuvwxyz';$complex += 1}
    if ($checkBox_autres.IsChecked)
        {$chars += '_!@#%$';$complex += 1}

    if($chars -ne ''){
        $bytes = New-Object System.Byte[] $len
        $rnd = New-Object System.Security.Cryptography.RNGCryptographyServiceProvider
        $rnd.GetBytes($bytes)
        $result = ''
        for( $i=0; $i -lt $len; $i++ )
        {
            $result += $chars[ $bytes[$i] % $chars.Length ]
        }
        $complex *= $(2.57*$len)
        if($complex -gt 100){ $complex = 100 }
        $progressBar.Value = $complex
        $textBox_resultat.Text = $result
    }
})

```

```

}) # fin du bloc du bouton "Générer"

$btnQuitter.add_click({ $Form.close() })

$Form.ShowDialog() | Out-Null

```

Une chose frappante en comparaison avec la version **Windows Forms** de ce même script est la concision. En effet nous sommes passés de plus de 200 lignes de code à moins de 80 !!! Soit un nombre de lignes divisé par deux et demi !

Non seulement cela rend le script plus lisible mais cela permet surtout de dissocier la partie conception de l'interface, de la partie métier. Ainsi, si l'envie vous prenait un jour de relooker complètement l'interface, vous n'auriez plus qu'à copier/coller le code XAML dans votre éditeur XAML préféré, le modifier, et le coller à nouveau dans le script. Tant que vous ne modifierez pas le nom des objets, tout fonctionnera à merveille. N'est-ce pas fantastique !?

d. Création de formes graphiques simples

WPF possède un grand nombre de primitives de base afin de réaliser des formes graphiques simples telles que : des ellipses, des rectangles, des lignes, des polygones, etc.

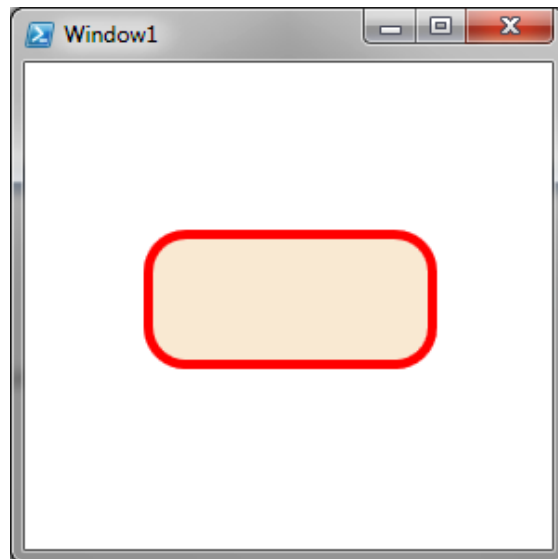
Essayons d'afficher un rectangle avec des coins arrondis :

```

# WPF_rectangle.ps1
[xml]$XAML = @"
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1" Height="300" Width="300" >
  <Grid>
    <Rectangle Margin="64,90,62,97" Name="rectangle1" Stroke="Red"
      Fill="AntiqueWhite" StrokeThickness="5" RadiusX="20" RadiusY="20" />
  </Grid>
</Window>
"@

$reader=New-Object System.Xml.XmlNodeReader $xaml
$Form=[Windows.Markup.XamlReader]::Load($reader)
$Form.ShowDialog() | Out-Null

```



*Création d'un rectangle arrondi avec **WPF***

Toujours dans l'idée de séparer le script de la partie présentation, nous avons créé dans le script une *Here-String* contenant le code XAML correspondant au rectangle que nous avons généré avec **Visual Studio 2008 Express**.

Même si **WPF** est une technologie riche, il n'existe malheureusement pas, nativement, de formes évoluées de types barreaux ou camemberts. Mais Microsoft a pensé à tout car à l'instar des **MS Charts Controls** pour **Windows Forms**, Microsoft fournit (gratuitement) le **WPF Toolkit** (téléchargeable à l'adresse : <http://wpf.codeplex.com>). Ce dernier embarquant le nécessaire pour réaliser de splendides graphiques appelés « *charts* ».

e. Réalisation de graphiques évolués avec le WPF Toolkit

La réalisation de graphiques avec **WPF** est relativement récente car les fonctionnalités de « *charting* » ont été incluses seulement dans la dernière version du **WPF Toolkit**, soit depuis Juin 2009. Avant cela, il fallait soit utiliser des bibliothèques tierces payantes, soit intégrer une **Windows Form** contenant des **MS Charts Controls** à l'intérieur d'un formulaire **WPF**.

Une fois le **WPF Toolkit** installé nous allons pouvoir tester l'exemple suivant. Ce dernier affiche sous forme de camembert les cinq processus les plus consommateurs de mémoire :

```
# WPF_Camembert.ps1
# Camembert en WPF avec le WPF Toolkit
$dataVisualization = 'C:\Program Files\WPF Toolkit' +
    '\v3.5.40619.1\System.Windows.Controls.DataVisualization.Toolkit.dll'

$wpfToolkit = 'C:\Program Files\WPF Toolkit' +
    '\v3.5.40619.1\WPFToolkit.dll'

Add-Type -Path $dataVisualization
Add-Type -Path $wpfToolkit

function ConvertTo-Hashtable
{
    param([string]$key, $value)
    Begin {
        $hash = @{}
    }
    Process {
        $thisKey = $_.Key
        $hash.$thisKey = $_.Value
    }
    End {
        Write-Output $hash
    }
} #ConvertTo-Hashtable

# Utilisation d'une Here-String contenant le code XAML
$xmlString = @'
<Window
xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'
xmlns:toolkit='http://schemas.microsoft.com/wpf/2008/toolkit'
xmlns:charting='clr-
namespace:System.Windows.Controls.DataVisualization.Charting;assembly=
System.Windows.Controls.DataVisualization.Toolkit'
Title="WPF Toolkit"
Background="LightBlue"
Height="400" Width="500" >
<charting:Chart x:Name="MyChart">
<charting:Chart.Series>
<charting:PieSeries ItemsSource="{Binding}"
DependentValuePath="Value"
IndependentValuePath="Key" />
</charting:Chart.Series>
</charting:Chart>
</Window>
'@

$xmlString = $xmlString
$reader=(New-Object System.Xml.XmlNodeReader $xml)
$control=[Windows.Markup.XamlReader]::Load($reader)

# Création d'une table de hachage contenant les données
$myHashTable = Get-Process | Sort-Object -Property PM |
Select-Object Name, PM -Last 5
$myHashTable = $myHashTable | ConvertTo-Hashtable Name PM

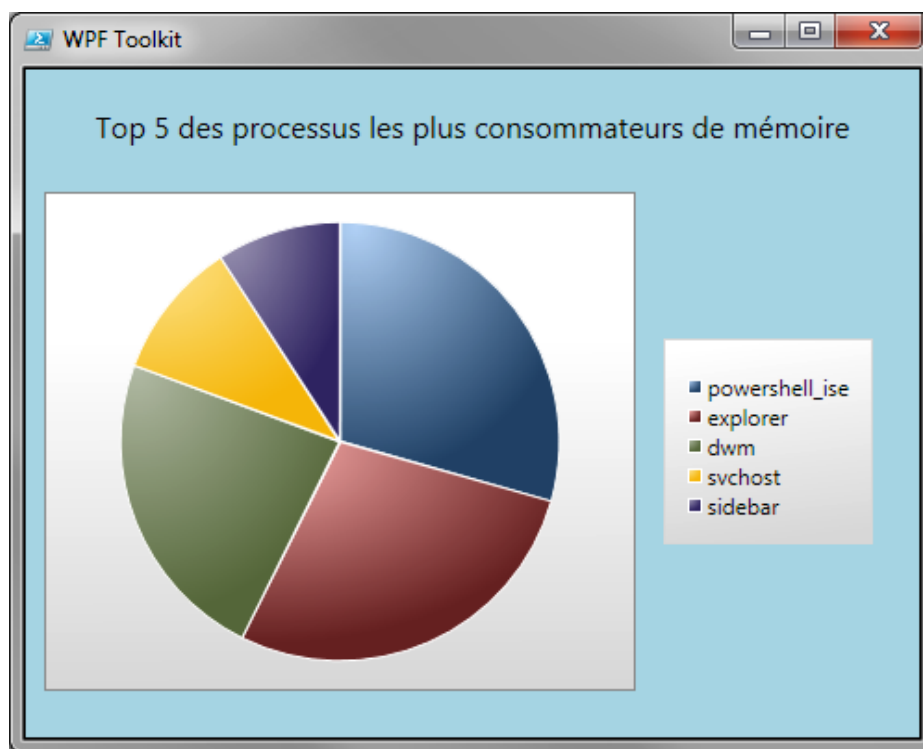
# Recherche du contrôle contenant la forme graphique
```

```

$chart = $control.FindName('MyChart')
$chart.DataContext = $myHashTable
$chart.Title = 'Top 5 des processus les plus consommateurs de mémoire'

# Affichage du résultat
$null = $control.ShowDialog()

```



Graphique au format camembert avec le WPF Toolkit

Le script récupère les cinq processus les plus gourmands en mémoire, les stocke dans un tableau associatif (appelé aussi « table de hachage ») puis les passe à la propriété `DataContent` du graphique. Le passage de données en PowerShell est possible car dans la partie XAML de l'interface, nous avons indiqué le mot clé « `{Binding}` ».

f. La bibliothèque PowerBoots

Comme l'on peut s'en rendre compte la réalisation de graphiques nécessite un certain nombre lignes de code XAML et de PowerShell. Pour ne pas faillir à notre réputation de « fainéants » nous les informaticiens, nous allons faire appel à une bibliothèque, nommée « **PowerBoots** », qui va doper notre productivité tout en limitant, bien sur, le nombre de lignes de code à écrire...

À l'heure où nous écrivons ces lignes, **PowerBoots** est en version Bêta.

PowerBoots pourrait bien s'imposer dans les mois et années à venir comme étant LA bibliothèque graphique de choix pour PowerShell.

Joel 'Jaykul' Bennet (MVP PowerShell) est la personne à l'origine de ce projet, projet que l'on peut trouver sur le site de partage communautaire Codeplex (<http://powerboots.codeplex.com>). L'idée de **PowerBoots** est de faciliter la création d'interfaces graphiques en PowerShell, en offrant une syntaxe simple. **PowerBoots** s'appuie sur **WPF** et supporte la gestion des événements, des threads, et beaucoup d'autres choses. « Boots » est un jeu de mot inventé par Jaykul pour faire un clin d'œil à la bibliothèque graphique du langage **Ruby** nommée « shoes ». Chose étonnante, la plupart des fonctionnalités de **PowerBoots** sont aussi utilisables avec PowerShell v1.

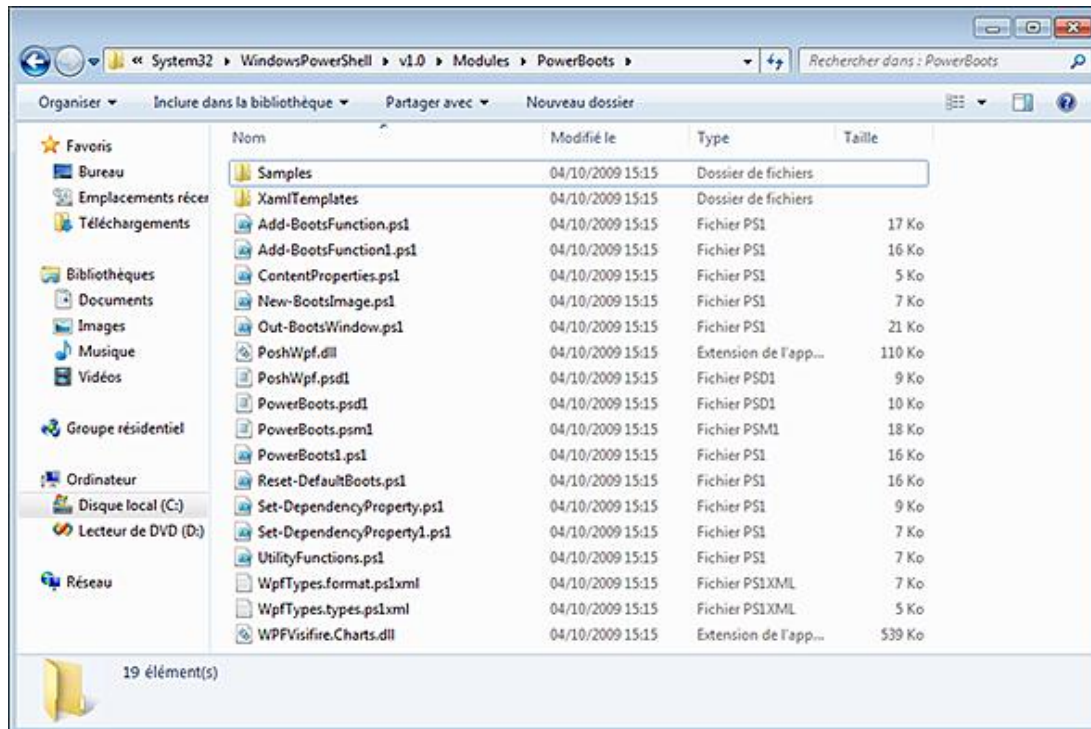
Installation

PowerBoots s'installe en tant que module.

Une fois l'archive `PowerBoots.GPL.zip` téléchargée, il est important de la débloquent (clic droit/propriétés puis Débloquer). Si vous n'êtes pas familier avec les mécanismes de sécurité alors reportez vous au chapitre La sécurité.

Ensuite décompressez l'archive dans l'un des répertoires de stockage des modules (répertoire utilisateur `C:\Users\Arnaud\Documents\WindowsPowerShell\Modules\PowerBoots` ou répertoire machine `C:\Windows\system32\WindowsPowerShell\v1.0\Modules\PowerBoots`).

Vous devriez obtenir une arborescence qui ressemble à celle-ci :



Répertoire d'installation de PowerBoots

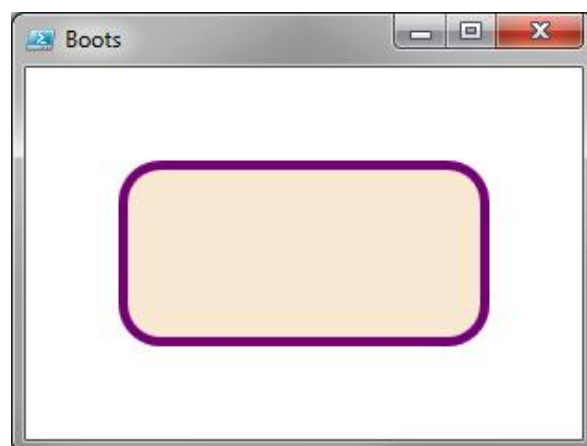
Ensuite, ouvrez une console PowerShell en mode Administrateur, positionnez vous dans le répertoire d'installation de **PowerBoots** et exécutez le script PowerBoots.ps1. Celui-ci créera le dossier Functions et générera à l'intérieur un grand nombre de scripts utilisés par la bibliothèque **PowerBoots**.

Utilisation de Boots

La dernière étape nécessaire pour pouvoir utiliser PowerBoots est le chargement du module. Pour effectuer cette étape, il n'est plus nécessaire d'être dans une console PowerShell en mode Administrateur :

```
PS > Import-Module Powerboots
```

À présent, utilisons **PowerBoots** pour créer un rectangle :



Création d'un rectangle arrondi avec PowerBoots

```
Boots {  
    Rectangle -Margin '50,50,50,50' -Width 200 -Height 100 -Stroke Purple `   
    -StrokeThickness 5 -Fill AntiqueWhite -RadiusX 20 -RadiusY 20  
}
```

2-3 lignes de code pour faire un rectangle en **WPF**, qui dit mieux ?

Pour afficher une image dans une fenêtre, c'est très simple aussi :

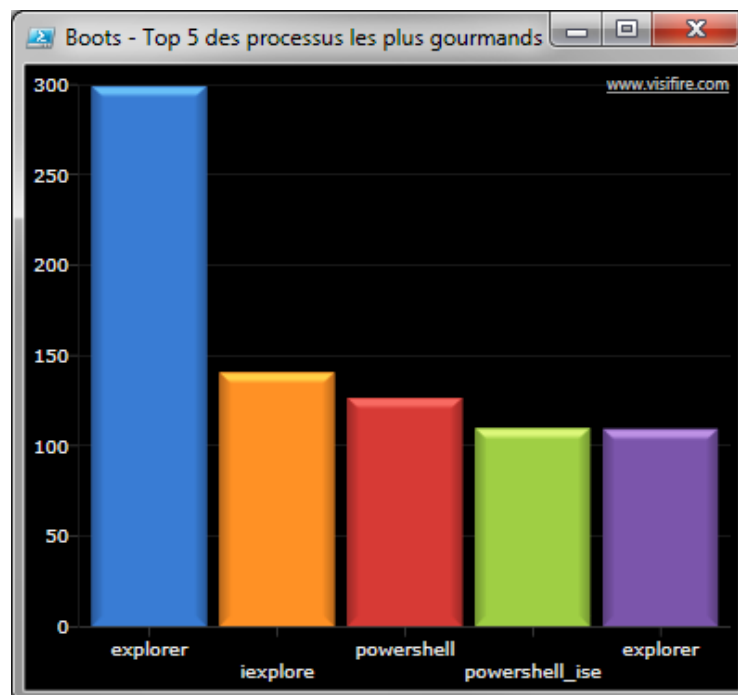
```
Boots {  
    Image -Source D:\Photos\Motos\gsx650f1.jpg -MaxWidth 400  
} -Title 'Quand je serais grand...' -Async
```



Affichage d'une image avec PowerBoots

Enfin, notre « classique » barregraphe des processus les plus consommateurs de mémoire en 6 lignes :

```
Boots {  
    Chart -MinWidth 200 -MinHeight 150 -Theme Theme3 {  
        DataSeries {  
            get-process | sort PM -Desc | Select -First 5 | %{  
                DataPoint -AxisXLabel ($_.ProcessName) -YValue ($_.PM/1MB)  
            }  
        }  
    }  
} -Title 'Boots - Top 5 des processus les plus gourmands'
```



Graphique en mode colonnes avec PowerBoots

4. Récapitulatif

Comme vous avez pu vous en rendre compte, il existe une multitude de possibilités pour créer des graphiques simples comme complexes avec PowerShell. Et il en existe encore bien d'autres, comme par exemple, les **PowerGadgets** (cf. chapitre Ressources complémentaires), mais ce sujet est tellement vaste qu'il pourrait faire l'objet d'un ouvrage entier.

Pour résumer cette partie sur la création d'interfaces graphiques, il n'est pas possible de dire laquelle des technologies **Windows Forms** ou **WPF** choisir. En effet, vous pouvez faire globalement la même chose avec ces deux technologies, et seule la mise en œuvre est différente. Votre choix se fera, en toute vraisemblance, en fonction du temps et des compétences dont vous disposez déjà.

WPF est une technologie vraiment puissante et sa mise en œuvre est relativement rapide. L'aspect le plus intéressant est le fait de pouvoir dissocier l'interface graphique de la partie métier. Vous ne devez jamais perdre de vue que **WPF** est la technologie qui succède à **Windows Forms**, par conséquent si vous débutez nous vous conseillons vivement d'opter directement pour **WPF** sans passer par la case **Windows Forms**. Ceci étant, soyons honnête, comme toute chose nouvelle vous ne trouverez certainement que peu d'aide sur Internet pour surmonter vos éventuels problèmes. Ce qui est certain en tout cas, c'est que **WPF** représente l'avenir à court ou moyen terme...

Une dernière chose : n'oubliez pas que pour utiliser **WPF**, PowerShell v2 est nécessaire. La version 2 de PowerShell étant relativement récente et il est probable qu'elle soit encore peu déployée. Par conséquent, si vous devez fournir un script avec interface graphique devant s'exécuter sur un grand nombre d'ordinateurs, il sera préférable d'utiliser le couple PowerShell v1/**Windows Forms**.

Introduction à la technologie COM

Nous allons dans ce chapitre nous intéresser au contrôle d'applications, c'est-à-dire comment interagir à travers un script PowerShell avec certaines applications grâce à leurs API (*Application Programming Interface*). Pour ce faire, nous allons utiliser la technologie COM (*Component Object Model*) qui, grâce à ces objets permet d'interagir avec de nombreuses applications en ligne de commande comme si vous le faisiez graphiquement.

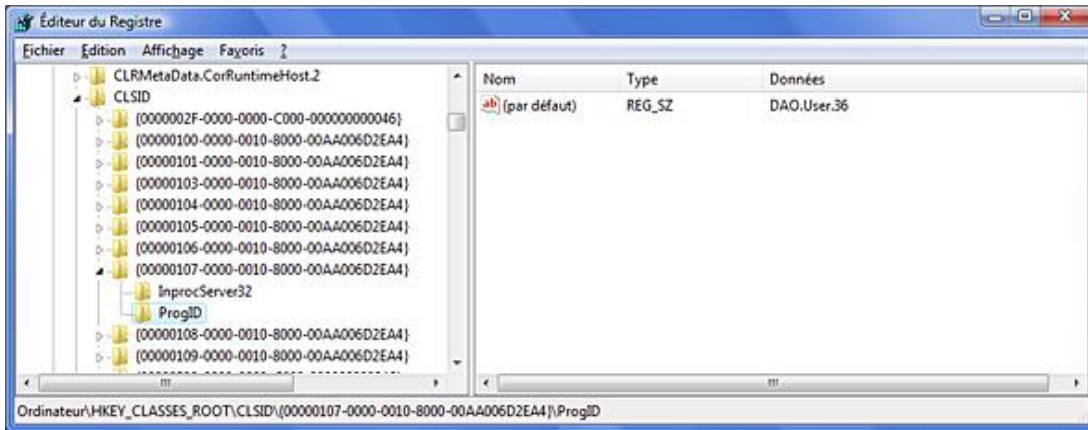
COM, les Origines

Introduit sous *Windows 2.x* en fin des années 80, la technologie Dynamic Data Exchange (DDE) fut l'une des premières à permettre les communications entre les applications Windows. À partir de ce moment là, les techniques de communication inter application ont évolué vers d'autres technologies comme Object Linking Embedding (OLE) qui intégra pour sa deuxième mouture (en 1993) le terme COM. La technologie COM a pour principal attrait, la souplesse et la modularité des communications entre les composants logiciels. Et c'est pour apporter cette notion de modularité que COM introduit la notion d'interface objet ; celle-ci permettant d'appeler les méthodes depuis un programme, et ce quel que soit le langage utilisé. L'interface est un élément important des objets COM, c'est elle qui rend disponible l'accès à l'ensemble des données et des fonctions de l'objet.

COM est une technologie qui à l'heure actuelle est encore très utilisée, notamment pour le contrôle d'applications comme Internet Explorer, Microsoft Office, et tout autre éditeur ayant développé des objets COM pour leurs applications. Sans oublier les nombreuses interfaces que fournit Microsoft pour des services d'application Windows comme Active Directory Service Interface (ADSI) et Windows Management Instrumentation (WMI).

Manipuler les objets COM

L'utilisation des objets COM n'est vraiment pas quelque chose de compliqué en soi. Mais pour pouvoir les utiliser, encore faut-il qu'ils soient disponibles sur votre poste de travail. Pour le savoir, c'est dans la base de registres qu'il faut fouiller, et plus précisément sous le chemin HKey_Classes_Root\CLSID. Comme on peut le voir sur la figure ci-dessous, les CLSID sont ces identifiants codés en hexadécimal permettant d'identifier de façon unique chaque objet COM. Peu compréhensible humainement, les CLSID se voient dotés de ce que l'on appelle un ProgID (Identifiant Programmatique). Ce ProgID qui n'est autre qu'une représentation sous forme de chaîne de caractères d'un objet COM est structuré de manière à rendre le plus explicite possible le rôle de l'objet COM. Ainsi, tous les ProgID sont composés de la façon suivante : <Programme>.<composant>.<numéro de version>. Remarquez qu'il est plus facile de retenir Word.Application.1 que {00000107-0000-0060-8000-00AA006D2EA4}



Affichage du ProgID depuis l'éditeur de la base de registre

1. Rechercher un objet

Bien entendu, plutôt que de parcourir graphiquement toutes les références contenues dans la base de registres pour y trouver un ProgID, il serait bien plus intéressant d'automatiser une recherche via une fonction PowerShell. La fonction Get-ProgID que nous vous proposons, répond à ce besoin. Elle parcourt la base de registres, compare chaque entrée et affiche la liste des ProgID dont le nom correspond à celui recherché.

```
Function Get-ProgID
{
param([string]$ProgID = '.')
Get-ChildItem REGISTRY::HKey_Classes_Root\clsid*\* |
Where-Object {$_.GetValue('') -match $ProgID} | ForEach-Object{$_.GetValue('')}
}
```

Une fois cette fonction réalisée, vous pourrez rapidement retrouver le nom d'un ProgID disponible sur votre système d'exploitation Windows. L'exemple ci-dessous vous montre quels résultats vous sont retournés après une recherche sur un Identifiant Programmatique dont une partie du nom est « Explorer » :

```
PS > Get-ProgID Explorer

InternetExplorer.Application.1
Shell.Explorer.2
Groove.WorkspaceExplorerApplication
Shell.Explorer.1
```

L'obtention des ProgID est également possible en faisant appel à la classe WMI Win32_ClassicCOMClassSetting. Exemple de la fonction Get-ProgID réalisée avec l'utilisation de la classe WMI :

```
Function Get-ProgID
{
param([string]$ProgID = '.')

Get-WmiObject Win32_ClassicCOMClassSetting |
Where-Object {$_.ProgId -match $ProgID} |
```

```
Select-Object ProgID,Description  
}
```

Notez cette fois-ci que nous affichons également la propriété description de chaque objet contenu dans la classe. Exemple d'utilisation avec le mot clé « Explorer » :

```
PS > Get-ProgID Explorer  
  
ProgID                                Description  
-----  
InternetExplorer.Application.1       Internet Explorer(Ver 1.0)  
Shell.Explorer.2                     Microsoft Web Browser  
Groove.WorkspaceExplorerApplication  Groove WorkspaceExplorerApplication  
Shell.Explorer.1                     Microsoft Web Browser Version 1
```

2. Créer un objet

Une fois que nous avons trouvé l'objet COM correspondant à nos besoins, l'instanciation se réalise avec la commande **New-Object** qui, nous vous rappelons, permet également de créer des objets .NET. Cette commande possède effectivement une double utilisation. Seulement, cette fois, pour créer un objet COM, il est nécessaire d'utiliser le paramètre **-ComObject** et de donner le ProgID qui va permettre d'identifier l'objet COM souhaité. Dans le cadre d'une utilisation avec les objets COM, la commande **New-Object** possède deux paramètres (hors paramètres communs), dont voici le détail :

Paramètre	Description
-ComObject <String> ou -Com <String>	Spécifie à la commande le ProgID de l'objet COM.
-Strict	Retourne un message d'erreur si l'objet COM passé en paramètre est en réalité un objet .NET « wrappé ».

Les objets COM n'échappant pas à la règle (sauf cas particulier), leurs propriétés et méthodes sont disponibles avec un simple **Get-Member**. Exemple avec la création d'un objet COM `wscript.shell` (représentation de l'environnement d'exécution WSH, voir fin de ce chapitre).

```
PS > $WShell = New-Object -ComObject WScript.Shell  
PS > $WShell | Get-Member  
  
    TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd35090}  
  
Name                MemberType Definition  
----  
AppActivate          Method      bool AppActivate (Variant, Variant)  
CreateShortcut        Method      IDispatch Create Shortcut (string)  
Exec                  Method      IWshExec Exec (string)  
ExpandEnvironmentStrings Method      string Expand EnvironmentStrings ...  
LogEvent              Method      bool LogEvent (Variant, string, st...  
Popup                 Method      int Popup (string, Variant, Varian...  
RegDelete             Method      void RegDelete (string)  
RegRead               Method      Variant RegRead (string)  
RegWrite              Method      void RegWrite (string, Variant, Var...  
Run                   Method      int Run (string, Variant, Variant)  
SendKeys              Method      void SendKeys (string, Variant)  
Environment           Parameter...IWshEnvironment Environm...  
CurrentDirectory      Property    string CurrentDirectory () {get} {set}  
SpecialFolders         Property    IWshCollection SpecialFolders () {get}
```

Agir sur des applications avec COM

1. Microsoft Office 2007

Dans cette section, nous allons nous intéresser à la manipulation des objets COM de façon à interagir avec les logiciels contenus dans le pack Office 2007 (Word, PowerPoint, Excel, etc.). Peut-être ne le savez-vous pas, mais lors de l'installation du pack Office sur un système d'exploitation Windows, ce dernier se voit créditer d'un bon nombre d'objets COM Microsoft Office, permettant aux utilisateurs de les utiliser dans leurs scripts ou autres programmes.

La simple utilisation de la fonction `Get-ProgID` (cf. partie sur rechercher un objet) associée aux noms Word, PowerPoint, Excel etc., vous donnera une liste impressionnante d'objets COM utilisables à votre convenance. Après cela, il ne vous restera plus qu'à vous rendre sur le site Internet MSDN pour obtenir une description détaillée des nombreuses propriétés et des méthodes de chaque objet.

a. Microsoft PowerPoint 2007

Pour nous rendre compte à quel point la manipulation de l'application PowerPoint 2007 est rendue facile grâce aux objets COM, voici comment en quelques lignes, nous pouvons automatiser l'ouverture d'un fichier. Premièrement, créons une instance de l'objet `PowerPoint.Application`.

```
PS > $ObjetPowerPoint = New-object -ComObject PowerPoint.Application
```

Notons au passage qu'après instanciation de l'objet COM `PowerPoint.Application`, le processus `POWERPNT`, correspondant au processus d'exécution Microsoft PowerPoint, est logiquement actif.

```
PS > Get-Process
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
25	2	604	1004	25	0,05	2456	acrotray
103	5	1100	1568	32		1104	alg
44	1	364	1168	16		960	ati2evxx
234	8	3324	3256	51	0,33	2944	ccApp
216	6	2472	1148	40		1820	ccEvtMgr
733	15	15000	12952	73		568	CcmExec
217	5	5820	136	66		1876	cisvc
33	2	516	1404	19		1924	DefWatch
447	12	14760	11636	81	41,41	4012	explorer
0	0	0	16	0		0	Idle
302	13	25100	712	154	11,81	3824	POWERPNT

Regardons à présent, encore une fois grâce à la commande `Get-Member`, quelles propriétés ou méthodes nous pouvons utiliser pour modifier le comportement de notre objet.

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

TypeName: Microsoft.Office.Interop.PowerPoint.ApplicationClass

Name	MemberType	Definition
-----	-----	-----
Activate	Method	void Activate ()
GetOptionFlag	Method	bool GetOptionFlag (int, bool)
Help	Method	void Help (string, int)
LaunchSpelling	Method	void LaunchSpelling (Document...
PPFileDialog	Method	IUnknown PPFileDialog (PpFile...
Quit	Method	void Quit ()
...		

Après un rapide coup d'œil sur les membres disponibles, commençons par rendre visible l'application grâce à la propriété `Visible`.

```
PS > $ObjetPowerPoint.Visible = 'MsoTrue'
```

Nous appliquons à la propriété Visible une valeur de type MsoTriState spécifique à Microsoft Office. Mais selon la version que vous utilisez, vous serez peut-être contraint d'utiliser la forme suivante :

```
PS > $ObjetPowerPoint.Visible = $true
```

Une fois l'application lancée et visible, utilisons la méthode Add afin de créer une présentation vide. Cela revient à faire dans PowerPoint « Fichier/Nouvelle présentation ». Une fois cette opération effectuée, nous pourrions ajouter des diapositives manuellement (car la fenêtre apparaît à l'écran) ou en ligne de commandes.

```
PS > $ObjetPowerPoint.Presentations.Add()
```

Bien entendu, il est également possible d'ouvrir une présentation déjà existante avec la méthode Open :

```
PS > $ObjetPowerPoint.Presentations.Open(<Chemin du fichier>)
```

En résumé, l'utilisation des objets COM Microsoft Office rend accessible par script toutes les opérations réalisables graphiquement, ouverture/fermeture d'un fichier, enregistrement d'un fichier, ajout de diaporama, etc.

Pour vous en donner un aperçu plus complet, voici un exemple de script PowerShell. Celui-ci crée un diaporama PowerPoint sur lequel chaque diapositive est une photo au format JPG située dans un répertoire donné. Le chemin étant passé en paramètre.

```
#diapo.ps1
#création d'une présentation PowerPoint à partir d'images jpg

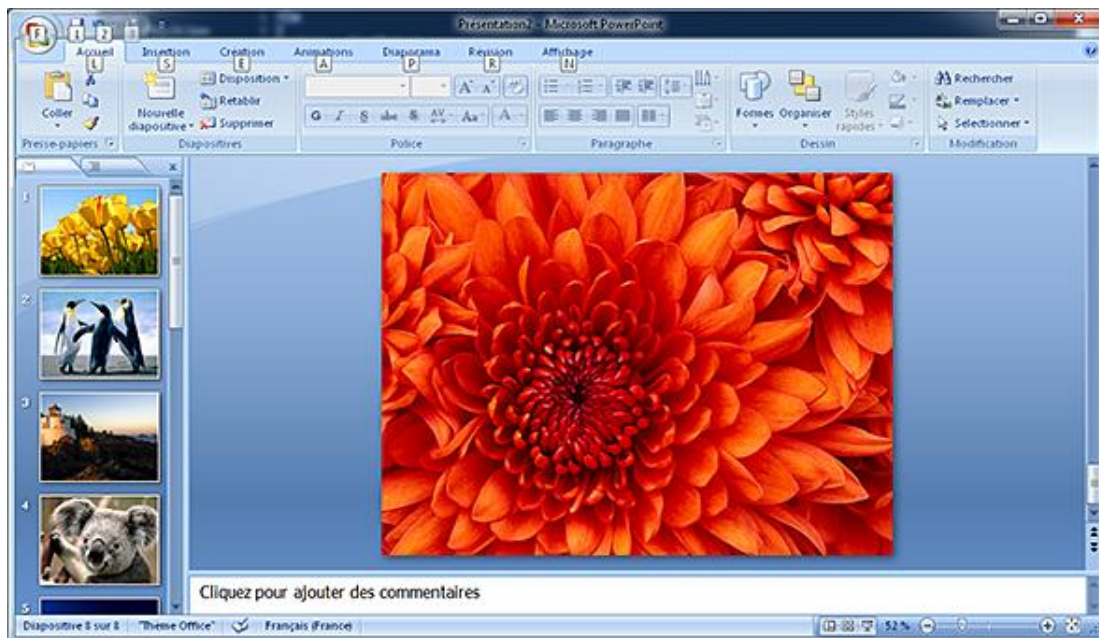
Param([string]$path)
$liste = Get-ChildItem $path | Where {$_.extension -eq '.jpg'} |
Foreach{$_.Fullname}

if($liste -ne $null)
{
    $Transition = $true
    $time_diapo = 2
    $objPPT = New-object -ComObject PowerPoint.Application
    $objPPT.Visible = 'Msottrue'
    $projet = $objPPT.Presentations.Add()
    foreach($image in $liste)
    {
        $slide = $projet.Slides.Add(1, 1)
        $slide.Shapes.AddPicture($image,1,0,0,0,720,540)
        $slide.SlideShowTransition.AdvanceOnTime = $Transition
        $slide.SlideShowTransition.AdvanceTime = $time_diapo
    }
    $projet.SlideShowSettings.Run()
}
Else
{
    Write-Host 'Pas de photo au format JPEG dans ce répertoire !'
}
```

Exemple d'utilisation avec l'échantillon d'image de Windows 7.

```
PS> ./diapo.ps1 'C:\Users\Public\Pictures\Sample Pictures'
```

Le résultat est le suivant.



b. Microsoft Word 2007

Toujours dans le domaine des objets COM du pack Office, allons voir ceux que Word 2007 nous propose.

Comme vous le savez, le pack Office 2007 est à l'origine des nouvelles extensions ajoutant un « x » aux extensions que nous connaissons déjà (*.docx, *.pptx, etc.). Plus qu'un simple changement, ces nouvelles extensions sont le signe d'un nouveau type de document Office au format XML. Par conséquent, ces fichiers sont rendus inutilisables avec les versions précédentes du Pack Office. Cependant, pour maintenir une compatibilité ascendante des documents Office, le pack 2007 permet d'enregistrer des fichiers dans un mode dit de « compatibilité ».

Ainsi en choisissant le bon format d'enregistrement vous pourrez continuer de partager vos documents avec d'autres personnes n'utilisant pas Office 2007.

Mais là où cela devient ennuyeux, c'est en cas de conversion massive de documents. La seule solution reste d'ouvrir les documents un par un, les enregistrer dans un mode de compatibilité 97-2003, et les refermer. Autrement dit voilà une tâche idéalement candidate à convertir en script PowerShell.

Prenons l'exemple des documents Word. Pour réaliser une ouverture puis l'enregistrement d'un document, nous avons premièrement besoin d'instancier l'objet COM Word.Application puis d'appliquer la méthode Open sur le membre Document.

```
PS > $objWord = New-object -ComObject Word.Application
PS > $objWord.Visible = 'MsoTrue'
PS > [void]$objWord.Documents.Open(<Nom de fichier.docx>)
```

Vous aurez remarqué l'usage de « [void] » devant l'appel à la méthode Open. Cela permet de ne pas afficher sur la console le résultat de la commande. En effet, lorsque l'on utilise la méthode Open, celle-ci génère des dizaines de lignes correspondant à toutes les propriétés du document ouvert. Or, comme nous n'avons que faire de ces informations (pour cet exemple) et que celles-ci nous « polluent » la console, [void] nous permet de ne pas les afficher. Au lieu d'utiliser [void] nous aurions pu affecter cette commande à une variable, ce qui serait revenu au même.

Ensuite, arrive le moment délicat de la sauvegarde via la méthode SaveAs et de ses seize paramètres.

Paramètre	Description
FileName	Permet de définir le nom du document. Si le document a déjà été enregistré, la valeur par défaut est son nom actuel.
FileFormat	Permet de définir le format du document enregistré. Dans le cas d'un document Word, il peut s'agir de n'importe quelle valeur wdSaveFormat (cf. tableau sur les formats d'enregistrement Word).

LockComments	Permet de définir un verrouillage des commentaires si la valeur True est associée. La valeur par défaut est \$False.
Password	Permet de définir un mot de passe pour l'ouverture du document.
AddToRecentFiles	Permet de définir l'ajout du document à la liste des derniers fichiers utilisés si la valeur True lui est associée. La valeur par défaut est True.
WritePassword	Permet de définir un mot de passe pour la modification du document.
ReadOnlyRecommended	Permet de définir l'ouverture du document en lecture seule si la valeur True est associée. La valeur par défaut est False.
EmbedTrueTypeFonts	Permet de définir l'enregistrement des polices TrueType si la valeur True est associée.
SaveNativePictureFormat	Permet d'enregistrer en version Windows des graphiques importés d'une autre plate-forme si la variable True est associée.
SaveFormsData	Définit l'enregistrement des données entrées par un utilisateur dans un formulaire.
SaveAsAOCELetter	Permet d'enregistrer un document au format AOCE (<i>Apple Open Collaboration Environment</i>) si la valeur True est associée.
Encoding	Permet de définir l'encodage du document (Arabic, Cyrillic, etc.).
InsertLineBreaks	Permet d'insérer des retours chariots à chaque fin de ligne si la valeur True est associée. Cette propriété peut être intéressante dans le cadre d'un enregistrement au format texte.
AllowSubstitutions	Permet dans le cas d'un enregistrement au format texte, de remplacer les symboles du document par une apparence similaire.
LineEnding	Permet de définir la manière dont Word va signaler les sauts de lignes pour les documents enregistrés au format texte.
AddBiDiMarks	Permet l'ajout de caractères de contrôle au fichier pour conserver la direction bidirectionnelle du texte dans le document d'origine.

En utilisant habilement la méthode SaveAs, nous pourrions enregistrer le document dans un format « document Word 97-2003 » ; et cela grâce au paramètre FileFormat dont voici la liste non exhaustive de ses valeurs possibles.

Format	Extension associée	Valeur du paramètre
wdFormatDocument	*.doc	0
wdFormatHTML	*.htm, *.html	8
wdFormatPDF	*.pdf	17
wdFormatTemplate	*.dot	1
wdFormatText	*.txt	2
wdFormatXMLDocument	*.docx	12
wdFormatXMLTemplate	*.dotx	14



Pour pouvoir enregistrer au format PDF, l'application Office 2007 doit disposer d'un composant supplémentaire (gratuit) téléchargeable sur le site de Microsoft.

En choisissant la valeur 0, le fichier sera enregistré en mode compatibilité Office 97-2003. Cependant, faites attention à bien spécifier un nom de fichier avec son extension « .doc » sinon, le fichier gardera son extension « .docx ». Notez également, que la commande **SavesAs** nécessite que ses arguments soient de type *PSReference* (référence à un objet), attention donc à bien spécifier le type [ref] devant chaque variable.

```
PS > $objword.ActiveDocument.SaveAs([ref]<nom du fichier.doc>,[ref]0)
```

Et enfin, pour terminer le processus, appelons les méthodes Close et Quit respectivement sur le membre Documents et sur l'objet COM.

```
PS > $objWord.Documents.Close()
PS > $objWord.Quit()
```

En associant les éléments que nous venons de présenter, nous arrivons au script suivant qui permet de convertir en masse tout document natif Word 2007 d'un répertoire en un document Word 97-2003.

```
Function Convert-Doc
{
param ([String]$path = '.')
$liste = Get-ChildItem $path *.docx
$objWord = New-Object -ComObject Word.Application
    Foreach ($fichier in $liste)
    {
        [void]$objWord.Documents.Open($($fichier.FullName))
        $nom_fichier = $($fichier.FullName).replace('.docx',' .doc')
        $objword.ActiveDocument.SaveAs([ref]$nom_fichier,[ref]0)
        $objWord.Documents.Close()
    }
$objWord.Quit()
}
```



Le script ne supprime pas l'occurrence du fichier .docx, mais crée un nouveau fichier .doc portant le même nom.

Parlons sécurité

Dans cet exemple, nous allons aller un peu plus loin dans la gestion de sauvegarde des fichiers Word, puisque nous allons désormais nous intéresser à la protection d'un document.

Peut-être ne le savez-vous pas, mais avec Microsoft Office, il est possible de définir un mot de passe pour la lecture et pour la modification de vos documents Word et Excel.

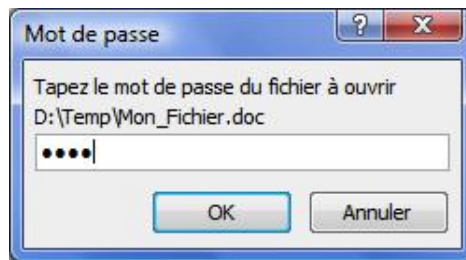
Comme nous avons pu le voir dans la partie précédente, la méthode SaveAs d'enregistrement d'un document dispose des paramètres Password et WritePassword. C'est tout simplement sur ces deux paramètres que nous allons jouer pour définir un mot de passe sur un document. Commençons par ouvrir un document Word existant :

```
PS > $objWord = New-object -ComObject Word.Application
PS > $objWord.Visible = 'MsoTrue'
PS > [void]$objWord.Documents.Open(<Nom de fichier>)
```

Puis, procédons à son enregistrement en prenant soin de bien renseigner les paramètres Password et WritePassword. Notez que pour laisser vides les nombreux paramètres optionnels de la méthode SaveAs, nous utilisons un objet de type Systeme.Missing (Non renseigné).

```
PS > $m = [system.type]::missing
PS > $objWord.ActiveDocument.SaveAs([ref]<nom du fichier>,[ref]12,
[ref]$m,[ref] <Password>,[ref]$m,[ref]<WritePassword>)
```

Ainsi, comme le montre la figure ci-après, dès la prochaine ouverture du document, Word invitera l'utilisateur à entrer un mot de passe pour la lecture et/ou pour la modification.



Demande du mot de passe pour un document Word

La fonction suivante nous permet d'automatiser cette tâche en saisissant simplement le chemin complet du répertoire contenant les fichiers à sécuriser ainsi que les mots de passe pour la lecture et pour la modification.

```
Function Secure-Doc
{
param ([String]$path = '.', [String]$password='', `
[String]$WritePassword='')
$m = [system.type]::missing
$fichiers = Get-ChildItem $path *.docx
$objWord = New-object -ComObject Word.Application
Foreach ($doc in $fichiers)
{
[void]$objWord.Documents.Open($doc.FullName)
$nom_fichiersecurise = $doc.FullName + 's'
$objWord.ActiveDocument.SaveAs($nom_fichiersecurise, `
[ref]12, [ref]$m, `
[ref]$password, [ref]$m, [ref]$writepassword)
$objWord.Documents.Close()
}
$objWord.Quit()
}
```

À l'inverse, comme le montrent les deux lignes de PowerShell suivantes, la désécurisation d'un document nécessitera quant à elle d'ouvrir le fichier avec la méthode Open, en prenant soin de renseigner les mots de passe nécessaires à l'ouverture et la modification du document. Puis d'enregistrer à nouveau le document en attribuant une valeur nulle aux propriétés Password et WritePassword.

```
# Ouverture du Document avec les mots de passes nécessaires

PS > $objWord.Documents.Open(<Nom de fichier>,$m,$m,$m, `
<password>,$m,$m, <writepassword>)

# Enregistrement du document sans mot de passe

PS > $objWord.ActiveDocument.SaveAs([ref]<Nom de fichier>, `
[ref]12,[ref]$m, [ref]'',[ref]$m,[ref]'')
```

2. Windows Live Messenger

Tout en continuant notre voyage à travers les objets COM, posons-nous quelques instants sur les objets COM de Windows Live Messenger. Dans cette partie, nous allons vous montrer comment interagir avec votre messagerie instantanée, et ce uniquement avec PowerShell.

Cependant prudence, car toutes les classes COM ne sont pas disponibles sous certaines versions de Windows. Donc, pour un souci de compréhension, sachez que tous les exemples que nous vous proposons ont été réalisés avec Windows Live Messenger 8.0 sous le système d'exploitation Windows 7.

a. Obtenir le statut de connexion

Commençons par instancier un objet COM qui va nous permettre d'interagir avec Windows Messenger.

```
PS > $msn = New-Object -ComObject Messenger.UIAutomation.1
```

Maintenant, regardons d'un peu plus près les membres qu'il contient :

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

```
TypeName: System.__ComObject#{d50c3486-0f89-48f8-b204-3604629dee10}
```

Name	MemberType	Definition
----	-----	-----
AddContact	Method	void AddContact (int, string)
AutoSignin	Method	void AutoSignin ()
CreateGroup	Method	IDispatch CreateGroup (strin...
FetchUserTile	Method	void FetchUserTile (Varian...
FindContact	Method	void FindContact (int, ...
GetContact	Method	IDispatch GetContact (str...
HideContactCard	Method	void HideContactCard (int)
InstantMessage	Method	IDispatch InstantMess...
...		
MyServiceName	Property	string MyServiceName () {get}
MySigninName	Property	string MySigninName () {get}
MyStatus	Property	MISTATUS MyStatus () {get} {...
ReceiveFileDirectory	Property	string ReceiveFileDirectory ()...
Services	Property	IDispatch Services () {get}
Window	Property	IDispatch Window () {get}

Nous voyons qu'il existe de nombreuses méthodes et propriétés. Commençons par appliquer la propriété MyStatus, qui comme son nom le laisse deviner retourne l'état de votre connexion.

```
PS > $msn.MyStatus  
2
```

Comme vous pouvez le constater, la propriété MyStatus renvoie un entier associé à un état. La liste des états définie par Windows Live Messenger est donnée ci-dessous :

Valeur	État
1	Non connecté
2	En ligne
6	Hors ligne
10	Occupé
14	De retour dans une minute
34	Absent
50	Au téléphone
66	Parti manger

Notons que la propriété MyStatus est aussi bien accessible en lecture qu'en écriture.

```
PS > $msn | Get-Member MyStatus | Format-List
```

```
TypeName   : System.__ComObject#{d50c3486-0f89-48f8-b204-3604629dee10}  
Name       : MyStatus  
MemberType : Property  
Definition : MISTATUS MyStatus () {get} {set}
```

Cela signifie que nous pouvons également changer notre statut depuis la console PowerShell.

Exemple :

```
PS > If ($MSN.MyStatus -eq 2) {$MSN.MyStatus = 10}
```

Il est également possible de retrouver des informations sur des contacts, et ainsi connaître leurs statuts. Pour faire cela, nous devons créer un objet contact qui va nous être retourné par la méthode `GetContact`, puis utiliser sa propriété `MyStatus`.

De façon à illustrer ces propos, voici un exemple, sous forme de script, qui nous permettra d'obtenir le statut d'un de nos contacts :

```
# Get-MSNStatus.ps1
# Permet d'obtenir le statut d'un contact

param ($mailAddress=$(Throw='Vous devez fournir une adresse mail !'))

$msn = New-Object -ComObject Messenger.UIAutomation.1
$contact = $msn.GetContact($mailAddress,$msn.MyServiceId)
$status = $contact.Status
$nom = $contact.Friendlyname
switch ($status)
{
    1 {Write-host "$nom est non connecté"}
    2 {Write-host "$nom est en ligne"}
    6 {Write-host "$nom est hors ligne"}
    10 {Write-host "$nom est de retour dans une minute"}
    14 {Write-host "$nom est absent"}
    34 {Write-host "$nom est occupé"}
    50 {Write-host "$nom est au téléphone"}
    66 {Write-host "$nom est parti manger"}
    default {Write-host "Inconnu"}
}
```

Exemple d'utilisation :

```
PS > ./Get-MSNStatus.ps1 edouardbracame@gaz.org
Ed est occupé
```

Nous pouvons aussi obtenir la liste des nos amis actuellement connectés, comme dans l'exemple suivant :

```
#Get-OnLineFriend
#Permet de lister les amis connectés

$msn = New-Object -ComObject Messenger.UIAutomation.1
$msn.MyContacts | Where{$_.status -ne 1} | ft FriendlyName
```

Résultat :

FriendlyName	Status
Ed	34
Joe	2
Paul	66

b. Ouverture et fermeture de session

Passons maintenant sur d'autres fonctionnalités de l'objet que sont l'ouverture et la fermeture de session. En ce qui concerne la fermeture, rien de bien sorcier, il nous suffira simplement d'appeler la méthode `Close` de l'objet COM. L'ouverture quant à elle, nécessite une étape supplémentaire que nous allons détailler.

Pour pouvoir interagir sur l'application, la première étape consiste à instancier l'objet `Messenger.UIAutomation.1` ainsi qu'à le rendre visible.

```
PS > $MSN = New-Object -ComObject Messenger.UIAutomation.1
PS > $MSN.Window.Show()
```

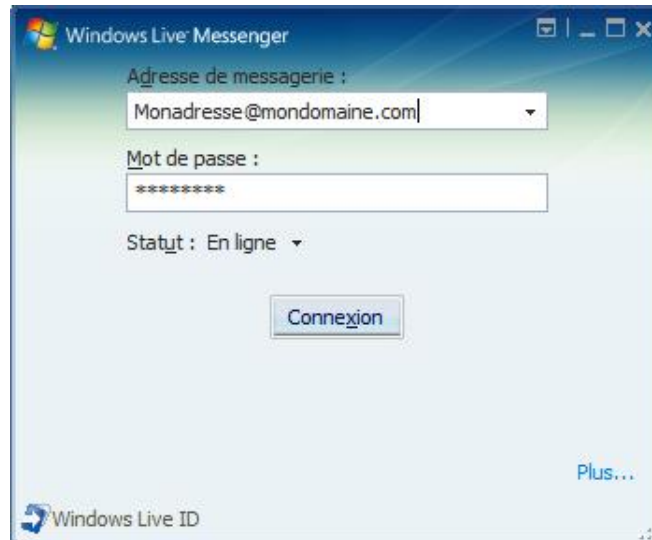
Procédons ensuite à l'ouverture de la session avec la méthode `Signin` avec comme paramètre :

- HwndParent égal à 0, pour signifier qu'il s'agit d'une fenêtre indépendante,
- l'adresse mail du compte Microsoft Passeport,
- le mot de passe associé.

```
PS > $MSN.Signin(0, 'Monadresse@mondomaine.com', 'P@SSw0rd')
```

Nous voici donc en présence de la fenêtre de connexion pré-remplie (cf. figure ci-dessous). Mais il reste un détail : lancer la connexion par un appui sur la touche [Entrée]. Pour ce faire, nous allons simplement, grâce à la classe `System.Windows.Forms.SendKeys` du Framework, envoyer une séquence de touches correspondant à l'appui de [Entrée] sur le clavier.

```
PS > [System.Reflection.Assembly]::LoadWithPartialName('System.windows.forms')
PS > [System.Windows.Forms.SendKeys]::SendWait('{enter}')
```



Fenêtre de connexion Windows Live Messenger

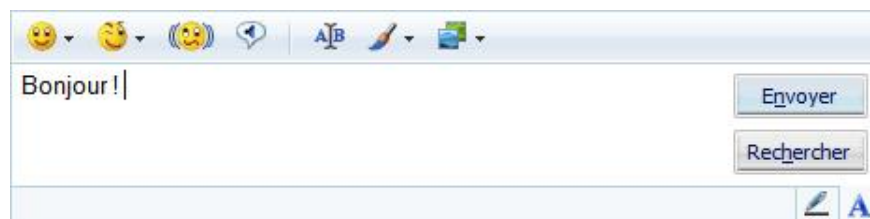
c. Envoyer un message instantané

Afin d'envoyer un message, nous utilisons la méthode `InstantMessage` à laquelle nous donnerons l'adresse du destinataire. Puis, comme pour l'ouverture de session, nous allons, pour l'envoi d'un message instantané, utiliser une fois de plus des séquences de touches.

Une première fois pour saisir le texte dans l'espace prévu pour l'édition du message (figure ci-après). Puis une seconde fois pour en valider l'envoi.

```
[void]$MSN.InstantMessage('edouardbracame@gaz.org')
[void][System.Windows.Forms.SendKeys]::SendWait('Bonjour!')
[void][System.Windows.Forms.SendKeys]::SendWait('{enter}')
```

Si vous le souhaitez, il ne vous reste plus qu'à introduire ces trois lignes dans une fonction adaptée vous permettant d'envoyer des messages automatiquement à certains de vos contacts.



Espace d'édition

d. Exporter ses contacts

Pour récupérer la liste de vos contacts, rien de plus simple. La seule propriété `MyContacts` nous en renverra, sous forme d'une collection, la liste complète. Puis, par le biais d'un `Format-Table`, nous affichons les propriétés que nous jugeons intéressantes. Exemple avec un filtre sur les propriétés `SignInName` et `FriendlyName` :

```
PS > $msn.MyContacts | Format-Table SignInName,FriendlyName
```

SignInName	FriendlyName
-----	-----
joebar@gaz.org	Joe
edouardbracame@gaz.org	Ed
paulposichon@gaz.org	Paul

Il ne reste plus qu'à les sauvegarder, par exemple dans un fichier délimité par des virgules, en utilisant la commande **Export-csv**, et le tour est joué.

```
PS > $msn.MyContacts | Select-Object SignInName,FriendlyName |  
Export-Csv -path 'C:\Temp\Mes_Contacts.csv'
```

Ce qui nous donne :

```
#TYPE System.Management.Automation.PSCustomObject  
SignInName,FriendlyName  
joebar@gaz.org,Joe  
edouardbracame@gaz.org,Ed  
paulposichon@gaz.org,Paul
```

Notez que cette fois nous n'utilisons pas `Format-Table`, mais plutôt `Select-Object`. La raison est simple : `Format-Table` est une commandelette faite pour le formatage de données à destination de la console, donc pas adaptée à utilisation d'un autre pipeline. Tandis que `Select-Object` effectue un filtre sur les objets à passer à travers le pipeline.

3. Internet Explorer

Comme pour de nombreuses applications Microsoft, Windows dispose nativement d'un objet COM permettant d'interagir avec le navigateur Internet Explorer. Et pour vous montrer avec quelle aisance cela est possible nous allons décrire comment naviguer sur Internet ou bien afficher une page HTML.

a. Naviguer

Afin de pouvoir utiliser Internet Explorer, il faut dans un premier temps instancier l'objet COM `InternetExplorer.Application.1`. La syntaxe est la suivante :

```
PS > $IE = New-Object -ComObject InternetExplorer.Application.1
```

Maintenant que l'objet est créé, attribuons la valeur `$true` à la propriété `Visible` afin d'afficher le navigateur à l'écran :

```
PS > $IE.Visible = $true
```

L'application Internet Explorer est désormais visible. Poursuivons en lui attribuant également une taille en hauteur ainsi qu'en largeur :

```
PS > $IE.Height = 700 #Attribution de la hauteur
```

```
PS > $IE.Width = 900 #Attribution de la largeur
```



Tout comme pour les Winforms, les tailles attribuées sont données en pixels.

Le navigateur est désormais opérationnel, il ne reste plus qu'à utiliser la méthode `navigate` pour lui indiquer sur quelle page Web se rendre.

```
PS > $IE.Navigate('http://www.powershell-scripting.com')
```



Ouverture de la page Web via PowerShell

b. Afficher une page HTML

Nous venons de voir comment naviguer sur Internet via la méthode `navigate`, mais celle-ci permet également l'affichage de vos propres pages Web. Supposons que nous souhaitions afficher une page HTML éditée avec Word, contenant un message d'information. Et bien là encore commençons par créer et rendre visible notre objet Internet Explorer :

```
PS > $IE = New-Object -ComObject InternetExplorer.Application.1
PS > $IE.Visible = $true
```

Continuons en attribuant une taille appropriée à la fenêtre Internet Explorer de façon à bien calibrer notre texte :

```
PS > $IE.Height = 190 #Attribution de la hauteur
PS > $IE.Width = 550 #Attribution de la largeur
```

De façon à épurer la fenêtre d'Internet Explorer, nous pouvons masquer la barre d'outils. Et pour cela, nous devons attribuer la valeur 0 à la propriété `ToolBar` :

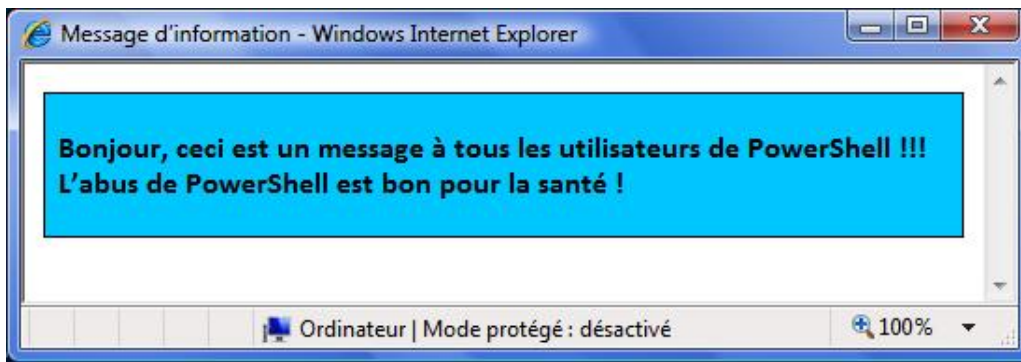
```
PS > $IE.ToolBar = 0
```

Et enfin, pour afficher le message, faisons appel à la méthode `Navigate` avec cette fois comme argument, non pas une adresse URL, mais le chemin complet de votre document HTML :

```
PS > $IE.Navigate('C:\Temp\MessageInformation.htm')
```

Et voilà, il ne reste plus qu'à changer le titre de la fenêtre de façon à le rendre plus explicite et à observer le résultat (figure ci-dessous).

```
PS > $IE.Document.Title = "Message d'information"
```



Affichage du fichier HTML

Comme vous pouvez le constater, nous venons de réaliser une fenêtre graphique contenant un message en seulement quelques lignes de code, alors que la même opération avec la création d'une Winform aurait été beaucoup plus longue et complexe. Ceci étant, l'usage n'est pas le même.

Avant d'en finir, voici le script complet permettant d'afficher une page HTML.

```
#Show-HtmlFile.ps1
# Affichage d'une page HTML

param([string]$fichier,[int]$hauteur,[int]$largeur)
$IE = New-Object -ComObject InternetExplorer.Application.1
$IE.Visible = $true

$IE.Height = $hauteur      #Attribution de la hauteur
$IE.Width = $largeur       #Attribution de la largeur
$IE.Navigate($fichier)     #Affichage de la page HTML
$IE.Document.Title = "Message d'information"
```

Une fois encore, vous avez pu vérifier à quel point il est facile d'interagir avec les applications avec PowerShell, le tout grâce aux objets COM.

4. Windows Script Host (WSH)

Windows Script Host est l'interpréteur de scripts natif des systèmes d'exploitation allant de Windows 98 à Windows Server 2008. Notamment utilisé pour interpréter les scripts VBS (*Visual Basic Script*), WSH est une technologie encore maintenue par Microsoft, qui en plus de son rôle d'interpréteur de scripts, incorpore un ensemble d'objet COM : WshShell, WshNetwork et WshController, tous trois utilisables depuis la console PowerShell bien entendu.

WSH étant bien plus ancien que PowerShell, dans la majorité des cas, les fonctionnalités proposées par ses objets COM sont disponibles nativement avec des objets .NET. Cependant, quelques méthodes restent exclusives aux objets COM de WSH, comme le mappage d'un disque réseau ou l'ajout d'une imprimante elle aussi en réseau.

a. WshShell

L'objet COM WshShell, est une instance du Shell WSH. Ainsi, les utilisateurs de PowerShell que nous sommes, allons pouvoir grâce à cet objet, utiliser des méthodes permettant notamment d'interagir avec la base de registres et le journal d'événements, envoyer des séquences de touches et d'exécuter des programmes. Cette instance du Shell WSH est encore très utilisée dans les scripts VBS, puisque ceux-là ne peuvent s'appuyer sur le Framework .NET.

Comme à son habitude, l'instanciation de l'objet WshShell nécessite la commandelette **New-Object** :

```
PS > $WshShell = New-Object -ComObject Wscript.Shell
```

WScript.Shell étant le ProgID de l'objet WshShell. Une fois l'objet créé, il ne reste plus qu'à en utiliser ses fonctionnalités. Regardons quels sont les membres dont cet objet dispose :

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

```
TypeName: System.__ComObject#{41904400-be18-11d3-
a28b-00104bd35090}
```

Name	MemberType	Definition
----	-----	-----
AppActivate	Method	bool AppActivate (Variant...
CreateShortcut	Method	IDispatch CreateShortcut...
Exec	Method	IWshExec Exec (string)
ExpandEnvironmentStrings	Method	string ExpandEnvironmentS...
LogEvent	Method	bool LogEvent (Variant,...
Popup	Method	int Popup (string, Varia...
RegDelete	Method	void RegDelete (string)
RegRead	Method	Variant RegRead (string)
RegWrite	Method	void RegWrite (strin...
Run	Method	int Run (string, Vari...
SendKeys	Method	void SendKeys (strin...
Environment	Parameterized	Property IWshEnvironment Environ...
CurrentDirectory	Property	string Current Directory ()...
SpecialFolders	Property	IWshCollection SpecialFol...

Exemples d'utilisation :

Internet n'étant pas avare en matière d'exemples d'utilisation de l'objet WshShell, nous ne nous attarderons pas sur les différentes déclinaisons possibles des méthodes proposées par l'objet. Cependant, voici quelques méthodes intéressantes, comme par exemple Popup qui, comme son nom l'indique permet la création d'une fenêtre pop-up, ainsi que Shortcut pour la création d'un raccourci :

Création d'une fenêtre Pop-up

La création d'un pop-up est rendu très simple avec l'objet WshShell, puisque comme vous le montre la figure ci-dessous, il suffit d'instancier l'objet, et d'y appliquer la méthode Popup avec le texte souhaité.

```
PS > $WshShell = New-Object -ComObject Wscript.Shell
PS > $WshShell.Popup('Hello World')
```



Affichage d'un Popup via WshShel

Manipulation de la base de registres

Bien que PowerShell sache gérer de façon native la base de registres avec les commandes ***-item** (cf. chapitre À la découverte de PowerShell, section Navigation dans les répertoires et les fichiers), voici une deuxième solution qui consiste à utiliser les méthodes RegRead et RegWrite pour respectivement lire et écrire dans la base de registres.

Exemple :

```
PS > $WshShell = New-Object -ComObject Wscript.Shell

# Lecture de la clé correspondant à la version du client FTP FileZilla
PS > $WshShell.RegRead('HKLM\SOFTWARE\FileZilla Client\Version')
3.0.1

# Changement de valeur de la clé Version
PS > $WshShell.RegWrite('HKLM\SOFTWARE\FileZilla Client\
Version','4.0','REG_SZ')
```

Création d'un raccourci

Dernier exemple d'utilisation que nous vous proposons, la création d'un raccourci via la méthode CreateShortcut :


```
$cible = 'C:\Temp\Mon_Fichier.doc'
$lien = 'C:\Users\Robin\Desktop\Mon_Raccourci.lnk'
$wshShell = New-Object -ComObject WScript.Shell
$raccourci = $wshShell.CreateShortcut($lien)
$raccourci.Targetpath = $cible
$raccourci.Save()
```

En résumé, l'objet WshShell ne nous apporte rien de réellement nouveau, si ce n'est quelques méthodes comme la création rapide d'un pop-up et d'un raccourci. Cependant, il est toujours rassurant de savoir qu'il existe une multitude de chemins pour arriver à ses fins.

b. WshNetwork

WshNetwork est le second objet COM proposé par WSH. Référencé sous le ProgID WScript.Network, il permet principalement l'ajout et la suppression d'imprimantes et de lecteurs réseaux, ainsi que l'énumération de certaines propriétés comme le nom de l'utilisateur courant, le domaine et le nom de l'ordinateur.

c. Exemples d'utilisation

Mappage d'un disque réseau

Pour mapper un disque réseau, commençons par créer une instance de l'objet WshNetwork.

```
$WshNetwork = New-Object -ComObject Wscript.Network
```

Puis, utilisons la méthode MapNetworkDrive avec les arguments quelle propose :

Argument	Obligatoire	Description
LocalName [string]	Oui	Définit la lettre assignée pour ce lecteur réseau.
RemoteName [string]	Oui	Définit le répertoire partagé.
UdpadeteProfile [Bool]	Non	Indique grâce à un booléen si les informations sur le montage du lecteur réseau doivent être inscrites dans le profil. La valeur par défaut est \$false.
UserName [string]	Non	Spécifie le nom d'utilisateur si le mappage nécessite une authentification.
Password [string]	Non	Spécifie le mot de passe associé au nom d'utilisateur.

Par exemple, la ligne de commande suivante associera, grâce à votre authentification, un lecteur réseau au répertoire \\SERVEUR2008\partage.

```
PS > $WshNetwork.MapNetworkDrive('x:', '\\SERVEUR2008\partage', $false, 'Nom
utilisateur', 'P@ssw0rd')
```

WshNetwork propose également une méthode inverse permettant la déconnexion d'un lecteur réseau. Pour cela, il faut utiliser la méthode RemoveNetworkDrive.

Exemple :

```
PS > $WshNetwork.RemoveNetworkDrive('x:')
```

Connecter une imprimante réseau

Pour connecter une imprimante réseau, commençons par créer une instance de l'objet WshNetwork.

```
PS > $WshNetwork = New-Object -ComObject Wscript.Network
```

Puis, utilisons la méthode AddWindowsPrinterConnection. Par exemple, supposons qu'une imprimante réseau

nommée Imprimante_1 soit située sur le serveur SERVEUR2008. La commande permettant de la connecter est :

```
PS > $WshNetwork.AddWindowsPrinterConnection('\\\\SERVEUR2008\Imprimante_1')
```

Et tout comme pour les lecteurs réseaux, l'objet nous offre également la possibilité de supprimer une connexion d'imprimante par la méthode RemovePrinterConnection.

Exemple :

```
PS > WshNetwork.RemovePrinterConnection('\\\\SERVEUR2008\Imprimante_1')
```

Introduction

Cette partie consacrée à la technologie WMI est très importante car elle permet à la fois de collecter des informations mais aussi d'agir sur toute ou partie du système d'exploitation, du matériel, ainsi que certaines applications. Et ce, que ce soit sur votre machine ou sur n'importe quelle autre machine de votre réseau.

WMI a la réputation d'être compliqué, mais ne vous laissez pas impressionner par tous les acronymes qui vont suivre ; dites-vous qu'avec PowerShell l'accès à WMI n'aura jamais été aussi simple...

Nous n'avons pas la prétention de couvrir entièrement WMI dans cette partie car cette technologie est vaste, si vaste qu'elle a donné naissance à de nombreux ouvrages. Tout au long de cette partie, nous tâcherons d'aller à l'essentiel pour que vous puissiez démarrer rapidement.

Qu'est-ce que WMI ?

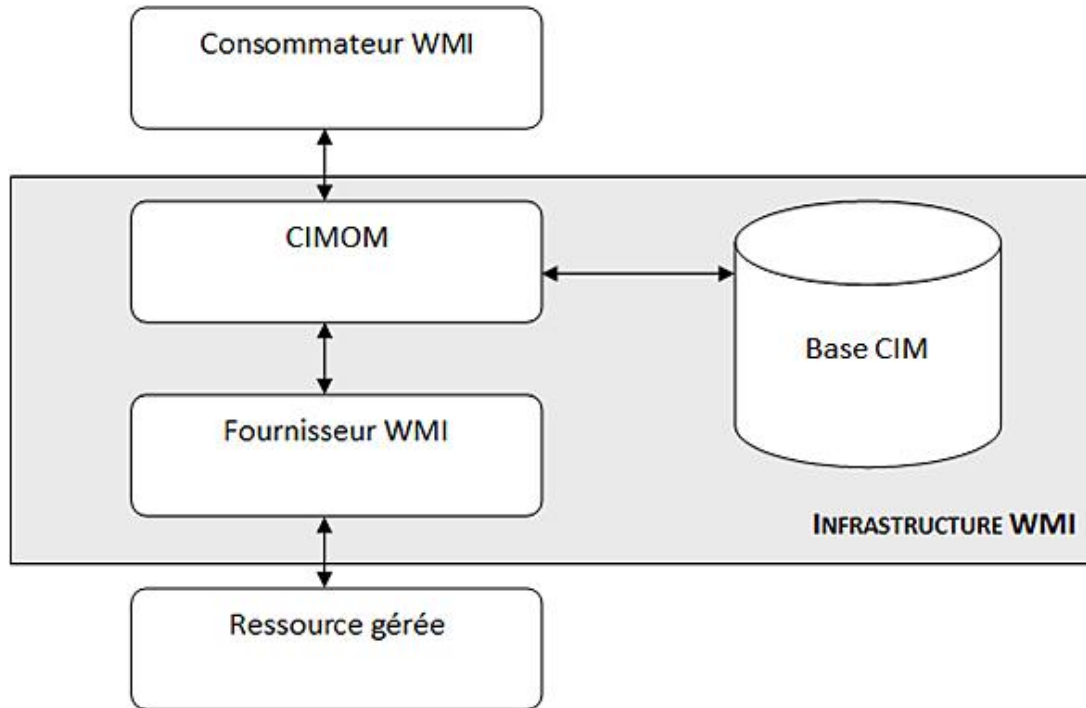
WMI est la mise en œuvre concrète par Microsoft de l'initiative industrielle nommée WBEM (*Web-Based Enterprise Management*). Cette initiative est le fruit d'un travail commun au sein du DMTF (*Distributed Management Task Force*). Le DMTF est un consortium composé d'un grand nombre de sociétés influentes dans le secteur informatique, telles que : HP, IBM, EMC, Cisco, Oracle et Microsoft pour ne citer que les plus grandes.

WBEM est un ensemble de technologies de gestion qui s'appuie sur des standards ouverts de l'Internet dont l'objectif est d'unifier la gestion de plates-formes et technologies disparates évoluant dans un milieu distribué.

WBEM et WMI s'appuient également sur un autre standard, le CIM (*Common Information Model*), qui définit les systèmes, les applications, les réseaux, les périphériques et tout autre composant géré.

Architecture WMI

L'architecture WMI se décompose en trois couches comme dans le schéma suivant :



Un **consommateur WMI** est le terme générique pour désigner l'application qui fait appel à WMI. Cela peut être simplement un script, comme nous en écrivons tant, un outil d'administration, ou bien encore une application d'entreprise telle que Microsoft System Center Configuration Manager 2007.

Une **ressource gérée** peut être n'importe quel composant physique ou logique administrable via WMI. Cela peut être tout composant du système d'exploitation tel que le sous-système disque, les journaux des événements, la base de registres, les services, les processus, etc. La liste est vraiment très longue, c'est incroyable tout ce que l'on peut gérer avec WMI !

Une ressource gérée dialogue avec l'infrastructure WMI exclusivement au travers d'un fournisseur WMI. Chaque ressource ou plutôt chaque classe de ressources est décrite dans un fichier texte au format MOF (*Managed Object Format*). Ce fichier contiendra toutes les propriétés, méthodes et autres informations utiles qui décriront tout ce qu'il sera possible de faire sur une ressource gérée au travers de WMI.

Afin d'être utilisable par l'infrastructure WMI, un fichier MOF doit être compilé ; cela chargera la définition de ressource dans la base CIM. Si vous connaissez SNMP (*Simple Network Management Protocol*), alors sachez qu'un fichier MOF est à WMI ce qu'une MIB (*Management Information Base*) est à SNMP.

L'**infrastructure WMI** est composée des trois composantes suivantes :

- Le **CIMOM** qui signifie *Common Information Model Object Manager*, est tout simplement le service WMI ; service au sens Windows du terme que l'on retrouve dans la liste des services sous la dénomination « Infrastructure de gestion Windows » (winmgmt). Son nom barbare (CIMOM) provient tout droit de l'initiative WBEM. Comme tout service, vous pouvez l'arrêter avec la commande `Stop-Service` et le redémarrer avec `Start-Service`. Le CIMOM a un rôle clé dans l'infrastructure dans la mesure où toute requête WMI passe par lui. C'est lui qui fait l'intermédiaire entre le consommateur et le fournisseur. Cependant il ne traite pas lui-même les requêtes émises par un consommateur mais il les oriente vers le fournisseur approprié. Ainsi, c'est grâce à lui qu'un consommateur peut effectuer les requêtes WMI dans un format homogène en interrogeant sans le savoir plusieurs fournisseurs différents. Le CIMOM sait quel fournisseur interroger car c'est lui qui effectue les enregistrements définitions de ressources/fournisseurs au sein de la base CIM. D'autre part, le CIMOM assure également les fonctions de requêtage WQL (*WMI Query Language*), de sécurité en veillant à ce que les requêtes soient effectuées avec les bons niveaux d'accès, et de gestion des événements.

Grâce au CIMOM, un consommateur WMI va pouvoir souscrire à un événement, et à intervalle de temps défini par le consommateur, le CIMOM va aller chercher les informations auprès du fournisseur qui la possède (un événement représente un changement d'état d'une ressource gérée).

Les événements WMI sont très intéressants car ils permettent une surveillance quasi en temps réel des informations système.

- La **base CIM** ou base WMI contient l'ensemble des classes correspondant aux ressources gérées. Ce concept de classes est exactement le même que celui d'*Active Directory Domain Services* (AD DS). Une classe n'est rien d'autre qu'une description abstraite des propriétés et des fonctionnalités qu'un certain composant logiciel ou matériel possède. Par rapport à AD DS, la différence réside dans le fait qu'il n'y a pratiquement aucune données dans la base CIM. En effet, les informations gérées par l'infrastructure WMI sont des informations dynamiques (par exemple, la quantité de mémoire restante, le taux d'occupation CPU, etc.) qu'il ne serait pas judicieux de placer à l'intérieur d'une base de données. Par conséquent, à chaque requête émise par un consommateur, les fournisseurs sont sollicités.

Dans la base CIM, les classes sont regroupées dans des espaces de noms, et ce dans un souci d'organisation. Car vu le nombre de classes à stocker, l'organisation est impérative ! Par exemple, l'espace de noms root/cimv2 inclut la plupart des classes qui représentent les ressources les plus souvent associées à un ordinateur et à son système d'exploitation.

- Le **fournisseur WMI** est la couche logicielle qui dialogue entre le CIMOM et les ressources gérées. Les fournisseurs dialoguent avec les ressources gérées en utilisant leurs API natives. Ainsi, grâce aux fournisseurs WMI nous n'avons pas besoin de connaître les différentes API correspondant aux différentes ressources. Et c'est précisément cela qui fait la grande force de WMI !

Avec le temps, WMI a pris de plus en plus d'ampleur, et ce n'est pas fini ! Pour vous en convaincre, voici le nombre de fournisseurs supportés par les versions successives de Windows :

- NT4 Server : 15
- Windows 2000 Server : 29
- Windows XP : 50
- Windows Vista : 51
- Windows Server 2003 : 80
- Windows Server 2008 : +100

Un fournisseur est en réalité un composant COM, stocké sur disque sous forme de fichier DLL. Les fournisseurs se trouvent dans le répertoire `%systemroot%\system32\wbem`.

Un peu de vocabulaire

Précédemment nous avons défini ce qu'est une **classe**. Pour mémoire, une classe est la description abstraite des propriétés et méthodes que possède un composant logiciel ou matériel. Les classes WMI sont stockées sous forme binaire dans la base CIM (processus réalisé par la compilation des fichiers MOF). Un exemple de classe pourrait être la classe *Win32_Service*. Celle-ci définit ce qu'est un service au sens générique du terme. Elle possède, entre autres, les propriétés suivantes : *name*, *description*, *status*. Et les méthodes : *startService*, *stopService*, *pauseService*.

Nous parlerons également souvent d'**instance de classe**. Une instance de classe est traditionnellement ce que l'on appelle un objet. Nous utiliserons indépendamment l'un ou l'autre de ces termes (instance ou objet). Typiquement, tous les services Windows que nous avons dans notre système d'exploitation sont des instances de la classe *Win32_Service*. Nous pourrions dire aussi que « les services Windows sont des objets de type *Win32_Service* ».

À la découverte des classes

Le plus difficile dans WMI c'est de savoir quelles sont les catégories d'objets (les classes) du système ou autre ressource sur lesquelles on peut agir. On passe bien souvent plus de temps à explorer WMI à la recherche d'une classe ou d'un membre d'une classe qu'à scripter. En effet, les classes sont extrêmement nombreuses ; il y en a plus de mille réparties dans les différents espaces de noms ! Il y a donc de très fortes chances pour que vous puissiez arriver à vos fins.

Ceci étant, une fois la classe trouvée, il nous faut encore savoir quelles sont les propriétés et méthodes qui la caractérisent afin qu'elle nous soit utile et surtout utilisable par script. Pour découvrir les classes, nous avons deux options possibles :

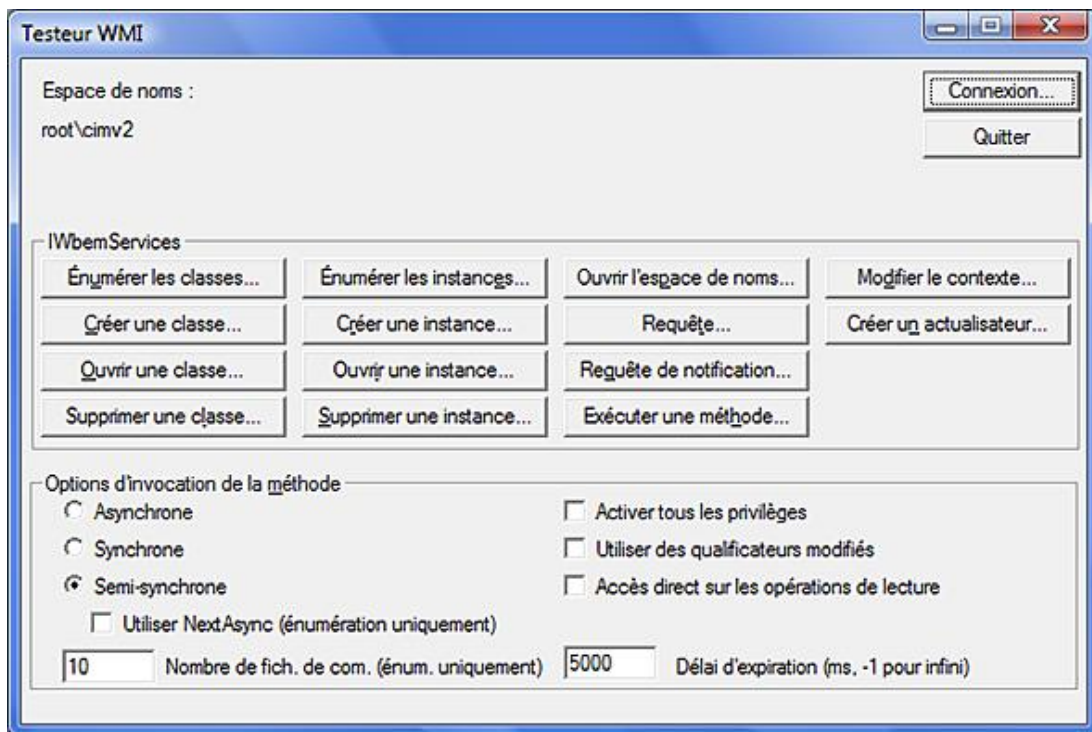
- Explorer la base de connaissance MSDN qui traite de WMI sur Internet.
- Utiliser un outil installé localement pour explorer la base CIM.

L'un n'empêchant pas l'autre, nous vous conseillons d'essayer ces deux options. Concernant les outils d'exploration, Microsoft en met quelques-uns à notre disposition ; ceux-ci nous faciliteront grandement la vie. Les plus connus sont le **testeur WMI** (Wbemtest.exe) et **CIM Studio**.

1. Testeur WMI

Le testeur WMI est fourni en standard dans toutes les versions de Windows depuis Windows NT4 SP4. Grâce à lui vous pouvez de façon graphique explorer le schéma de la base CIM, examiner les définitions de classes, visualiser et agir sur les instances en cours d'exécution. C'est un très bon outil pour commencer à découvrir WMI, et son grand avantage est qu'il est installé sur toutes les machines. Cependant, on ne peut pas dire que son interface graphique soit d'une ergonomie transcendante...

Voici à quoi ressemble son interface graphique :

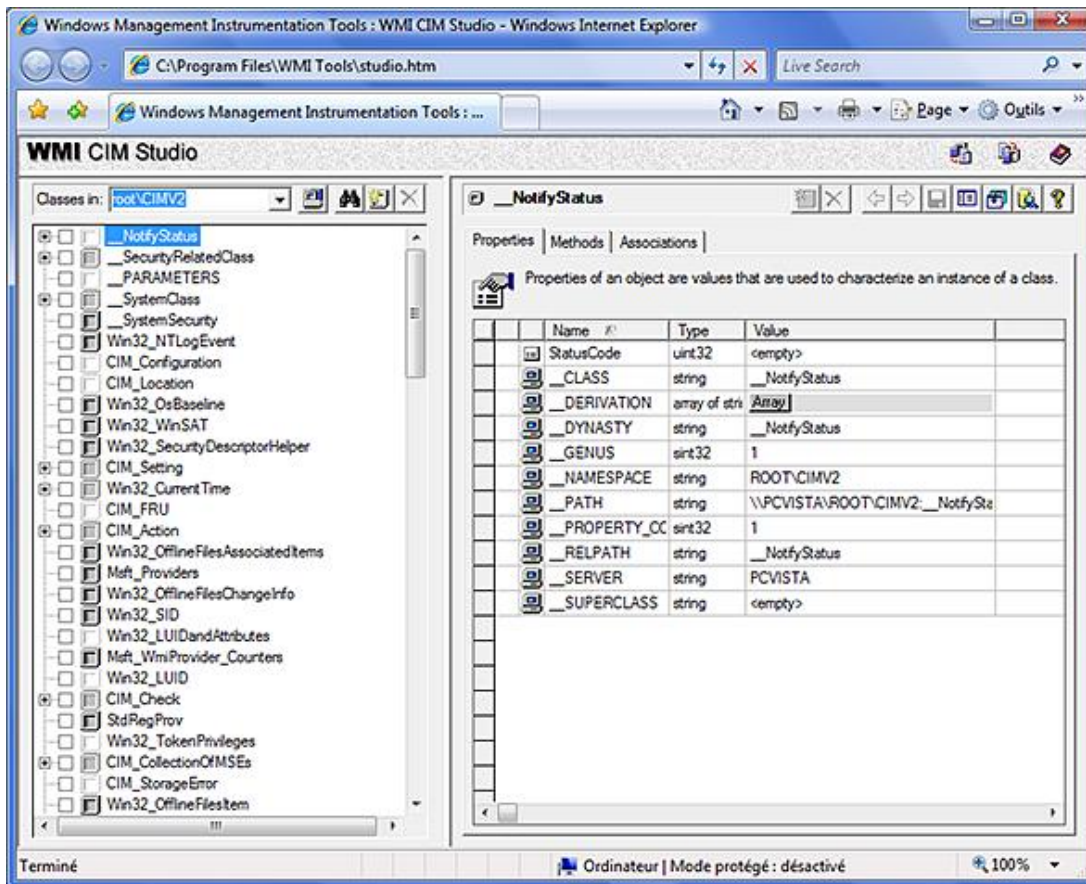


2. CIM Studio

CIM Studio fait partie du kit de développement WMI (WMI SDK) ; c'est une application Web. Le SDK est fourni gratuitement par Microsoft. CIM Studio reprend l'essentiel des fonctionnalités de Wbemtest mais apporte une interface graphique nettement plus ergonomique avec quelques fonctionnalités supplémentaires comme la recherche de classes, et la recherche de propriétés et méthodes.

L'exploration du schéma CIM s'effectue sous une forme arborescente, ce qui est très bien car cela nous permet de découvrir la hiérarchie des classes. D'autre part les carrés foncés en face du nom des classes indiquent que celles-ci possèdent des instances en cours d'exécution. Deux clics supplémentaires et vous avez la liste des instances de cette classe ainsi que toutes leurs propriétés et méthodes, un vrai régal !

Voici une capture d'écran de ce très bon outil :



Un dernier outil incontournable dont nous sommes obligés de vous parler est le **PowerShell WMI Explorer**. Celui-ci est développé par un gourou PowerShell qui se nomme Marc van Orsouw (alias /V\oV\). MOW est un MVP (*Most Valuable Professional*) PowerShell hollandais dont les compétences ne sont plus à démontrer.

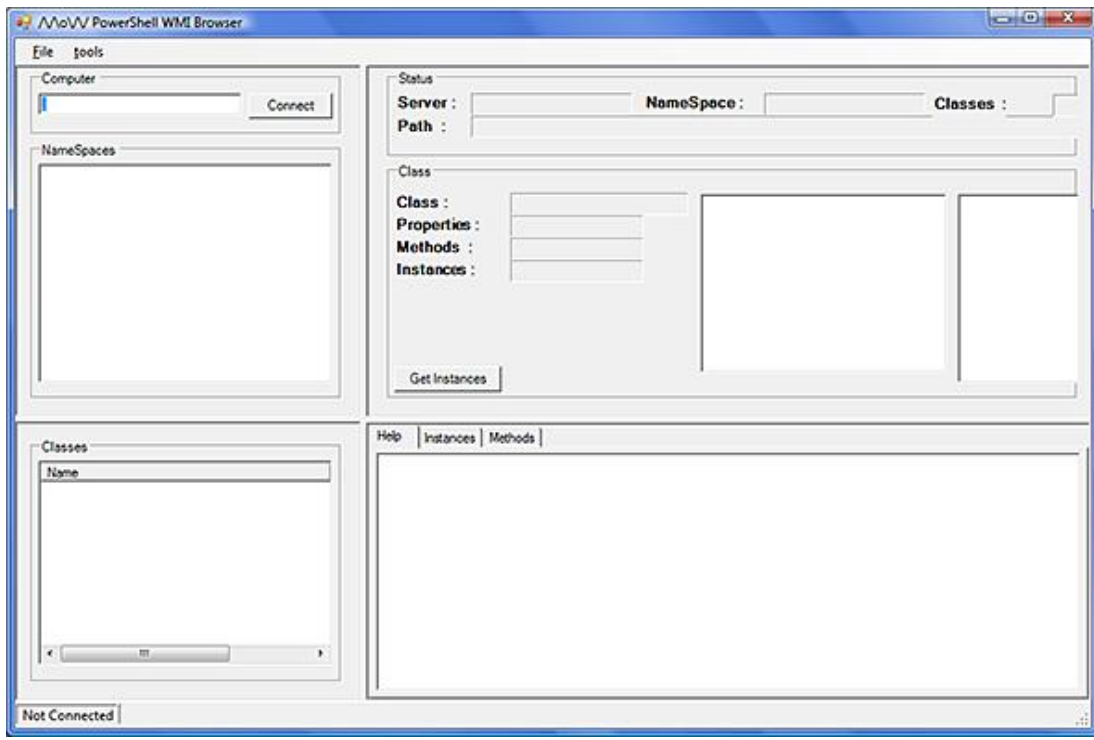
3. PowerShell WMI Explorer

Vous avez certainement dû entendre parler dans une vie antérieure (avant PowerShell) du **Script-o-Matic** ? Pour ceux qui ne savent pas ce qu'est le Script-o-Matic, il s'agit d'un outil d'exploration de la base CIM dont la particularité est de générer directement du code VBScript ou JScript pour faciliter l'utilisation de WMI dans le monde du scripting. Le Script-o-Matic est développé par les Scripting Guys de chez Microsoft. Bien qu'excellent, cet outil a la fâcheuse lacune de ne pas générer de scripts PowerShell. Qu'à cela ne tienne, MOW a développé son propre outil, le WMI Explorer, qui est en quelque sorte un clone du Script-o-Matic, mais adapté à PowerShell et écrit entièrement en ...PowerShell ! C'est une prouesse technique exceptionnelle !!! Nous vous encourageons vivement à l'utiliser et à en abuser !

Vous le trouverez en libre téléchargement à l'adresse suivante :

<http://thepowershellguy.com/blogs/posh/archive/tags/WMI+Explorer/default.aspx>

Voici à quoi ressemble l'interface graphique du PowerShell WMI Explorer :



Premiers pas dans l'écriture de scripts WMI

Assez parlé passons à présent à l'action ! PowerShell possède dans son jeu de commandelettes, la commandelette **Get-WmiObject** (alias : **gwmi**). C'est grâce à elle que nous allons pouvoir dialoguer avec la couche WMI de notre système ou d'un système distant. Vous verrez que c'est d'une étonnante facilité.

Voici les paramètres les plus couramment utilisés (hors paramètres communs) de la commandelette :

Paramètre	Description
Class <String>	Permet de définir le nom de la classe dont on souhaite récupérer les instances.
Property <String>	Permet de définir le nom de la propriété ou jeu de propriétés à récupérer.
Namespace <String>	Permet de définir l'espace de nom dans lequel se trouve la classe. La valeur par défaut est root\cimv2.
Query <String>	Permet de définir la requête à exécuter en utilisant le langage de requête WQL.
ComputerName <String>	Permet de définir le nom de l'ordinateur sur lequel s'applique la commande. Par défaut la valeur est l'ordinateur local (valeur « . »).
Filter <String>	Permet de définir une clause « Where » au format WQL.
Credential <PSCredential>	Permet de fournir les informations d'authentification si la commande doit s'effectuer avec un autre compte que le compte courant.
List [<SwitchParameter>]	Permet de lister les classes WMI. Cette propriété fonctionne de concert avec -namespace . Si namespace est omis, alors travaille dans l'espace de noms root\cimv2.

1. Lister les classes

Nous avons vu qu'il existait un certain nombre d'outils nous permettant de trouver des classes dans notre système. Cependant, PowerShell sait très bien le faire également, peut-être d'une façon un peu moins conviviale qu'une interface graphique, mais cela reste une affaire de goûts. Regardons le résultat de la commande suivante :

PS > Get-WmiObject -list		
...		
Name	Methods	Properties
----	-----	-----
Win32_CurrentTime	{}	{Day, DayOfWeek, Ho...
Win32_LocalTime	{}	{Day, DayOfWeek, Ho...
Win32_UTCTime	{}	{Day, DayOfWeek, Ho...
Win32_NTLogEvent	{}	{Category, Category...
CIM_ManagedSystemElement	{}	{Caption, Descripti...
CIM_LogicalElement	{}	{Caption, Descripti...
CIM_OperatingSystem	{Reboot, Shutdown}	{Caption, CreationC...
Win32_OperatingSystem	{Reboot, Shutdown...	{BootDevice, BuildN...
CIM_Process	{}	{Caption, CreationC...
Win32_Process	{Create, Terminat...	{Caption, CommandLi...
CIM_System	{}	{Caption, CreationC...
CIM_ComputerSystem	{}	{Caption, CreationC...
CIM_UnitaryComputerSystem	{SetPowerState}	{Caption, CreationC...
Win32_ComputerSystem	{SetPowerState, R...	{AdminPasswordStatu...
CIM_ApplicationSystem	{}	{Caption, CreationC...
Win32_NTDomain	{}	{Caption, ClientSit...
CIM_SoftwareElement	{}	{BuildNumber, Capti...
CIM_BIOSElement	{}	{BuildNumber, Capti...

...

Nous n'afficherons pas tous les résultats car cela occuperait des pages et des pages entières. En effet, il y a un nombre faramineux de classes qu'il serait intéressant justement de compter en écrivant ceci :

```
PS > (Get-WmiObject -list).count
965
```

Nous avons, avec Windows Vista, neuf cent soixante-cinq classes à notre portée ! Vous devez certainement vous dire « mais il en manque, on nous a dit qu'il y en avait plus de mille ! ». Cela est tout à fait exact, mais comme vous aurez pu le remarquer dans l'explication sur l'usage de la commandelette Get-WmiObject, comme nous n'avons pas précisé d'espace de noms avec le paramètre **-namespace**, c'est donc **root/cimv2** qui a été utilisé par défaut. Si nous additionnions les classes des autres espaces de noms, nous aurions bien plus de mille classes.



Le nombre de classes WMI est en constante augmentation. Nous ne connaissons pas leur nombre exact au sein des différents systèmes d'exploitation Microsoft, mais sachez que pour chaque nouvelle version d'OS le nombre de classes augmente. Cela prouve, si tant est qu'il était nécessaire de le faire, que WMI est une technologie très importante qui nous permet d'effectuer de plus en plus de tâches.

2. Rechercher une classe

Maintenant que nous savons lister toutes les classes disponibles du système, il va nous falloir appliquer un filtre sur le résultat afin de trouver celle que l'on recherche. Par exemple, imaginons que nous souhaitions obtenir la date et l'heure d'une machine distante afin de vérifier si celle-ci est correcte. Spontanément nous pourrions penser à une classe dont le nom contiendrait le mot « date » ou « time ». Essayons donc de filtrer sur ces mots :

```
PS > Get-WmiObject -list | Where {$_.name -match 'date'}
```

Pas de chance, un filtre sur « date » ne nous retourne aucune classe. Essayons maintenant de filtrer sur « time » :

```
PS > Get-WmiObject -list | Where {$_.name -match 'time'}

NameSpace: ROOT\cimv2

Name                                Methods Properties
----
__TimerNextFiring                   {}          {NextEvent64BitTime, TimerId}
MSFT_NetConnectionTimeout           {}          {Milliseconds, SECURITY_DESCRIPTOR, Servi...
MSFT_NetTransactTimeout              {}          {Milliseconds, SECURITY_DESCRIPTOR, Servi...
MSFT_NetReadfileTimeout              {}          {Milliseconds, SECURITY_DESCRIPTOR, TIME...
__TimerEvent                         {}          {NumFirings, SECURITY_DESCRIPTOR, TIME_CR...
__TimerInstruction                   {}          {SkipIfPassed, TimerId}
__AbsoluteTimerInstruction            {}          {EventDateTime, SkipIfPassed, TimerId}
__IntervalTimerInstruction            {}          {IntervalBetweenEvents, SkipIfPassed, Tim...
Win32_CurrentTime                    {}          {Day, DayOfWeek, Hour, Milliseconds...}
Win32_LocalTime                      {}          {Day, DayOfWeek, Hour, Milliseconds...}
Win32_UTCTime                        {}          {Day, DayOfWeek, Hour, Milliseconds...}
Win32_TimeZone                       {}          {Bias, Caption, DaylightBias, DaylightDay...
Win32_SystemTimeZone                 {}          {Element, Setting}
```

Cette fois la pêche est bonne et nous avons obtenu une dizaine de classes. Notre sens aigu du discernement, de part une certaine expérience, nous pousse à lister les instances de la classe **Win32_UTCTime** :

```
PS > Get-WmiObject Win32_UTCTime

__GENUS          : 2
__CLASS           : Win32_UTCTime
__SUPERCLASS      : Win32_CurrentTime
__DYNASTY         : Win32_CurrentTime
__RELPATH         : Win32_UTCTime=@
__PROPERTY_COUNT  : 10
__DERIVATION      : {Win32_CurrentTime}
__SERVER          : WIN7_BUREAU
__NAMESPACE       : root\cimv2
__PATH            : \\WIN7_BUREAU\root\cimv2:Win32_UTCTime=@
```

```
Day : 18
DayOfWeek : 0
Hour : 9
Milliseconds :
Minute : 47
Month : 10
Quarter : 4
Second : 23
WeekInMonth : 4
Year : 2009
```

Nous avons réussi à déterminer dans quelle classe WMI se « cache » l'heure et la date (UTC (*Universal Time Coordinated*)) de notre système. Ne reste maintenant plus qu'à envoyer cette requête à une machine distante pour terminer notre exemple. Pour ce faire, il va nous suffire d'ajouter simplement le paramètre **-computer** à notre requête et le tour sera joué :

```
PS > Get-WmiObject Win32_UTCTime -computer W2K8R2SRV

__GENUS : 2
__CLASS : Win32_UTCTime
__SUPERCLASS : Win32_CurrentTime
__DYNASTY : Win32_CurrentTime
__RELPATH : Win32_UTCTime=@
__PROPERTY_COUNT : 10
__DERIVATION : {Win32_CurrentTime}
__SERVER : W2K8R2SRV
__NAMESPACE : root\cimv2
__PATH : \\W2K8R2SRV\root\cimv2:Win32_UTCTime=@
Day : 18
DayOfWeek : 0
Hour : 9
Milliseconds :
Minute : 50
Month : 10
Quarter : 4
Second : 8
WeekInMonth : 4
Year : 2009
```



Pour exécuter une requête WMI à distance vous devez être membre du groupe Administrateur local de la machine distante ou Administrateur du domaine.



Cependant nous pouvons aussi spécifier des credentials différents lors de l'exécution de la requête WMI en spécifiant le paramètre **-credential**, comme ceci :

```
PS > Get-WmiObject Win32_UTCTime -computer W2K8R2SRV -credential (Get-Credential)
```

Une boîte de dialogue s'ouvrira alors et vous pourrez saisir le login et le mot de passe administrateur de la machine distante.

3. Rechercher une propriété

Parfois il est encore difficile de trouver l'information qui nous intéresse, et ce même en recherchant du mieux que l'on peut sur le nom d'une classe WMI. Dans ce cas, il va nous falloir tenter d'identifier des noms de propriétés qui pourraient être pertinentes. En d'autres termes, nous allons passer au peigne fin le nom des propriétés de toutes les classes WMI d'un espace de noms.

Dans cet exemple, nous souhaiterions obtenir la taille de la mémoire de notre ordinateur. Comme la recherche sur le nom des classes ne donne rien, nous partirons donc cette fois à la recherche d'une propriété dont le nom contient « memory ». Pour tenter d'y parvenir, essayons cette petite ligne de commande fort sympathique :

```
PS > Get-WmiObject -namespace root/cimv2 -list -recurse |
foreach{$_ .PSBase.properties} | where {$_.name -match 'memory'} |
Select-Object origin, name -unique
```

Origin	Name
----	----
CIM_OperatingSystem	FreePhysicalMemory
CIM_OperatingSystem	FreeVirtualMemory
CIM_OperatingSystem	MaxProcessMemorySize
CIM_OperatingSystem	TotalVirtualMemorySize
CIM_OperatingSystem	TotalVisibleMemorySize
Win32_ComputerSystem	TotalPhysicalMemory
CIM_VideoController	MaxMemorySupported
CIM_VideoController	VideoMemoryType
Win32_DeviceMemoryAddress	MemoryType
CIM_PhysicalMemory	MemoryType
Win32_PhysicalMemoryArray	MemoryDevices
Win32_PhysicalMemoryArray	MemoryErrorCorrection
Win32_NamedJobObjectActgInfo	PeakJobMemoryUsed
Win32_NamedJobObjectActgInfo	PeakProcessMemoryUsed
Win32_WinSAT	MemoryScore
Win32_NamedJobObjectLimitSetting	JobMemoryLimit
Win32_NamedJobObjectLimitSetting	ProcessMemoryLimit
Win32_NetworkAdapterConfiguration	ForwardBufferMemory
CIM_MemoryCheck	MemorySize
CIM_MemoryCapacity	MaximumMemoryCapacity
CIM_MemoryCapacity	MemoryType
CIM_MemoryCapacity	MinimumMemoryCapacity
Win32_PerfFormattedData_AuthorizationManag...	NumberOfScopesloadedinmemory
Win32_PerfRawData_AuthorizationManag...	NumberOfScopesloadedinmemory
...	

Bien que cette commande mette un peu de temps à s'exécuter, et cela est tout à fait normal dans la mesure où il y a un grand nombre de classe à analyser, elle nous retourne toutes les propriétés répondant au critère de notre recherche. De plus, elle nous affiche le nom de la classe dont est issue la propriété retournée, ce qui nous facilitera grandement la vie par la suite comme nous le verrons dans le prochain exemple.

Notez que nous avons dû utiliser la propriété PSBase, qui pour mémoire, permet de passer outre l'adaptation des types PowerShell par défaut (cf. Chapitre Maîtrise du shell - Ajout de méthodes et propriétés personnalisées). Ainsi nous pouvons obtenir la propriété Origin qui nous permet de savoir dans quelle classe se trouve la propriété recherchée.

4. Récupération de la taille de la mémoire physique

Maintenant que nous avons connaissance de la classe à utiliser pour déterminer la quantité de mémoire d'un ordinateur. Faisons-y appel avec la propriété **TotalPhysicalMemory**.

```
PS > $machine = Get-WmiObject Win32_ComputerSystem
PS > $machine.TotalPhysicalMemory
2136428544
```

Nous aurions pu également écrire ceci :

```
PS > (Get-WmiObject Win32_ComputerSystem).TotalPhysicalMemory
2136428544
```

Cette formulation, bien que plus concise est un peu moins lisible.

La taille nous est retournée en octets, pour la convertir en méga-octets nous allons la diviser par 1024*1024 ou mieux par le quantificateur de taille **1MB**.

```
PS > (Get-WmiObject Win32_ComputerSystem).TotalPhysicalMemory / 1MB
2037,45703125
```

Nous avons donc 2037 Mo de RAM installés dans notre système, soit approximativement 2 Go de mémoire. Nous n'avons pas exactement 2048 Mo disponibles car nous utilisons la carte graphique intégrée sur la carte mère et celle-ci se réserve quelques Mo de mémoire vive pour fonctionner.

Pour arrondir le résultat retourné, nous pouvons faire appel à la méthode statique **round** de classe **math** du Framework .NET.

```
PS > [math]::round((Get-WmiObject `
Win32_ComputerSystem).TotalPhysicalMemory / 1MB)
2037
```

Pour faire de même pour un ordinateur distant, il suffit simplement d'ajouter le paramètre **-computer** et de spécifier le nom de la machine distante, et c'est tout !

```
PS > (Get-WmiObject Win32_ComputerSystem `
-computer machineD distante).TotalPhysicalMemory / 1MB
```

Vous pouvez constater l'exceptionnelle concision de nos commandes. Ceux d'entre vous qui ont pratiqué le VBScript dans une vie antérieure l'auront forcément remarqué !

5. Récupération d'informations système

Continuons nos investigations dans WMI à la recherche d'informations système plus générales que la mémoire, et ce, toujours avec la classe Win32_ComputerSystem. Regardons les informations qu'elle peut nous rapporter :

```
PS > Get-WmiObject Win32_ComputerSystem

Domain                : WORKGROUP
Manufacturer          : Gigabyte Technology Co., Ltd.
Model                 : G33M-DS2R
Name                  : WIN7_BUREAU
PrimaryOwnerName      : Arnaud
TotalPhysicalMemory    : 2136428544
```

Nous obtenons en retour de cette commande un certain nombre d'informations utiles. Ces informations ne sont qu'un petit échantillon des propriétés de la classe Win32_ComputerSystem. En effet celle-ci en possède plus de cinquante. Alors pourquoi ne les voyons-nous pas ?

Tout simplement parce que PowerShell applique par défaut les vues prédéfinies pour les objets WMI et notamment pour la classe Win32_ComputerSystem. Pour le vérifier tapez la commande suivante :

```
PS > Set-Location $pshome
PS > Select-String -path *.ps1xml -pattern 'win32_Computersystem'

types.ps1xml:738:      <Name>System.Management.ManagementObject
                        #root\cimv2\Win32_ComputerSystem</Name>
types.ps1xml:840:      <Name>System.Management.ManagementObject
                        #root\cimv2\Win32_ComputerSystemProduct</Name>
```

Cette commande nous indique que dans le fichier **types.ps1xml**, à la ligne 738 se trouve la définition du type de la classe Win32_ComputerSystem.

Comme d'habitude, pour passer outre l'affichage par défaut, nous pouvons écrire la commande suivante :

```
PS > Get-WmiObject Win32_ComputerSystem | Format-List *

AdminPasswordStatus      : 3
BootupState              : Normal boot
ChassisBootupState       : 2
KeyboardPasswordStatus   : 3
PowerOnPasswordStatus    : 3
PowerSupplyState         : 2
PowerState               : 0
FrontPanelResetStatus    : 3
ThermalState             : 2
Status                   : OK
Name                     : WIN7_BUREAU
PowerManagementCapabilities :
PowerManagementSupported  :
__GENUS                  : 2
__CLASS                   : Win32_ComputerSystem
__SUPERCLASS              : CIM_UnitaryComputerSystem
__DYNASTY                  : CIM_ManagedSystemElement
```

```

__RELPATH           : Win32_ComputerSystem.Name="WIN7_BUREAU"
__PROPERTY_COUNT    : 58
__DERIVATION         : {CIM_UnitaryComputerSystem, CIM_ComputerSystem,
                        CIM_System, CIM_LogicalElement...}
__SERVER            : WIN7_BUREAU
__NAMESPACE         : root\cimv2
__PATH              : \\WIN7_BUREAU\root\cimv2:Win32_ComputerSystem.
                        Name="WIN7_BUREAU"
AutomaticManagedPagefile : True
AutomaticResetBootOption : True
AutomaticResetCapability : True
BootOptionOnLimit       : 
BootOptionOnWatchDog    : 
BootROMSupported        : True
Caption                : WIN7_BUREAU
CreationClassName       : Win32_ComputerSystem
CurrentTimeZone         : 120
DaylightInEffect        : True
Description             : AT/AT COMPATIBLE
DNSHostName            : Win7_Bureau
Domain                 : WORKGROUP
DomainRole              : 0
EnableDaylightSavingsTime : True
InfraredSupported       : False
InitialLoadInfo         : 
InstallDate            : 
LastLoadInfo           : 
Manufacturer           : Gigabyte Technology Co., Ltd.
Model                  : G33M-DS2R
NameFormat              : 
NetworkServerModeEnabled : True
NumberOfLogicalProcessors : 2
NumberOfProcessors      : 1
OEMLogoBitmap          : 
OEMStringArray          : 
PartOfDomain            : False
PauseAfterReset         : -1
PCSystemType           : 1
PrimaryOwnerContact     : 
PrimaryOwnerName        : Arnaud
ResetCapability         : 1
ResetCount              : -1
ResetLimit              : -1
Roles                   : {LM_Workstation, LM_Server, Print, NT...}
SupportContactDescription : 
SystemStartupDelay      : 
SystemStartupOptions    : 
SystemStartupSetting    : 
SystemType              : X86-based PC
TotalPhysicalMemory     : 2136428544
UserName                : Win7_Bureau\Arnaud
WakeUpType              : 6
Workgroup               : WORKGROUP
Scope                   : System.Management.ManagementScope
Path                    : \\WIN7_BUREAU\root\cimv2:Win32_ComputerSystem.
                        Name="WIN7_BUREAU"
Options                 : System.Management.ObjectGetOptions
ClassPath               : \\WIN7_BUREAU\root\cimv2:Win32_ComputerSystem
Properties               : {AdminPasswordStatus, AutomaticManagedPagefile,
                        AutomaticResetBootOption,
                        AutomaticResetCapability, ...}
SystemProperties         : {__GENUS, __CLASS, __SUPERCLASS, __DYNASTY...}
Qualifiers              : {dynamic, Locale, provider, UUID}
Site                    : 
Container               : 

```

À la vue de cette liste, vous comprenez mieux pourquoi l'affichage par défaut se contente de n'afficher qu'un petit échantillon des propriétés les plus significatives.

Nous venons de lister les propriétés et leurs valeurs, regardons à présent quelles sont les méthodes de cet objet. Et

pour cela, nous allons utiliser la commandelette `Get-Member -MemberType method`.

```
PS > Get-WmiObject Win32_ComputerSystem | Get-Member -MemberType method
```

```
    TypeName: System.Management.ManagementObject#root\cimv2\
Win32_ComputerSystem
```

Name	MemberType	Definition
JoinDomainOrWorkgroup	Method	System.Management.ManagementBaseObj...
Rename	Method	System.Management.ManagementBaseObj...
SetPowerState	Method	System.Management.ManagementBaseObj...
UnjoinDomainOrWorkgroup	Method	System.Management.ManagementBaseObj...

Grâce à `Get-Member` nul besoin d'aller explorer le schéma des objets avec un quelconque outil car PowerShell le fait pour nous ! Il nous retourne aussi la définition de chaque méthode pour nous aider à les utiliser. Cela est bien pour nous donner une idée de leur utilisation, mais lorsqu'il s'agit de « méthodes à risques » nous vous conseillons tout de même d'aller prendre des informations plus détaillées sur MSDN.

Nous pouvons remarquer au passage le *TypeName* de notre objet. Nous voyons qu'il provient de l'espace de nom **root\cimv2**. Encore une fois, si l'on ne précise pas l'espace de noms, c'est dans celui-ci que PowerShell va chercher tous les objets.

6. Agir sur le système en utilisant des méthodes WMI

Jusqu'à présent avec WMI, nous avons cherché à récupérer des informations. Autrement dit nous n'avons fait qu'employer des propriétés d'objets. Objets auxquels nous nous sommes connectés grâce à la commandelette `Get-WmiObject`.

Agir sur le système est synonyme d'appliquer des méthodes à des objets WMI. Bien que cela soit possible depuis la version 1 de PowerShell, nous allons voir que la version 2 simplifie encore l'usage de WMI.

a. Appel de méthodes conventionnelles

En prêtant attention à l'exemple précédent, nous avons découvert quatre méthodes (**JoinDomainOrWorkgroup**, **Rename**, **SetPowerState** et **UnjoinDomainOrWorkgroup**) pour agir sur notre objet.

```
PS > Get-WmiObject Win32_ComputerSystem | Get-Member -MemberType method
```

```
    TypeName: System.Management.ManagementObject#root\cimv2\
Win32_ComputerSystem
```

Name	MemberType	Definition
JoinDomainOrWorkgroup	Method	System.Management.ManagementBaseObj...
Rename	Method	System.Management.ManagementBaseObj...
SetPowerState	Method	System.Management.ManagementBaseObj...
UnjoinDomainOrWorkgroup	Method	System.Management.ManagementBaseObj...

Voyons à présent comment en utiliser une parmi ces quatre. Prenons par exemple celle qui nous permet de faire adhérer une machine à un domaine.

Après renseignements pris auprès de MSDN sur le fonctionnement de la méthode **JoinDomainOrWorkgroup**, nous pouvons écrire les deux lignes suivantes pour faire adhérer notre machine au domaine *ps-scripting.com* :

```
PS > $machine = Get-WmiObject Win32_ComputerSystem
PS > $machine.JoinDomainOrWorkgroup('ps-scripting.com', 'P@ssw0rd', `
    'administrateur@ps-scripting.com', $null, 3)
```

```
__GENUS          : 2
__CLASS           : __PARAMETERS
__SUPERCLASS      :
__DYNASTY         : __PARAMETERS
__RELPATH         :
__PROPERTY_COUNT  : 1
__DERIVATION      : {}
__SERVER          :
```

```
__NAMESPACE      :  
__PATH           :  
ReturnValue      : 0
```

Le premier argument est le nom complet du domaine cible, suivi du mot de passe d'un compte ayant les droits d'entrer des machines dans un domaine, suivi du compte sous la forme *Domaine\User* ou *User@Domaine*, suivi de *\$null* pour ne pas indiquer d'unité d'organisation de destination particulière, suivi de la valeur 3 pour spécifier qu'il s'agit d'une adhésion au domaine avec création du compte d'ordinateur.

Comme l'opération a fonctionné, nous avons une valeur de retour (**ReturnValue**) de zéro.

Il ne reste à présent plus qu'à redémarrer l'ordinateur pour terminer l'adhésion au domaine. Redémarrage que nous pourrions faire ainsi localement :

```
PS > $machine = Get-WmiObject Win32_OperatingSystem  
PS > $machine.Reboot()
```

Ou à distance comme ceci :

```
PS > $machine =  
Get-WmiObject Win32_OperatingSystem -computer NomDeMachineDistante  
PS > $machine.Reboot()
```

À nouveau, une valeur de zéro pour la propriété **ReturnValue** signifie que l'opération s'est déroulée normalement.



Sur les systèmes d'exploitation Windows 7 ou Windows Server 2008 R2 il est nécessaire d'ajouter le paramètre `-EnableAllPrivileges` à la commande `Get-WmiObject`.

b. Appel de méthodes avec `Invoke-WmiMethod`

Nous vous le disions au début de cette partie, PowerShell v2 simplifie l'emploi des méthodes grâce à l'apport de la commande `Invoke-WmiMethod`.

Voici les paramètres les plus couramment utilisés (hors paramètres communs) de la commande :

Paramètre	Description
ArgumentList <Object[]>	Spécifie les paramètres à passer à la méthode appelée. La valeur de ce paramètre doit être un tableau d'objets et ils doivent apparaître dans l'ordre requis par la méthode appelée.
AsJob [<SwitchParameter>]	Exécute la commande en tant que tâche en arrière-plan. Utilisez ce paramètre pour exécuter des commandes dont l'exécution nécessite beaucoup de temps.
Authentication <AuthenticationLevel>	Spécifie le niveau d'authentification à utiliser avec la connexion WMI
Authority <String>	Spécifie l'autorité à utiliser pour authentifier la connexion WMI. Vous pouvez spécifier l'authentification Kerberos ou NTLM standard.
Class <String>	Spécifie la classe WMI qui contient une méthode statique à appeler.
ComputerName <String[]>	Spécifie l'ordinateur sur lequel vous voulez exécuter l'opération de gestion. La valeur peut être un nom de domaine complet, un nom NetBIOS ou une adresse IP.
Credential <PSCredential>	Spécifie un compte d'utilisateur qui a l'autorisation d'exécuter cette action. La valeur par défaut est l'utilisateur actuel.
EnableAllPrivileges [<SwitchParameter>]	Active tous les privilèges de l'utilisateur actuel avant que la commande ne passe l'appel WMI.
Impersonation <ImpersonationLevel>	Spécifie le niveau d'emprunt d'identité à utiliser.
InputObject <ManagementObject>	Spécifie un objet <code>ManagementObject</code> à utiliser en entrée. Lorsque ce

	paramètre est utilisé, tous les autres paramètres sont ignorés.
Locale <String>	Spécifie les paramètres régionaux par défaut pour les objets WMI. Spécifiez la valeur du paramètre Locale sous forme de tableau au format MS_<LCID> dans l'ordre de préférence.
Name <String>	Spécifie le nom de la méthode à appeler. Ce paramètre est obligatoire et ne peut ni avoir la valeur Null ni être vide.
Namespace <String>	Lorsqu'il est utilisé avec le paramètre Class, ce paramètre spécifie l'espace de noms du répertoire de stockage WMI dans lequel figure la classe ou l'objet WMI référencé.
Path <String>	Spécifie le chemin d'accès de l'objet WMI d'une classe WMI, ou spécifie le chemin d'accès de l'objet WMI d'une instance d'une classe WMI. La classe ou l'instance que vous spécifiez doit contenir la méthode spécifiée dans le paramètre Name.
ThrottleLimit <Int>	Permet à l'utilisateur de spécifier une valeur de limitation pour le nombre d'opérations WMI pouvant être exécutées simultanément. Ce paramètre est utilisé avec le paramètre AsJob. La limite d'accélération s'applique uniquement à la commande actuelle, et non à la session ou à l'ordinateur.

Nous pouvons aussi effectuer un reboot avec cette fois-ci l'instruction Invoke-WmiMethod :

```
PS > $machine = Get-WmiObject Win32_OperatingSystem -computer
NomDeMachineDistant | Invoke-WmiMethod -name reboot
```

Pour que le reboot soit effectif, et vous éviter l'erreur « *The RPC server is unavailable. (Exception from HRESULT: 0x800706BA)* », vous devez vous assurer que le pare-feu de la machine distante autorise les appels WMI. Pour ce faire, sur la machine gérée, autorisez le programme nommé « Infrastructure de gestion Windows (WMI) ».

Voici d'autres petits exemples d'utilisation de Invoke-WmiMethod, comment créer un processus :

```
PS > Invoke-WmiMethod -path Win32_Process -name create -argumentlist
notepad.exe
```

Ou comment ajouter une connexion à une imprimante réseau :

```
PS > Invoke-WmiMethod -path Win32_Printer -name AddPrinterConnection `
-argumentlist \\monServeur\printer1
```

Ou encore définir l'imprimante « Fax » par défaut :

```
PS > Get-WmiObject -class Win32_Printer -filter "Name='Fax'" |
Invoke-WmiMethod -name setDefaultPrinter
```

7. Utilisation de filtres WMI avec WQL

Il ne serait pas question de requêtes WMI s'il n'existait de langage de requête. WMI apporte donc le langage WQL (*WMI Query Language*). Le WQL est un sous ensemble simplifié du SQL ANSI (*Structured Query Language*) bien connu des administrateurs de bases de données. WQL permet seulement la récupération d'informations ; il ne possède pas de fonctions de modification ou de suppression de données. Il ne permet pas non plus de trier les informations retournées par WMI, mais cela est facilement réalisable grâce à PowerShell et à ses fonctions de tri natives (Sort-Object).

L'utilisation de filtres est intéressante lorsque l'on manipule un volume important de données. En effet, quand on utilise Get-WmiObject *maClasse*, ceci nous retourne toutes les instances de la classe passée en argument. Bien que pratique du fait de la concision de la commande, cela peut faire transiter beaucoup d'informations sur le réseau et nécessiter du temps de traitement côté client. Surtout si l'on ne recherche qu'une propriété d'une instance particulière.

À retenir :

L'objectif des filtres est justement d'effectuer un pré-traitement côté serveur et de ne retourner que les informations nécessaires au client (ou consommateur).

Get-WmiObject nous permet de construire un filtre avec les deux paramètres suivants :

- **-Property**, pour ne récupérer que la ou les propriétés spécifiées.
- **-Query**, pour effectuer une requête WQL.

a. Interroger le journal des événements d'une machine distante

Bien que nous ayons déjà traité cet exemple avec .NET, nous pouvons également le faire avec WMI. Cela démontre que les technologies .NET, COM, ADSI et WMI se chevauchent et qu'il existe de nombreuses façons de traiter un problème. Néanmoins, le journal des événements reste un excellent candidat pour l'illustration de nos propos sur WMI...

Même si la commande Get-Eventlog existe dans le jeu standard de PowerShell, avec PowerShell v1 elle ne permet que de manipuler les journaux de la machine locale. On ne peut pas récupérer les événements d'une machine distante.

Pour lister les journaux d'événements (et non pas les événements eux-mêmes) d'une machine distante, nous devons passer par une requête WMI du genre :

```
PS > Get-WmiObject Win32_NTEventLogFile -computer 192.168.1.5
```

FileSize	LogfileName	Name	NumberOfRecords
-----	-----	----	-----
131072	Application	C:\WINDOWS\system32\config\AppEv...	394
65536	Directory Service	C:\WINDOWS\system32\config\NTDS...	251
65536	DNS Server	C:\WINDOWS\system32\config\DnsEven...	30
65536	File Replication Service	C:\WINDOWS\system32\config\NtFrs.E...	43
65536	Internet Explorer	C:\WINDOWS\System32\Config\Internet Explo...	0
48168960	Security	C:\WINDOWS\System32\config\SecEvent.Evt	104096
524288	System	C:\WINDOWS\system32\config\SysEvent.Evt	1470
393216	Windows PowerShell	C:\WINDOWS\System32\config\WindowsPowerSh...	475

➤ À ce stade, nous pourrions sans trop d'efforts sauvegarder ou effacer un journal. Par exemple, prenons celui situé à l'indice zéro de notre tableau (rappelez-vous, tous les tableaux commencent à l'indice à zéro), correspondant au journal Application.

```
PS > $journal = Get-WmiObject Win32_NTEventLogFile -computer 192.168.1.5
PS > $journal[0].BackupEventlog('c:\backup-Application.evt')
```

Attention, la sauvegarde s'effectue sur la machine distante !

Pour vider le journal, il suffit de remplacer la deuxième ligne de notre exemple par :

```
PS > $journal[0].ClearEventlog()
```

Sachez également qu'il existe de nombreuses autres méthodes possibles comme : changer les permissions, compresser, renommer, etc. Pour en obtenir la liste, comme d'habitude il faut jouer de la commande Get-Member.

Nous nous rendons compte que le journal d'événements Security est très volumineux : il fait environ 48 Mo. Imaginez le temps qu'il faudrait pour l'interroger si nous devons rapatrier en local tout le journal à la recherche d'un événement en particulier !

Pour limiter le temps de téléchargement des informations puis de traitement côté client, il est préférable d'effectuer une requête WQL pour restreindre le résultat. Celle-ci a l'avantage de s'exécuter sur la machine distante.

Par exemple, nous voudrions savoir quand ont eu lieu tous les arrêts d'un serveur distant. Pour ce faire, nous allons rechercher dans le journal de sécurité tous les événements dont le code est 513 :

```
PS > Get-WmiObject -query "select eventcode,sourcename,timewritten,
```

```
message from win32_NTLogEvent where logfile='Security' AND
EventCode='513'" -computer 192.168.1.5 | Format-List [a-z]*
```

```
EventCode    : 513
Message      : Windows s'arrête.
              Toutes les sessions vont être fermées par cet arrêt.

SourceName   : SECURITY
TimeWritten  : 20071019235518.000000+120

EventCode    : 513
Message      : Windows s'arrête.
              Toutes les sessions vont être fermées par cet arrêt.

SourceName   : SECURITY
TimeWritten  : 20071019234320.000000+120

...
```

Dans cet exemple, nous avons utilisé non plus la classe `Win32_NTEventLogFile` mais la classe `Win32_NTLogEvent`. En effet, cette dernière contient toutes les instances correspondant aux événements quels que soient leurs types. Nous avons donc appliqué un filtre où nous disons que nous ne voulons que les événements contenus dans le journal *Security* ET dont le code est *513*. Notez l'utilisation de **Format-List [a-z]*** où nous demandons l'affichage uniquement des propriétés dont la lettre commence par « a » jusqu'à « z » ; soit toute lettre de l'alphabet hors caractères spéciaux. Nous faisons cela car sinon nous aurions eu parmi les résultats des noms de propriétés internes à WMI commençant par « __ » comme « __CLASS » par exemple.

b. Dépouillement des données côté client

L'autre possibilité pour répondre à notre besoin pourrait être d'effectuer une partie du traitement de la requête par le client. Vous l'aurez compris, c'est une bien mauvaise idée dans la mesure où nous allons devoir passer en revue tous les événements. Et qui dit passer en revue, dit télécharger localement tous les événements. Soit, dans notre cas présent, environ 40 Mo de données !

Allez, on y croit et on essaye cette ligne de commandes :

```
PS > Get-WmiObject -query
      'SELECT eventcode,sourcename,timewritten,message,logfile
      FROM win32_NTLogEvent' -computer 192.168.1.5 |
      Where-Object {$_.logfile -eq 'Security' -and $_.eventcode -eq 513} |
      Format-List [a-z]*
```

Bien qu'en théorie cela doive fonctionner pour de petites quantités de données, il est néanmoins possible que vous rencontriez une erreur de type « violation de quota WMI ». Cela peut se produire lorsqu'il y a trop d'informations à récupérer. Cependant, le traitement sur le client peut être très utile pour les personnes pas très à l'aise avec le langage WQL mais qui maîtrisent PowerShell et en particulier la commandelette **Where-Object**. Mais pour cela, vous l'aurez compris, il faut que le volume d'informations à manipuler reste raisonnable afin de maximiser les performances.

8. Réglages de la sécurité WMI

Il y a deux aspects auxquels s'intéresser lorsque l'on parle de sécurité avec WMI. Le premier concerne les accès à distance à travers des pare-feu. Étant donné que WMI s'appuie sur la technologie COM (en particulier les fournisseurs), en environnement distribué c'est donc la technologie DCOM/RPC qui entre en jeu. Il va donc falloir être très vigilant sur la configuration des pare-feu de vos machines, car DCOM et RPC utilisent des ports qui sont généralement filtrés.

Le second volet relatif à la sécurité concerne le compte à partir duquel les requêtes WMI sont effectuées ; et que ces dernières s'appliquent localement ou sur des machines distantes.

Par défaut, `Get-WmiObject` s'exécute avec les droits de l'utilisateur connecté. Le plus souvent il faut être Administrateur pour que les requêtes se déroulent pour le mieux. La sécurité étant de nos jours de plus en plus présente, la preuve en est que depuis Windows Vista lorsque nous sommes connectés avec un compte Administrateur, toutes nos actions s'effectuent avec un privilège moindre ; celui d'un simple utilisateur, et il faut confirmer toute action s'effectuant avec des privilèges plus importants. Ce mécanisme s'appelle l'UAC (*User Account Control*).

`Get-WmiObject` sait s'affranchir de cette problématique avec le paramètre `-credential`. Grâce à ce dernier, il est possible de spécifier une identité alternative pour exécuter une requête WMI.

Exemple :

```
PS > $cred = Get-Credential          # une boîte de dialogue s'ouvre vous
                                     demandant de vous authentifier

PS > Get-WmiObject Win32_NTEventLogFile -computer 192.168.1.5 -cred $cred
```

Sans entrer dans les mécanismes internes et complexes de la sécurité liés à WMI, sachez également que depuis la version 2 de PowerShell il est possible de spécifier certains paramètres tels que le niveau d'emprunt d'identité (impersonation) et le niveau d'authentification (authentication) à utiliser pour les connexions WMI sur des machines distantes. Ces mécanismes de sécurité sont en réalité ceux des technologies COM et DCOM.

Il est cependant assez rare d'avoir besoin de modifier ces valeurs. En général nous nous contentons simplement de spécifier un couple login / mot de passe (credentials) administrateur de la machine distante avec le paramètre -credential.

Voici un tableau récapitulatif des différents paramètres de sécurité applicables au jeu de commandes WMI (Get-WmiObject, Set-WmiInstance, Invoke-WmiMethod, Remove-WmiInstance) qui peuvent, dans certains cas, vous être utiles :

Paramètre	Description
Authentication <AuthenticationLevel>	Spécifie le niveau d'authentification à utiliser avec la connexion WMI. Les valeurs valides sont : -1 : Unchanged 0 : Default 1 : None (aucune authentification n'est effectuée.) 2 : Connect (l'authentification est effectuée uniquement lorsque le client établit une relation avec l'application.) 3 : Call (l'authentification est effectuée uniquement au début de chaque appel, quand l'application reçoit une demande.) 4 : Packet (l'authentification est effectuée sur toutes les données reçues du client.) 5 : PacketIntegrity (toutes les données transférées entre le client et l'application sont authentifiées et vérifiées.) 6 : PacketPrivacy (les propriétés des autres niveaux d'authentification sont utilisées, et toutes les données sont chiffrées.)
Authority <String>	Spécifie l'autorité à utiliser pour authentifier la connexion WMI. Vous pouvez spécifier l'authentification Kerberos ou NTLM standard. Pour utiliser NTLM, affectez au paramètre d'autorité la valeur ntlm:domain:<Nom_Domaine>, où <Nom_Domaine> identifie un nom de domaine NTLM valide. Pour utiliser Kerberos, spécifiez « kerberos:<Nom_Domaine>\<Nom_Serveur> ». À noter que vous ne pouvez pas inclure le paramètre d'autorité lorsque vous vous connectez à l'ordinateur local.
Credential <PSCredential>	Spécifie un compte d'utilisateur qui a l'autorisation d'exécuter cette action. La valeur par défaut est l'utilisateur actuel. Tapez un nom d'utilisateur, tel que « User01 », « Domain01\User01 » ou « User@Contoso.com ». Vous pouvez également entrer un objet PSCredential, tel qu'un objet qui est retourné par l'applet de commande Get-Credential. Lorsque vous tapez un nom d'utilisateur, vous êtes invité à entrer un mot de passe.
EnableAllPrivileges [<SwitchParameter>]	Active tous les privilèges de l'utilisateur actuel avant que la commande ne passe l'appel WMI.
Impersonation <ImpersonationLevel>	Spécifie le niveau d'emprunt d'identité à utiliser. Les valeurs valides sont : 0 : Default (lit le Registre local pour connaître le niveau d'emprunt d'identité par défaut, qui a généralement la valeur « 3 : Impersonate ».) 1 : Anonymous (masque les informations d'identification de l'appelant.) 2 : Identify (permet aux objets d'interroger les informations d'identification de l'appelant.)

	3 : Impersonate (permet aux objets d'utiliser les informations d'identification de l'appelant.)
--	---

	4 : Delegate (permet aux objets d'autoriser d'autres objets à utiliser les informations d'identification de l'appelant.)
--	--

Monitoring de ressources avec la gestion des événements

Une autre facette de WMI assez méconnue mais pourtant très utile est la gestion des événements (ou *events* en anglais). WMI nous permet de surveiller ou de monitorer des événements en nous renvoyant une notification. Ensuite, libre à nous de décider quelle action entreprendre sur réception de tel ou tel événement.

La gestion des événements WMI peut se révéler être un formidable allié pour nous aider, nous administrateurs système, à éviter de nous transformer en de véritables pompiers. En effet, grâce à ce mécanisme, nous allons, par exemple, être prévenus en cas de remplissage à 80 % d'un disque logique d'une machine. « Être prévenu » peut signifier recevoir un e-mail, un SMS, ou un pop-up ; cela dépend uniquement de votre script et du mode de notification que vous avez choisi. Vous l'aurez compris, nous pouvons être notifiés de l'arrivée d'un événement sur toute ressource gérée par WMI. Nous pouvons donc surveiller le bon fonctionnement de certains services sur certains ordinateurs distants, surveiller l'exécution d'un processus particulier, ou bien encore monitorer certaines clés de registres. Vues toutes les classes WMI disponibles, il y a de grandes chances pour que vous puissiez monitorer LA ressource qui vous faisait défaut et qui vous permettra à l'avenir de dormir sur vos deux oreilles...

Pour ceux d'entre vous qui connaissent *System Center Operations Manager* (SCOM, ex MOM), et bien sachez que le principe des notifications sur événement est le même ; et pour ceux qui ne connaissent pas dites-vous que vous allez pouvoir faire de même que SCOM mais à une échelle infiniment plus petite...

Si les notifications WMI n'existaient pas, et pour tenter de faire de même, nous serions contraints de développer des scripts qui se déclencheraient à intervalles de temps réguliers pour surveiller certaines ressources gérées. Bien que cela soit possible, cette technique peut s'avérer très consommatrice en ressources car le script doit s'exécuter très souvent. Les notifications WMI ont justement été pensées pour être le moyen le plus efficace de réaliser ces tâches.

1. Surveiller la création d'un processus

Prenons un exemple simple : la surveillance du processus `MSPaint.exe` correspondant à l'application de dessin standard de Windows que tout le monde connaît. L'objectif est de déclencher un événement lorsque le système détecte le lancement du processus `MSPaint.exe`. L'événement sera simplement l'affichage d'un message.

Dans la première édition de ce présent ouvrage, avec la version 1 de PowerShell nous écrivions le script suivant :

```
# Monitoring du processus mspaint.exe

$strComputer = '.'
$query = "SELECT * FROM __InstanceCreationEvent
        WITHIN 3
        WHERE TargetInstance ISA 'Win32_process'
        AND TargetInstance.Name='mspaint.exe'"

$query = New-Object System.Management.WqlEventQuery $query

$scope = New-Object `
System.Management.ManagementScope "\\$strComputer\root\cimv2"

$watcher = New-Object `
System.Management.ManagementEventWatcher $scope,$query

$watcher.Start()
$event=$watcher.WaitForNextEvent()
Write-host "Une instance de MSPaint vient d'être créée."
```

Lorsque vous lancerez ce script, vous constaterez que la console PowerShell se fige. Cela est ainsi tant que le processus `MSPaint.exe` est attendu. L'exécution du script est en quelque sorte suspendue au niveau de l'appel de la méthode `WaitForNextEvent`. Dès lors que le processus `MSPaint.exe` est lancé et détecté par WMI, le script continue son exécution. La commande `Write-Host` affiche son message, puis le script se termine.

Nous n'utilisons pas `Get-WmiObject` dans cet exemple, pour la simple et bonne raison que cette commande ne prend pas en charge les événements WMI. Nous faisons donc directement appel aux classes .NET disponibles dans l'espace de noms `System.Management`. Nous créons deux objets qui sont : une requête WQL, et une étendue qui indique l'arborescence WMI dans laquelle se trouve l'instance à monitorer. À partir de ces deux objets, nous en créons un troisième qui sera un « observateur ». C'est lui qui recevra l'événement lorsqu'il sera détecté.

Mais attardons-nous quelques instants sur la requête WQL car c'est elle la plus importante dans cet exemple ; et ce sera seulement elle que nous modifierons à l'avenir pour les autres exemples.

Le début de la requête est semblable à celles que nous avons déjà réalisées précédemment, à savoir qu'elle est constituée de **SELECT** et de **FROM**. Par contre, nous indiquons ici que nous voulons obtenir des événements de type

__InstanceCreationEvent ; à savoir, comme le nom le laisse supposer, des événements de création d'objets. Ensuite, apparaissent deux nouveaux mots clés : **WITHIN** et **ISA**. Le premier indique un intervalle en secondes qui détermine la fréquence d'exécution du gestionnaire d'événements, et le second indique que l'instance à monitorer doit appartenir à une certaine classe WMI. Enfin on définit le nom de l'instance sur laquelle porte notre attention avec **TargetInstance.Name='processus'** . Si nous n'avions pas précisé un nom de processus (« MSPaint.exe »), le script nous aurait retourné la première instance de processus détectée.



Tous les scripts donnés en exemple peuvent sans problème s'exécuter sur une machine distante (dans la mesure où vous avez les privilèges adéquats) simplement en remplaçant le contenu de la variable **strComputer** par un nom d'ordinateur ou une adresse IP.

Bien que cet exemple soit parfaitement fonctionnel avec PowerShell v2, nous pouvons cependant lui apporter quelques modifications afin de tirer parti des nouvelles possibilités apportées par cette version.

PowerShell v2 dispose de la commande **Register-WmiEvent**. Celle-ci permet de s'abonner à des événements WMI ; ces derniers étant d'ailleurs dans la terminologie WMI appelés des « abonnés ».

Register-WmiEvent possède les paramètres suivants :

Paramètre	Description
Action <ScriptBlock>	Spécifie les commandes qui gèrent les événements. Les commandes spécifiées dans le paramètre Action s'exécutent quand un événement est déclenché, au lieu d'envoyer l'événement à la file d'attente d'événements. Placez les commandes entre accolades ({ }) pour créer un bloc de script. La valeur du paramètre Action peut inclure les variables automatiques \$Event, \$EventSubscriber, \$Sender, \$SourceEventArgs et \$SourceArgs, qui fournissent des informations sur l'événement au bloc de script Action.
Class <String>	Spécifie l'événement auquel vous vous abonnez. Entrez la classe WMI qui génère les événements. Un paramètre Class ou Query est obligatoire dans chaque commande.
ComputerName <String>	Spécifie un ordinateur distant. La valeur par défaut est l'ordinateur local. Entrez un nom NetBIOS, une adresse IP ou un nom de domaine complet.
Credential <PSCredential>	Spécifie un compte d'utilisateur qui a l'autorisation d'exécuter cette action. Tapez un nom d'utilisateur, tel que « User01 » ou « Domain01\User01 ». Vous pouvez également entrer un objet PSCredential, tel que celui généré par l'applet de commande Get-Credential . Si vous tapez un nom d'utilisateur, vous êtes invité à entrer un mot de passe.
Forward [<SwitchParameter>]	Envoie les événements pour cet abonnement à la session sur l'ordinateur local. Utilisez ce paramètre lorsque vous vous inscrivez aux événements sur un ordinateur distant ou dans une session à distance.
MessageData <PSObject>	Spécifie toutes les données supplémentaires à associer à cet abonnement aux événements. La valeur de ce paramètre apparaît dans la propriété MessageData de tous les événements associés à cet abonnement.
Namespace <String>	Spécifie l'espace de noms de la classe WMI.
Query <String>	Spécifie une requête dans le Langage de requêtes WMI (WQL) qui identifie la classe d'événements WMI (« select * from __InstanceDeletionEvent », par exemple).
SourceIdentifier <String>	Spécifie un nom que vous sélectionnez pour l'abonnement. Le nom que vous sélectionnez doit être unique dans la session active. La valeur par défaut est le GUID affecté par Windows PowerShell. La valeur de ce paramètre apparaît dans la valeur de la propriété SourceIdentifier de l'objet abonné et de tous les objets événements associés à cet abonnement.
SupportEvent [<SwitchParameter>]	Masque l'abonnement aux événements. Utilisez ce paramètre lorsque l'abonnement actuel fait partie d'un mécanisme d'inscription d'événement plus complexe et qu'il ne doit pas être découvert indépendamment.

	Pour afficher ou annuler un abonnement qui a été créé avec le paramètre <code>SupportEvent</code> , utilisez le paramètre Force des applets de commande <code>Get-EventSubscriber</code> et <code>Unregister-Event</code> .
Timeout <Int64>	Détermine le délai d'attente de Windows PowerShell jusqu'à ce que cette commande soit exécutée. La valeur par défaut, 0 (zéro), indique qu'aucun délai d'attente n'est défini et elle fait attendre Windows PowerShell indéfiniment.

À présent que nous en savons davantage sur cette commandelette, voyons comment nous pouvons réécrire notre précédent script :

```
#Requires -Version 2
# Monitoring du processus MSPaint.exe - v2

$query = "SELECT * FROM __InstanceCreationEvent
        WITHIN 3
        WHERE TargetInstance ISA 'Win32_process'
        AND TargetInstance.Name='mspaint.exe'"

$action = {Write-Host "
Une instance de MSPaint vient d'être créée à $($event.timegenerated)."}

Register-WmiEvent -query $query -SourceId 'Paint' -Action $action
```

Ce qui saute immédiatement aux yeux par rapport à la version précédente de notre script est son amaigrissement ! Mais si l'on sort de ces considérations purement esthétiques nous pouvons remarquer les choses suivantes :

- La requête WMI est toujours la même.
- Register-WmiEvent a considérablement donné plus de clarté au script ; ce qui facilite d'autant sa compréhension.
- L'action à effectuer lors de la survenue de l'événement est clairement identifiée par un bloc de script.
- Nous avons utilisé la variable `$event`. Celle-ci contient la référence de l'objet correspondant à l'événement qui a été déclenché. C'est ainsi que nous récupérerons la date et l'heure de déclenchement.

Lançons le script afin de voir comment ce dernier fonctionne :

```
PS > ./monitoring_PaintProcess.ps1

Id      Name      State      HasMoreData  Location      Command
--      -
8       Paint     NotStarted False         Write-Host "..."
```

Le script rend la main immédiatement contrairement à la version précédente. Cela vient du fait que dans PowerShell v2 la gestion des événements WMI est prise en charge par le mécanisme des jobs en arrière-plan (cf. Chapitre Maîtrise du Shell). Le résultat d'exécution de la commande `Get-Job` aurait été le même que ci-dessus, sauf qu'elle nous aurait en plus affiché tous les jobs en cours d'exécution.

Observez bien le champ « *State* ». On peut remarquer que notre événement nommé « *Paint* » est dans l'état « *NotStarted* », autrement dit il n'a pas encore été déclenché.

Si maintenant nous démarrons l'application MS Paint, voilà ce que la console PowerShell nous renvoie :

Une instance de MSPaint vient d'être créée à 10/22/2009 23:28:34.

À présent que nous avons déclenché l'événement, regardons de nouveau l'état de ce dernier avec la commande `Get-Job` :

```
PS > Get-Job

Id      Name      State      HasMoreData  Location      Command
--      -
8       Paint     Running    True         Write-Host "..."
```

Nous pouvons constater que l'état est maintenant passé à l'état « *Running* » et que la propriété `HasMoreData` est passée de la valeur « *False* » à « *True* ».

Notez que nous pouvons également utiliser la commande `Get-EventSubscriber` pour avoir des informations complémentaires sur notre gestionnaire d'événement WMI :

```
PS > Get-EventSubscriber

SubscriptionId      : 8
SourceObject        : System.Management.ManagementEventWatcher
EventName           : EventArrived
SourceIdentifier    : Paint
Action              : System.Management.Automation.PSEventJob
HandlerDelegate     :
SupportEvent        : False
ForwardEvent        : False
```

Si vous souhaitez désenregistrer le gestionnaire d'événement afin, par exemple, de libérer de la mémoire nous pouvons faire ceci :

```
PS > Get-EventSubscriber -SubscriptionId 8 | Unregister-Event
```

Ou

```
PS > Get-Job -id 8 | Remove-Job -force
```

Le paramètre **-force** est nécessaire afin de supprimer des jobs qui n'ont pas encore été exécutés.

Notez que si l'on ne précise pas les paramètres **-SubscriptionId** et **-id**, tous les jobs seront supprimés.

2. Surveiller le taux d'occupation disque d'un serveur

Comme nous vous le disions au début de cette partie, nous pouvons surveiller l'espace disque d'une machine locale ou distante sur le réseau. Avec WMI, lorsque l'on spécifie l'étendue de la requête, il suffit d'indiquer le nom de la machine distante en lieu et place de la valeur « . ». Dans les scripts basés sur WMI une bonne pratique est d'utiliser une variable pour indiquer le nom de la machine sur laquelle le script s'applique. Dans nos scripts, nous utilisons pour ce faire, la variable `$StrComputer`.

Dans cet exemple, nous allons déclencher une notification dès que le pourcentage d'espace libre du disque C : est inférieur à 10 Go. Nous allons baser notre requête sur la classe **Win32_LogicalDisk**. Nous nous contenterons d'afficher un message à l'écran indiquant que l'on a atteint un seuil critique et afficherons l'espace disque restant. Notez que cela n'est pas ce que l'on peut faire de mieux « dans la vraie vie » mais bien évidemment libre à vous d'écrire un script plus évolué. Comme PowerShell v2 apporte des facilités pour l'envoi de mails, nous modifierons la seconde version de notre script afin de notifier l'administrateur système par l'envoi d'un e-mail.

Avec la version 1 de PowerShell nous pouvons construire le script de la façon suivante :

```
# Surveillance de l'espace disque restant sur C:

$StrComputer = '.'
$query = "SELECT * FROM __InstanceModificationEvent
        WITHIN 60
        WHERE TargetInstance ISA 'Win32_LogicalDisk'
        AND TargetInstance.DeviceID = `\"C:\""
        AND TargetInstance.FreeSpace < 10737418240"

$query = New-Object System.Management.WqlEventQuery $query

$scope = New-Object System.Management.ManagementScope `
"\\$StrComputer\root\cimv2"
$watcher = New-Object System.Management.ManagementEventWatcher `
$scope, $query
$watcher.Start()

While ($true)
{
    $w=$watcher.WaitForNextEvent()
```

```

$freeSpace = $w.TargetInstance.FreeSpace/1GB
$freeSpace = [System.Math]::Round($freeSpace,2)
Write-Host "Seuil critique atteint !"
    Taille restante : $FreeSpace Go"
}

```

Le résultat d'exécution de ce script pourrait être celui-ci :

```

Seuil critique atteint !
    Taille restante : 8.63 Go

```

Vous remarquerez que cette fois, comme nous recherchons les événements de type modification, nous avons remplacé **__InstanceCreationEvent** par **__InstanceModificationEvent**. Nous avons modifié l'intervalle de monitoring en considérant qu'il n'était pas utile de solliciter la machine toutes les trois secondes, mais que soixante suffisaient.

Les plus attentifs d'entre vous auront également noté une petite subtilité dans l'affectation de la propriété **TargetInstance.DeviceID** au niveau de la requête WQL. En effet, nous avons dû employer le caractère d'échappement « backtick » (`) devant les guillemets. Cela est ainsi car la requête doit obligatoirement contenir des guillemets, et si nous ne mettons pas les backticks, PowerShell considérera que nous refermons la chaîne de caractères précédemment ouverte. Cela provoquerait donc invariablement une erreur. Nous avons également utilisé le backtick dans les lignes qui suivent afin de « casser » des lignes de script trop longues pour rendre notre script plus compréhensible après des lecteurs.

Nous avons aussi utilisé une méthode statique du Framework .NET, celle de la classe mathématiques (Math) nommée Round. Ainsi nous arrondissons le résultat de la conversion Octets -> GigaOctets à deux décimales.

Enfin, grâce à l'instruction **While** nous créons une boucle infinie afin que le script continue indéfiniment son exécution une fois la notification reçue.

Cet exemple, bien que parfaitement fonctionnel avec PowerShell v2, peut cependant être amélioré afin de tirer parti des nouvelles possibilités apportées par cette version.

Voici la seconde version de notre script :

```

#Requires -Version 2
# Surveillance de l'espace disque restant sur C:

$query = "SELECT * FROM __InstanceModificationEvent
        WITHIN 60
        WHERE TargetInstance ISA 'Win32_LogicalDisk'
        AND TargetInstance.DeviceID = `\"C:\""
        AND TargetInstance.FreeSpace < 10737418240"

$action = {
    $e = $Event.SourceEventArgs.NewEvent
    $freeSpace = $e.TargetInstance.FreeSpace/1GB
    $freeSpace = [System.Math]::Round($freeSpace,2)
    Write-Host "Seuil critique atteint !"
        Taille restante : $FreeSpace Go"
}

Register-WmiEvent -query $query -sourceid 'EspaceLibre' -action $action

```

Voilà le résultat après le lancement :

Id	Name	State	HasMoreData	Location	Command
--	----	-----	-----	-----	-----
5	EspaceLibre	NotStarted	False		...

Chose promise, chose due. Nous allons envoyer un e-mail à la place d'afficher une chaîne de caractères, ce qui sera plus représentatif de la réalité...

PowerShell v2 nous apporte la commandelette Send-MailMessage, mais nous ne détaillerons pas ici son fonctionnement. Voici ce à quoi pourrait ressembler notre script modifié :

```

#Requires -Version 2
# Surveillance de l'espace disque restant sur C:

$query = "SELECT * FROM __InstanceModificationEvent
        WITHIN 60
        WHERE TargetInstance ISA 'Win32_LogicalDisk'
        AND TargetInstance.DeviceID = `\"C:\""

```

```

        AND TargetInstance.FreeSpace < 10737418240"

$action = {
    $e = $Event.SourceEventArgs.NewEvent
    $freeSpace = $e.TargetInstance.FreeSpace/1GB
    $freeSpace = [System.Math]::Round($freeSpace,2)
    $message = "Seuil critique atteint ! Taille restante : $FreeSpace Go"

    Send-MailMessage -to 'admin@masociete.fr' -from 'robot@masociete.fr' `
        -subject 'Espace disque faible' -body $message -smtpServer `
        mailsrv.masociete.fr
    }

Register-WmiEvent -query $query -sourceid 'EspaceLibre' -action $action

```

3. Monitorer la suppression de fichiers

Après avoir illustré la surveillance de création et de modification d'instances, il ne nous restait plus qu'à tester la suppression d'instances. Dans cet exemple, nous surveillerons le répertoire « c:\temp » dans le but de détecter toute suppression de fichiers à l'intérieur de ce dernier.

```

# Surveillance de la suppression de fichiers dans C:\temp - v1

$strComputer = '.'
$query = "SELECT * FROM __InstanceDeletionEvent
    WITHIN 3
    WHERE TargetInstance ISA 'CIM_DirectoryContainsFile'
    AND TargetInstance.GroupComponent='Win32_Directory.Name=`C:\\\\temp`'"

$query = New-Object System.Management.WqlEventQuery $query

$scope = New-Object `
    System.Management.ManagementScope "\\$strComputer\root\cimv2"

$watcher = New-Object `
    System.Management.ManagementEventWatcher $scope,$query
$watcher.Start()

$file=$watcher.WaitForNextEvent()
Write-Host "Fichier $($file.TargetInstance.PartComponent) supprimé !"

```

Le résultat de l'exécution de ce script pourrait donner ceci :

```

Fichier \\WIN7_BUREAU\root\cimv2:CIM_DataFile.Name="C:\\temp\\test.txt"
supprimé !

```

Ce chemin correspond au chemin du fichier au format WMI ; à nous ensuite d'extraire le nom de notre fichier.

Avec PowerShell v2, nous pouvons transformer ce script ainsi :

```

#Requires -Version 2
# Surveillance de la suppression de fichiers dans C:\temp - v2

$query = "SELECT * FROM __InstanceDeletionEvent
    WITHIN 3
    WHERE TargetInstance ISA 'CIM_DirectoryContainsFile'
    AND TargetInstance.GroupComponent='Win32_Directory.Name=`C:\\\\temp`'"

$action =
{
    $e = $Event.SourceEventArgs.NewEvent
    Write-Host "Le fichier $($e.TargetInstance.PartComponent) a été supprimé !"
}

Register-WmiEvent -query $query -sourceid 'suppression' -action $action

```

4. Quelques explications complémentaires

Vous aurez pu remarquer au travers de ces quelques exemples que seule la requête WQL change ; le reste du script est quasiment identique. Une bonne maîtrise de la gestion des événements WMI passe donc forcément par une bonne compréhension du langage WQL et surtout du schéma de la base WMI.

Nous avons utilisé les trois classes d'événements suivantes :

- __InstanceCreationEvent
- __InstanceModificationEvent
- __InstanceDeletionEvent

Ces classes nous ont permis de monitorer des instances, ou autrement dit, des objets de notre système d'exploitation. Sachez cependant qu'il existe d'autres classes pour monitorer des opérations sur des classes et sur les espaces de noms WMI mais celles-ci n'ont que peu d'intérêt pour un public d'administrateurs système. Enfin une autre catégorie de classes d'événements susceptible de nous intéresser est celle qui permet le monitoring de la base de registres ; ceci étant nous n'allons pas nous y attarder dans la mesure où le principe reste toujours le même.

Éviter de sortir sauvagement de la console dans l'attente d'événements

Lorsque nous faisons une boucle infinie avec **While (\$true) { ... }** comme dans l'exemple où nous surveillons le taux d'occupation disque, le seul moyen d'interrompre le script est de presser [Ctrl] [Pause] car [Ctrl] **C** n'a aucun effet. Bien que cette séquence de touches soit efficace, elle l'est même un peu trop, car elle ferme aussi la console PowerShell.

Le problème ne vient pas de l'instruction **while**, mais de l'observateur WMI (classe `ManagementEventWatcher`) qui attend une notification. En effet, celui-ci est insensible au [Ctrl] **C**. Il existe néanmoins une petite astuce pour contourner ce problème. Celle-ci va consister à définir un timeout pour notre observateur WMI. C'est-à-dire qu'au bout d'un certain temps d'attente, si une notification n'est toujours pas reçue, l'observateur cessera son travail et redonnera la main au script. Le script ne sera donc plus figé et l'exécution pourra continuer son cours normal. C'est donc à ce moment là, que nous pourrons effectuer un [Ctrl] **C** pour quitter le script, juste avant qu'il ne se remette à attendre une autre notification. Cependant, pour que cela fonctionne correctement, il va nous falloir effectuer un « **trap** » d'erreur car l'observateur, à l'issue du timeout, émet une exception. Et si nous « n'attrapons » pas cette exception, le script s'interrompt car il a affaire à une erreur critique (si besoin reportez-vous au chapitre Maîtrise du Shell sur la gestion des erreurs). Nous définirons donc un gestionnaire d'interception, qui, lorsqu'il attrapera une exception de type **System.Management.ManagementException** fera en sorte que le script continue normalement son exécution. Nous ferons également un test afin de déterminer si un événement s'est produit ou non. Et si tel est le cas, alors nous afficherons un message.

Revoici notre second exemple revu et corrigé :

```
# Surveillance de l'espace disque restant sur C: - v1
# avec possibilité de quitter avec [CTRL]+C

$StrComputer = '.'
$query = "SELECT * FROM __InstanceModificationEvent
        WITHIN 3
        WHERE TargetInstance ISA 'Win32_LogicalDisk'
        AND TargetInstance.DeviceID = `\"C:\""
        AND TargetInstance.FreeSpace < 10737418240"
$query = New-Object System.Management.WqlEventQuery $query

$scope = New-Object System.Management.ManagementScope `
"\\$StrComputer\root\cimv2"

$watcher = New-Object `
System.Management.ManagementEventWatcher $scope,$query
$options = New-Object System.Management.EventWatcherOptions
$options.Timeout = [timespan]"0.0:0:1" # Timeout d'1 seconde
$watcher.Options = $options
$watcher.Start()

While ($true){
    trap [System.Management.ManagementException] {continue}
    $w=$watcher.WaitForNextEvent()
```

```
if ($w.TargetInstance -ne $null) {  
    $freeSpace = $w.TargetInstance.FreeSpace/1GB  
    $freeSpace = [System.Math]::Round($freeSpace,2)  
  
    Write-Host "Seuil critique atteint !  
        Taille restante : $freeSpace Go"  
    $w = $null  
}  
}
```

En utilisant cette technique, nous pouvons dire à présent que nous faisons de la gestion des événements semi-synchrone.

➤ Si nous voulons éviter l’affichage sur la console du message d’erreur provoqué par l’exception (même si le script continue son exécution), nous devons donner la valeur **SilentlyContinue** à la variable de préférence **\$ErrorActionPreference**. Si nous ne le faisons pas, voici le message d’erreur que nous obtiendrons : Exception lors de l’appel de « WaitForNextEvent » avec « 0 » argument(s) : « Délai dépassé ». Bien entendu, avec la version 2 de PowerShell tout ce travail devient superflu car, de base, la gestion des événements si vous la faites avec Register-WmiEvent est asynchrone.

➤ Remarque à l’attention des utilisateurs de PowerShell v1 : la gestion des événements asynchrones, contrairement aux événements synchrones, n’est pas censée bloquer l’exécution d’un script. En effet dans ce contexte, les événements asynchrones devraient en théorie être détectés en tâche de fond. Leur gestion avec PowerShell relève de la programmation avancée plutôt que de scripting. C’est la raison pour laquelle le projet open-source « PowerShell Eventing » a été créé sur le site communautaire CodePlex.com, et a fédéré un petit groupe de développeurs durant environ cinq mois. C’est ainsi qu’a vu le jour un petit jeu de commandes dédié à la gestion des événements (synchrones et asynchrones). Pour information, ces commandelettes aux noms évocateurs sont : **New-Event**, **Get-Event**, **Get-EventBinding**, **Connect-EventListener**, **Disconnect-EventListener** et **Start-KeyHandler**.

➤ Téléchargez le snap-in PowerShell Eventing à l’adresse suivante : <http://www.codeplex.com/PSEventing>

Introduction

Cette partie est spécifique à PowerShell version 2 et ultérieures.

Nous avons vu à travers WMI qu'il était possible d'exécuter des requêtes afin de gérer des ordinateurs locaux ou bien distants. Néanmoins, vous découvrirez dans ce chapitre qu'il existe d'autres moyens de gérer des ordinateurs distants. Nous verrons notamment comment y parvenir, tout en utilisant exclusivement PowerShell.

Communications à distance du Framework .NET 2.0

Vous le savez, PowerShell s'appuie pleinement sur le Framework .NET 2.0 et à ce titre il bénéficie des fonctionnalités d'exécution à distance de ce dernier. C'est ainsi que quelques commandelettes ont hérité du paramètre **-ComputerName**. Certaines de ces heureuses élues permettent de s'exécuter sur un ou plusieurs ordinateurs distants sans même que PowerShell n'ait besoin d'être installé sur ces derniers.

Les communications à distance du Framework .NET représentent la manière la plus simple d'agir sur des machines distantes, mais attention elles s'appuient sur le protocole RPC. Ce protocole étant la plupart du temps filtré par les routeurs, n'est par conséquent utilisable que sur des réseaux de type LAN.

1. Pré-requis

- Être membre du groupe Administrateurs de l'ordinateur distant ou être membre du groupe Administrateurs du domaine,
- Disposer de PowerShell v2 sur votre ordinateur et uniquement sur le vôtre.

2. Déterminer les commandes à distance du Framework .NET 2.0

Pour connaître toutes les commandelettes pourvues du paramètre **-ComputerName**, tapez :

```
PS > Get-Help * -parameter ComputerName
```

Ensuite pour s'assurer que la commandelette s'appuie sur les mécanismes du Framework .NET, il faut en fait vérifier, via l'aide en ligne, qu'elle ne s'appuie pas sur « la communication à distance Windows PowerShell » (nous verrons ce que c'est dans la partie suivante de ce chapitre).

Exemple :

```
PS > Get-Help Get-Process -parameter ComputerName

-ComputerName <String[]>
    Obtient les processus qui s'exécutent sur les ordinateurs spécifiés.
La valeur par défaut est l'ordinateur local.

    Tapez le nom NetBIOS, une adresse IP ou un nom de domaine complet
d'un ou de plusieurs ordinateurs. Pour spécifier l'ordinateur local,
tapez le nom de l'ordinateur, un point (.) ou « localhost ».

    Ce paramètre ne s'appuie pas sur la communication à distance Windows
PowerShell. Vous pouvez utiliser le paramètre ComputerName de
Get-Process même si votre ordinateur n'est pas configuré pour exécuter
des commandes distantes.
```

Nous pouvons lire dans l'aide de cette commande que celle-ci « ne s'appuie pas sur la communication à distance Windows PowerShell », cela signifie donc qu'elle s'appuie sur les mécanismes de communication à distance du Framework .NET 2.0.

Cette démarche peut sembler un peu particulière pour déterminer quelles sont les commandes qui s'appuient les mécanismes de communication à distance du Framework .NET 2.0 de celles qui s'appuient sur les mécanismes de communication à distance PowerShell ; mais malheureusement il n'y a pas d'autres solutions.

L'idée de la « Team PowerShell » est sans doute que l'utilisateur final ne se pose pas toutes ces questions. En effet, la finalité est de proposer le même paramètre à différentes commandes, peu importe la technologie qui se cache derrière. Ceci étant, connaître la technologie sous-jacente peut avoir son importance car il va se passer encore des années avant que toutes les entreprises aient fini le déploiement de PowerShell v2 sur tout leur parc de machines ! Non seulement PowerShell v2 doit être installé, mais il doit aussi être configuré pour accepter les communications à distance PowerShell. C'est pourquoi il nous semble important de mettre en avant et de faire connaître les commandelettes qui peuvent s'employer à distance avec un minimum de pré-requis (voir plus haut).

Comme il est fastidieux de consulter l'aide de chaque commande pour vérifier la présence de cette chaîne de caractères, un petit script s'impose :

```
# Find-DotNetRemoteCmdlets.ps1
# Liste les commandelettes qui ne s'appuient pas sur les fonctionnalités
# de communication à distance PowerShell v2

$trouve = @()
$pasTrouve = @()
$pattern = 'pas sur la communication à distance Windows PowerShell'
$liste = Get-Help * -parameter ComputerName

ForEach ($cmde in $liste)
{
    $description =
        (Get-Help $cmde.name -parameter ComputerName).Description | Out-String
    If ($description | Select-String -pattern $pattern)
    {
        $trouve += $cmde.name
    }
    Else
    {
        $pasTrouve += $cmde.name
    }
}

$trouve | Sort-Object
```

Le script commence par initialiser deux variables de type tableau (\$trouve et \$pasTrouve) dans lesquelles il stockera les résultats des recherches. La variable \$pattern contient la chaîne à rechercher et la variable \$liste contient la liste des commandelettes qui possède un paramètre nommé **ComputerName**. Puis pour chaque commande de la liste \$liste, on itère. L'itération consiste à rappeler la commande Get-Help de chaque commande afin de rechercher au moyen de Select-String la chaîne caractéristique contenue dans \$pattern. Notez l'utilisation de Out-String afin de convertir l'aide en une chaîne de caractères nécessaire à l'utilisation de Select-String. Enfin si la recherche aboutit, on stocke le nom de la commande dans la variable \$trouve, dans le cas contraire on la stocke dans \$pasTrouve. Puis on retourne le résultat de la variable \$trouve sous forme triée par ordre alphabétique.

Ce qui donne le résultat suivant :

```
Clear-EventLog
Get-Counter
Get-EventLog
Get-HotFix
Get-Process
Get-Service
Get-WinEvent
Get-WmiObject
Limit-EventLog
New-EventLog
Remove-EventLog
Remove-WmiObject
Restart-Computer
Set-Service
Set-WmiInstance
Show-EventLog
Stop-Computer
Test-Connection
Write-EventLog
```

3. Le jeu de commandes

Voyons un peu plus en détail ce qu'il est possible de faire avec chacune de ces commandes. Nous les avons regroupées par thème :

Commande	Description
Clear-EventLog	Supprime toutes les entrées des journaux des événements spécifiés.

Get-EventLog	Obtient les événements d'un journal des événements ou la liste des journaux des événements.
Limit-EventLog	Définit les propriétés de journal des événements qui limitent la taille du journal des événements et l'ancienneté de ses entrées.
Remove-EventLog	Supprime un journal des événements ou annule l'inscription d'une source d'événement.
Show-EventLog	Affiche les journaux des événements.
New-EventLog	Crée un journal des événements et une source d'événement.
Write-EventLog	Écrit un événement dans un journal des événements.
Get-WinEvent	Obtient des événements à partir des journaux des événements et des fichiers journaux de suivi d'événements.
Get-Service	Obtient la liste des services.
Set-Service	Démarre, arrête et interrompt un service, puis modifie ses propriétés.
Get-HotFix	Obtient les correctifs logiciels du système qui ont été installés.
Set-WmiInstance	Crée ou met à jour une instance d'une classe WMI existante.
Get-WmiObject	Obtient des instances de classes WMI ou des informations sur les classes disponibles.
Remove-WmiObject	Supprime une instance d'une classe WMI existante.
Get-Process	Obtient la liste des processus en cours d'exécution.
Restart-Computer	Redémarre le système d'exploitation.
Stop-Computer	Arrêt du système d'exploitation.
Test-Connection	Envoie les paquets de demande d'écho ICMP (« pings »).
Get-Counter	Obtient des données de compteur de performance.

4. Envoi de commandes à distance

L'envoi d'une commande de la liste du tableau précédent est on ne peut plus simple car peu importe si PowerShell est installé ou non sur les machines distantes. Le seul pré-requis nécessaire est d'être connecté avec un compte qui soit au minimum administrateur de la machine distante. En effet, toutes ces commandelettes ne permettent pas le passage d'autorisations alternatives.

Exemple 1 : arrêt/redémarrage du service W32Time d'un serveur à distance

Arrêt du service :

```
PS > Get-Service -ComputerName W2K8R2VM -Name W32time | Set-Service
-Status stopped
```

Vérification de l'état du service :

```
PS > Get-Service -ComputerName W2K8R2VM -name W32time

Status  Name                DisplayName
```

```
-----  ----  -----
Stopped W32time      Windows Time
```

Démarrage du service :

```
PS > Get-Service -ComputerName W2K8R2VM -name W32time | Set-Service
-Status running
```

Exemple 2 : lire les journaux d'événements d'une machine distante

```
PS > Get-EventLog -ComputerName W2K8R2VM -LogName system -Newest 10
```

Cette ligne de commandes récupère les 10 entrées les plus récentes du journal système d'une machine distante.

Index	Time	EntryType	Source	InstanceID	Message
6109	oct. 27 23:19	Information	Service Control M...	1073748860	Le service Expéri...
6108	oct. 27 23:18	Information	Service Control M...	1073748860	Le service Servic...
6107	oct. 27 23:09	Information	Service Control M...	1073748860	Le service Protec...
6106	oct. 27 23:08	Information	Service Control M...	1073748860	Le service Servic...
6105	oct. 27 23:08	Information	Service Control M...	1073748860	Le service Planif...
6104	oct. 27 23:07	Information	Service Control M...	1073748860	Le service Servic...
6103	oct. 27 23:07	Information	Service Control M...	1073748860	Le service Découv...
6102	oct. 27 23:07	Information	Service Control M...	1073748860	Le service Servic...
6101	oct. 27 23:07	Information	Service Control M...	1073748860	Le service Expéri...
6100	oct. 27 23:07	Information	Service Control M...	1073748860	Le service Servic...

Et si l'on veut filtrer pour n'afficher que les 10 dernières erreurs, c'est aussi simple que cela :

```
PS > Get-EventLog -ComputerName W2K8R2VM -LogName system |
Where {$_.EntryType -eq 'Error'} | Select-Object -First 10
```

Index	Time	EntryType	Source	InstanceID	Message
1216	oct. 26 14:46	Error	DCOM	3221235481	La description de...
1215	oct. 26 14:46	Error	DCOM	3221235481	La description de...
1214	oct. 26 14:46	Error	DCOM	3221235481	La description de...
1213	oct. 26 14:44	Error	DCOM	3221235481	La description de...
1212	oct. 26 14:44	Error	DCOM	3221235481	La description de...
995	oct. 19 00:17	Error	DCOM	3221235482	La description de...
993	oct. 19 00:05	Error	DCOM	3221235478	La description de...
992	oct. 18 23:47	Error	NETLOGON	5805	The session setup...
991	oct. 18 23:40	Error	NETLOGON	5723	The session setup...
974	oct. 15 00:06	Error	UmrdpService	1111	Driver Send To Mi...

Communications à distance Windows PowerShell

Le mécanisme de communication à distance Windows PowerShell apporté dans la version 2 de PowerShell s'appuie quand à lui sur le protocole WS-MAN, appelé aussi « Gestion des services Web » ou encore « Gestion à distance de Windows (WinRM) ».

WinRM une technologie qui a émergé récemment, elle est arrivée avec Windows Server 2003 R2. WinRM, qui signifie Windows Remote Management, permet d'envoyer des requêtes de gestion à des machines sur le réseau via les ports 5985 (HTTP) /5986 (HTTPS). Ces derniers sont les nouveaux ports définis dans Windows 7 et Windows Server 2008 R2 comme port d'écoute concernant les communications WinRM. Il reste cependant possible d'utiliser les ports « standard » 80 (HTTP) et 443 (HTTPS) en prenant soin de les spécifier (voir ci après). Le protocole qui se cache derrière cette technologie s'appelle WS-Management ; il est basé sur SOAP (*Simple Object Access Protocol*).

À l'instar de WBEM, WS-Management (ou WSMAN) apporte un standard (schéma XML) orienté gestion matérielle pour faciliter l'interopérabilité entre des systèmes hétérogènes au sein d'une infrastructure IT. Très concrètement, WinRM établit une session avec un ordinateur distant à travers le protocole WS-Management au lieu de DCOM, comme le fait WMI en environnement distribué. Par conséquent, le résultat d'une requête WS-Management est un flux de données XML qui transite par des ports standard du Web.

La sécurité de WinRM repose sur des méthodes standard pour l'authentification et le chiffrement des échanges ; pour l'authentification Kerberos est le protocole utilisé par défaut à l'intérieur d'un domaine. Mais d'autres méthodes d'authentification peuvent être utilisées comme : NTLM, le mappage de certificat client, ou encore et toujours l'authentification basique avec login/mot de passe. Bien sûr celle-ci est fortement déconseillée dans un environnement de production, à moins de l'utiliser sur HTTPS ou d'utiliser IPSEC (*Internet Protocol SECURITY*).

1. Installation des pré-requis

Les pré-requis à la fois sur la machine locale et la ou les machines distante(s) sont les suivants :

- Windows PowerShell v2 ou ultérieur
- Framework .NET 2.0 ou ultérieur
- Windows Remote Management 2.0

Tous ces composants sont installés nativement dans Windows 7 et Windows Server 2008 R2. Ils font également partie du package d'installation de PowerShell v2 pour les versions antérieures de Windows.

Pour être en mesure d'exécuter des commandes à distance, vous devez être membre du groupe Administrateurs de l'ordinateur distant ou être capable de fournir des informations d'identification d'un administrateur.

Il est également nécessaire de démarrer la session PowerShell en tant qu'administrateur si vous souhaitez faire au moins l'une des tâches suivantes :

- Connexion locale au moyen d'une connexion à distance. Cette opération est appelée « bouclage »,
- Gestion des configurations de session sur l'ordinateur local,
- Affichage et modification des paramètres de Gestion des services Web sur l'ordinateur local au moyen du fournisseur WSMAN:.

Pour vérifier la version de PowerShell installée, utilisez la variable automatique \$PSVersionTable. Voir ci-après :

```
PS > $PSVersionTable

Name                           Value
----                           -
CLRVersion                     2.0.50727.4927
BuildVersion                   6.1.7600.16385
PSVersion                      2.0
WSManStackVersion              2.0
PSCompatibleVersions           {1.0, 2.0}
SerializationVersion           1.1.0.1
PSRemotingProtocolVersion      2.1
```

2. Configuration du système

Pour autoriser un système à recevoir des commandes à distance il est nécessaire d'utiliser la commandelette Enable-PSRemoting. Notez que pour envoyer des commandes, cette étape n'est pas nécessaire. Elle est même fortement déconseillée si elle n'est pas nécessaire car elle ouvre le système à d'éventuelles attaques.

Enable-PSRemoting effectue les opérations de configuration suivantes :

- Démarre le service WinRM s'il n'est pas démarré,
- Modifie le type de démarrage du service WinRM sur « automatique »,
- Crée un écouteur WinRM pour accepter les demandes sur toute adresse IP,
- Active une exception dans le pare-feu pour les communications WS-Man,
- Active toutes les configurations de session inscrites de façon à pouvoir recevoir des ordres d'un ordinateur distant,
- Inscrit la configuration de session « Microsoft.PowerShell » (nous verrons par la suite ce qu'est une configuration de session),
- Inscrit la configuration de session « Microsoft.PowerShell32 » si le système d'exploitation est 64 bits,
- Supprime l'autorisation « Refuser Tout le monde » du descripteur de sécurité pour toutes les configurations de session existantes,
- Redémarre le service WinRM pour appliquer les changements.



La configuration du service WinRM rend active la règle du pare-feu nommée « Windows Remote Management (HTTP-in) » en ouvrant le port TCP 5985.



Le nom de cette règle est « Gestion à distance de Windows » sur les versions Windows 7 et les versions serveurs antérieures à Windows Server 2008.

Si le système n'a jamais été configuré alors vous devriez obtenir quelque chose d'assez semblable :

```
PS > Enable-PSRemoting
```

Configuration rapide du service WinRM (Gestion à distance de Windows)

Exécution de la commande « Set-WSManQuickConfig » pour activer l'administration à distance de cet ordinateur via le service WinRM.

Cette administration inclut les opérations suivantes :

1. Démarrage ou redémarrage (s'il est déjà démarré) du service WinRM.
2. Affectation du démarrage automatique au service WinRM.
3. Création d'un écouteur pour accepter les demandes sur n'importe quelle adresse IP.
4. Activation de l'exception de pare-feu pour le trafic du service Gestion des services Web (pour HTTP uniquement).

Voulez-vous continuer ?

[O] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S] Suspendre

[?] Aide (la valeur par défaut est « 0 ») : 0

WinRM a été mis à jour pour recevoir des demandes.

Le type du service WinRM a été correctement modifié.

Le service WinRM a démarré.

WinRM a été mis à jour pour la gestion à distance.

Écouteur WinRM créé sur HTTP://* pour accepter les demandes de la gestion des services Web sur toutes les adresses IP de cet ordinateur.

Exception de pare-feu WinRM activée.

Confirmer

Êtes-vous sûr de vouloir effectuer cette action ?

Opération « Inscription de la configuration de la session » en cours sur la cible « La configuration de la session « Microsoft.PowerShell32 » est introuvable. Exécution de la commande « Register-PSSessionConfiguration Microsoft.PowerShell32 -processorarchitecture x86 -force » pour créer la configuration de la session « Microsoft.PowerShell32 ».

Cela redémarrera le service WinRM. ».

[O] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S] Suspendre

[?] Aide (la valeur par défaut est « 0 ») : 0

Pour vérifier la bonne configuration du système vous pouvez essayer de créer une session à l'aide de la commande New-PSSession, comme ci-après :

```
PS > New-PSSession
```

Id	Name	ComputerName	State	ConfigurationName	Availability
1	Session1	localhost	Opened	Microsoft.PowerShell	Available

Si vous obtenez le même résultat alors félicitations, votre système est à présent prêt à recevoir des commandes à distance !

3. Gestion des configurations des sessions à distance

Si vous avez observé avec attention le processus de configuration de la machine, vous avez pu constater qu'il y a eu la création d'une configuration de session avec l'aide de la commande Register-PSSessionConfiguration.

Une configuration de session est un groupe de paramètres sur l'ordinateur local qui définit l'environnement pour les sessions à distance créées lorsque les utilisateurs distants se connectent. Il est possible de créer des configurations de session plus ou moins restrictives en termes de sécurité selon les besoins du moment. Par défaut, seuls les membres du groupe Administrateurs ont l'autorisation de se connecter à distance. Il est par exemple possible dans une configuration de session de limiter la taille des objets que l'ordinateur reçoit dans la session, de définir le mode de langage et de spécifier les commandelettes, fournisseurs et fonctions qui sont disponibles dans la session.

Les configurations de session PowerShell sont utilisées uniquement lorsque vous vous connectez à distance avec les commandes New-PSSession, Enter-PSSession et Invoke-Command.

Les commandes de gestion des configurations sont les suivantes, elles s'appliquent toujours localement :

Commande	Description
Register-PSSessionConfiguration	Crée une configuration de session sur l'ordinateur local
Unregister-PSSessionConfiguration	Supprime une configuration de session
Get-PSSessionConfiguration	Obtient les configurations de session inscrites
Set-PSSessionConfiguration	Modifie les propriétés d'une configuration de session inscrite
Enable-PSSessionConfiguration	Active les configurations de session
Disable-PSSessionConfiguration	Refuse l'accès aux configurations de session
New-PSSessionOption	Crée un objet qui contient les options avancées d'une session PSSession

Configuration de session par défaut

Windows PowerShell inclut une configuration de session intégrée nommée Microsoft.PowerShell. Sur les ordinateurs qui exécutent des versions 64 bits de Windows (ce qui est notre cas), Windows PowerShell fournit également une deuxième configuration de session nommée Microsoft.PowerShell32.

Ces configurations de session sont utilisées par défaut pour les sessions, autrement dit lorsqu'une commande permettant de créer une session n'inclut pas le paramètre -ConfigurationName.

Il est également possible de modifier la configuration de session par défaut en utilisant la variable de préférence \$PSSessionConfigurationName.

Observons notre configuration de session par défaut :

```
PS > Get-PSSessionConfiguration Microsoft.PowerShell | Format-List

Name                : microsoft.powershell
Filename            : %windir%\system32\pwrshplugin.dll
SDKVersion          : 1
XmlRenderingType    : text
lang               : fr-FR
PSVersion           : 2.0
ResourceUri         : http://schemas.microsoft.com/powershell/
microsoft.powershell
SupportsOptions     : true
Capability          : {Shell}
xmlns               : http://schemas.microsoft.com/wbem/wsman/1/
config/PluginConf...
Uri                 : http://schemas.microsoft.com/powershell/
microsoft.powershell
ExactMatch          : true
SecurityDescriptorSddl : 0:NSG:BAD:P(A;;GA;;;BA)S:P(AU;FA;GA;;;WD)
(AU;SA;GXGW;;;WD)
Permission          : BUILTIN\Administrators AccessAllowed
```

Notez qu'il est également possible d'accéder à ces informations à l'aide du fournisseur WSMAN.

4. Créer une session à distance

Avant de pouvoir envoyer des ordres à un ordinateur distant il est nécessaire d'établir une session à distance.

Une session à distance peut être :

- **Temporaire** : une session temporaire est établie juste pour la durée de l'envoi d'une commande avec Invoke-Command ou Enter-PSSession. Tel est le cas lors de l'utilisation du paramètre -ComputerName.
- **Permanente** : une session « permanente » persiste durant le temps de la session PowerShell. Une session permanente est utile dans les cas où l'on doit exécuter plusieurs commandes qui partagent des données par l'intermédiaire de variables ou de fonctions. On crée une session permanente lorsque l'on utilise le paramètre -Session des commandes Invoke-Command ou Enter-PSSession.

La création d'une connexion permanente à un ordinateur local ou distant s'effectue avec la commande New-PSSession. Observons ses paramètres à l'aide du tableau suivant :

Paramètre	Description
AllowRedirection <Switch>	Autorise la redirection de cette connexion vers un autre URI (<i>Uniform Resource Identifier</i>).
ApplicationName <String>	Spécifie le segment du nom d'application dans l'URI de connexion.
Authentication <AuthenticationMechanism>	Spécifie le mécanisme permettant d'authentifier les informations d'identification de l'utilisateur.
CertificateThumbprint <String>	Spécifie le certificat de clé publique numérique (X509) d'un compte d'utilisateur qui a l'autorisation d'exécuter cette action.
ComputerName <String[]>	Crée une connexion permanente à l'ordinateur spécifié.
ConfigurationName <String>	Spécifie la configuration de session utilisée pour la nouvelle session PSSession.

ConnectionURI <Uri[]>	Spécifie un URI qui définit le point de terminaison de connexion.
Credential <PSCredential>	Spécifie un compte d'utilisateur qui a l'autorisation d'exécuter cette action.
Name <String[]>	Spécifie un nom convivial pour la session PSSession.
Port <Int>	Spécifie le port réseau sur l'ordinateur distant utilisé pour cette commande. La valeur par défaut est le port 80 (port HTTP).
Session <PSSession[]>	Utilise la session PSSession spécifiée en tant que modèle pour la nouvelle session PSSession.
SessionOption <PSSessionOption>	Définit des options avancées pour la session
ThrottleLimit <Int>	Spécifie le nombre maximal de connexions simultanées qui peuvent être établies pour exécuter cette commande.
UseSSL <Switch>	Utilise le protocole Secure Socket Layer sur HTTPS pour établir la connexion. Par défaut SSL n'est pas utilisé.

Voici quelques exemples d'utilisation de New-PSSession :

Exemple 1 :

```
PS > $session = New-PSSession
```

Création d'une session sur l'ordinateur local et stockage de la référence de l'objet PSSession dans la variable \$session.

Exemple 2 :

```
PS > $session = New-PSSession -ComputerName W2K8R2SRV
```

Création d'une session sur l'ordinateur distant « W2K8R2SRV » et stockage de la référence de l'objet PSSession dans la variable \$session.

Exemple 3 :

```
PS > $cred = Get-Credential
PS > $session = New-PSSession -ComputerName W2K8R2SRV -Credential $cred
```

Création d'une session sur l'ordinateur distant « W2K8R2SRV » en spécifiant des autorisations alternatives et stockage de la référence de l'objet PSSession dans la variable \$session.

Exemple 4 :

```
PS > $machines = Get-Content C:\temp\machines.txt
PS > $sessions = New-PSSession -ComputerName $machines -ThrottleLimit 50
```

Création de multiples sessions sur une liste d'ordinateurs distants en spécifiant une valeur de connexions maximales simultanées de 50.

5. Exécution de commandes à distance

Ça y est, entrons à présent dans le vif du sujet ! La commande nécessaire pour l'exécution de commandes à distance est Invoke-Command. Celle-ci possède de nombreux paramètres que nous verrons un peu plus tard.

Comme nous vous le disions dans la partie précédente, une session peut être temporaire ou permanente. On peut donc choisir l'une ou l'autre en fonction de son besoin du moment. Pour l'envoi ponctuel d'une commande à distance il peut être plus facile d'utiliser une session temporaire, tandis que si vous devez envoyer une succession de commandes, il sera plus commode d'établir une session permanente.

Pour illustrer ces propos, intéressons-nous à quelques cas d'usages :

Exemple 1 : récupération d'une variable d'environnement sur un ordinateur distant

```
PS > Invoke-Command -ComputerName W2K8R2SRV -ScriptBlock
{$env:PROCESSOR_IDENTIFIER}
Intel64 Family 6 Model 15 Stepping 11, GenuineIntel
```

Pas de complication particulière, on spécifie avec le paramètre **-ComputerName** le nom de la machine sur laquelle exécuter le bloc de script passé entre accolades au paramètre **-ScriptBlock**.

Exemple 2 : lister les services suspendus sur plusieurs ordinateurs

```
PS > $cmde = { Get-Service | Where {$_.Status -eq 'paused'} }
PS > Invoke-Command -ComputerName W2K8R2SRV, localhost -ScriptBlock $cmde
```

Status	Name	DisplayName	PSComputerName
----	----	-----	-----
Paused	vmacthlp	Hyper-V Time Synchronization Service	w2k8r2srv
Paused	Winmgmt	Infrastructure de gestion Windows	localhost

Voyez avec quelle étonnante facilité nous avons pu faire exécuter un petit bloc de script sur une liste d'ordinateurs. Remarquez l'apparition de la propriété `PSComputerName`. Celle-ci est bien pratique car elle nous indique le nom de la machine à qui appartient le résultat.

Établissons à présent une session permanente sur la machine « W2K8R2SRV » et voyons ce que l'on peut faire de plus par rapport à une session temporaire.

Exemple 3 : changer l'état d'un service de l'état « Paused » à l'état « Running »

Tout d'abord créons une session à distance :

```
PS > $psession = New-PSSession -ComputerName W2K8R2SRV
PS > $psession
```

Id	Name	ComputerName	State	ConfigurationName	Availability
--	----	-----	----	-----	-----
1	Session1	w2k8r2srv	Opened	Microsoft.PowerShell	Available

Ensuite, utilisons la session pour nous connecter à la machine à distance :

```
PS > Invoke-Command -Session $psession -ScriptBlock {
$S = Get-Service vmacthlp}
```

Nous avons à présent récupéré dans la variable `$s` l'objet correspondant au service « `vmacthlp` ».

Essayons de manipuler ce service afin de le sortir de son état de pause :

```
PS > Invoke-Command -Session $psession -ScriptBlock {$s}
```

Status	Name	DisplayName	PSComputerName
----	----	-----	-----
Paused	vmacthlp	Hyper-V Time Synchronization Service	w2k8r2srv

Faisons appel à la méthode `Continue` pour modifier l'état du service :

```
PS > Invoke-Command -Session $psession -ScriptBlock {$s.continue()}
```

Puis un petit `Refresh` de l'état pour voir s'il s'est passé quelque chose :

```
PS > Invoke-Command -Session $psession -ScriptBlock {$s.refresh()}
```

On rappelle la variable `$s` pour voir son contenu :

Status	Name	DisplayName	PSComputerName
----	----	-----	-----
Running	vmacthlp	Hyper-V Time Synchronization Service	w2k8r2vm

Parfait, ça a fonctionné !

À présent essayons de voir si nous pouvons faire de même en utilisant une session temporaire.

```
PS > Invoke-Command -ComputerName W2K8R2SRV -ScriptBlock {  
$s = Get-Service vmictimesync}  
PS > Invoke-Command -ComputerName W2K8R2SRV -ScriptBlock {$s}
```

Il ne se passe rien pour la simple et bonne raison qu'il n'y a pas de persistance de données entre les sessions. Lorsque l'on utilise une session temporaire, c'est comme si à chaque fois on ouvrait et on fermait une console PowerShell à distance. Par conséquent, toutes les variables sont systématiquement détruites en fin de session.

6. Exécution de scripts à distance

L'exécution de scripts à distance se fait de la même manière que l'exécution d'un bloc de script, sauf que le paramètre à utiliser à la place de **-ScriptBlock** est celui nommé **-FilePath**.

Le chemin indiqué par le paramètre **-FilePath** est le chemin du script situé sur la machine locale. Avec ce paramètre, PowerShell convertit le contenu du fichier de script en un bloc de script, transmet le bloc à l'ordinateur distant puis l'exécute sur l'ordinateur distant.

Notez cependant que pour spécifier des valeurs de paramètre au script, il faut de plus utiliser le paramètre **-ArgumentList** pour les spécifier.



Comme PowerShell effectue une conversion script -> bloc de script puis transmet le bloc de script aux ordinateurs distants, il n'y a donc pas de risque de voir l'exécution du bloc de script rejetée par les stratégies d'exécutions des machines distantes. En effet, même en mode « restricted » l'exécution de commandes est possible ; ce qui n'est pas le cas des scripts, exécutés localement.

Exemple 1 :

Exécution d'un script de récupération de l'espace disque restant (ce script est donné à la fin de cette partie)

Nous avons un petit script nommé `Get-FreeSpace.ps1`, qui comme son nom le laisse supposer, retourne l'espace disque libre de tous les lecteurs.

Voici ce que donne son exécution locale :

```
PS > C:\scripts\get-freespace.ps1
```

Système	Disque	Disponible (Go)	% restant
WIN7_US_X64	C:	1,2	10

À présent, exécutons ce script sur une machine distante et observons le résultat :

```
PS > Invoke-Command -ComputerName w2K8R2SRV -FilePath  
C:\scripts\get-freespace.ps1
```

```
Système           : W2K8R2SRV  
Disque            : C:  
Disponible (Go)   : 118,4  
% restant         : 93  
PSComputerName    : w2k8r2srv  
RunspaceId        : a145d985-e35c-42a8-816c-62fa1a1bc8a5  
PSShowComputerName : True
```

L'affichage du résultat en mode liste au lieu du mode tableau est normal dans la mesure où il y a plus de quatre propriétés à afficher (cf. Chapitre Formatage de l'affichage). Mais ce n'est pas cela le plus important... En effet, nous pouvons constater l'apparition des propriétés supplémentaires `PSComputerName`, `RunspaceId` et `PSShowComputerName`. En réalité ces propriétés sont toujours présentes lorsque l'on travaille avec les mécanismes de communication à distance PowerShell, mais elles ne sont pas toujours visibles.

Le `RunspaceId` correspond à une sorte de « bulle » dans laquelle s'exécute la session à distance PowerShell. Lorsque l'on travaille avec des sessions temporaires, le `RunspaceId` change à chaque nouvelle commande invoquée par `Invoke-`

Command ; ce qui n'est pas le cas lorsque l'on travaille avec des sessions permanentes.

Voici les paramètres de la commande Invoke-Command :

Paramètre	Description
AllowRedirection <Switch>	Autorise la redirection de cette connexion vers un autre URI (<i>Uniform Resource Identifier</i>).
ApplicationName <String>	Spécifie le segment du nom d'application dans l'URI de connexion.
ArgumentList <Object[]>	Fournit les valeurs des variables locales dans la commande. Les variables de la commande sont remplacées par ces valeurs avant l'exécution de la commande sur l'ordinateur distant. Entrez les valeurs dans une liste séparée par des virgules. Les valeurs sont associées aux variables dans leur ordre d'affichage.
AsJob <Switch>	Exécute la commande en tant que tâche en arrière-plan sur un ordinateur distant. Utilisez ce paramètre pour exécuter des commandes dont l'exécution nécessite beaucoup de temps.
Authentication <AuthenticationMechanism>	Spécifie le mécanisme permettant d'authentifier les informations d'identification de l'utilisateur.
CertificateThumbprint <String>	Spécifie le certificat de clé publique numérique (X509) d'un compte d'utilisateur qui a l'autorisation d'exécuter cette action.
ComputerName <String[]>	Spécifie les ordinateurs sur lesquels la commande s'exécute. La valeur par défaut est l'ordinateur local.
ConfigurationName <String>	Spécifie la configuration de session utilisée pour la nouvelle session PSSession.
ConnectionURI <Uri[]>	Spécifie un URI qui définit le point de terminaison de connexion.
Credential <PSCredential>	Spécifie un compte d'utilisateur qui a l'autorisation d'exécuter cette action.
FilePath <String[]>	Chemin local du script à exécuter à distance.
HideComputerName <Switch>	Omet le nom d'ordinateur de chaque objet de l'affichage de sortie (propriété PSComputerName). Par défaut, le nom de l'ordinateur qui a généré l'objet apparaît dans l'affichage.
InputObject <psobject>	Spécifie l'entrée de la commande. Entrez une variable contenant les objets ou tapez une commande ou une expression qui obtient les objets.
JobName <String>	Spécifie un nom convivial pour la tâche en arrière-plan. Par défaut, les tâches sont nommées « Tâche<n> », où <n> est un nombre ordinal. Ce paramètre est valide uniquement avec le paramètre AsJob.
Port <Int>	Spécifie le port réseau sur l'ordinateur distant utilisé pour cette commande. La valeur par défaut est le port 80 (port HTTP).
ScriptBlock <scriptblock>	Spécifie les commandes à exécuter. Placez les commandes entre accolades ({ }) pour créer un bloc de script. Ce paramètre est obligatoire.
Session <PSSession[]>	Exécute la commande dans les sessions Windows PowerShell spécifiées (PSSession). Entrez une variable qui contient les sessions PSSession ou une commande qui crée ou obtient les sessions PSSession, telle qu'une commande New-PSSession ou Get-PSSession.

SessionOption <PSSessionOption>	Définit des options avancées pour la session.
ThrottleLimit <Int>	Spécifie le nombre maximal de connexions simultanées qui peuvent être établies pour exécuter cette commande. Valeur par défaut : 32.
UseSSL <Switch>	Utilise le protocole Secure Socket Layer sur HTTPS pour établir la connexion. Par défaut SSL n'est pas utilisé.

Script Get-FreeSpace.ps1 :

```
# get-freespace.ps1
param ($computer = '.')

# récupère tous les disques logiques de l'ordinateur:

Get-WmiObject -Computer $computer Win32_LogicalDisk |
  Where {$_.drivetype -eq 3} |
  Select-Object @{e={$_.systemname};n='Système'},
    @{e={$_.name};n='Disque'},
    @{e=[math]::Round($_.freespace/1GB,1);n='Disponible (Go)'},
    @{e=[math]::Round((([int64]$_.freespace/[int64]$_.size*100),0));
      n='% restant'}
```



Vous pouvez remarquer dans cet exemple une notation un peu particulière utilisée avec la commande Select-Object. Nous lui avons passé en réalité une table de hachage. Il s'agit d'une astuce très utile, qui permet : d'effectuer des calculs sur des propriétés d'objets ; de définir le nom des propriétés de l'objet résultant. L'objet qui résulte d'une telle commande est un objet personnalisé de type PSObject.

7. Ouverture d'une console PowerShell à distance

Le dernier point à aborder avant clore cette partie sur les communications à distance PowerShell, concerne la possibilité d'exécuter des commandes en mode interactif sur une machine distante ; c'est-à-dire que toutes les commandes tapées dans la console s'exécuteront sur une machine distante. Il s'agit d'un fonctionnement similaire à la commande Telnet ; mais en plus sécurisé bien évidemment !

Pour ce faire, nous avons deux possibilités : la première consiste à ouvrir une console classique avec une commande ad-hoc, et la seconde s'appuie sur la console PowerShell en mode graphique (PowerShell ISE).

a. Enter-PSSession

La commande Enter-PSSession démarre une session interactive avec un ordinateur distant unique. Une seule session interactive peut être ouverte à la fois. Les éventuels profils PowerShell présents sur l'ordinateur distant ne sont pas chargés.

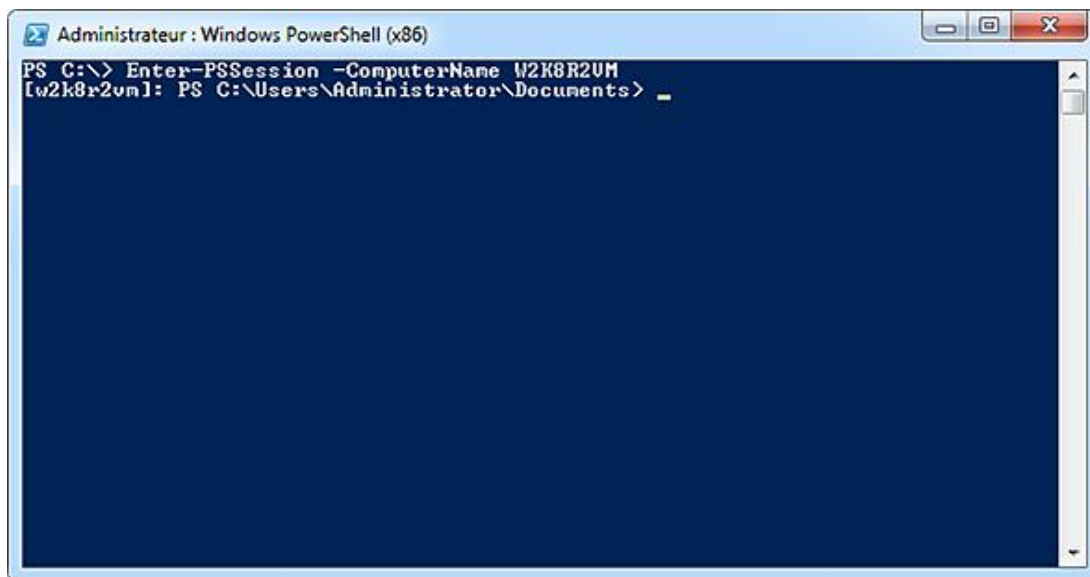
Une fois la session terminée, tapez Exit-PSSession ou simplement Exit pour vous déconnecter. Pour démarrer une session PowerShell à distance, vous devez démarrer PowerShell en tant qu'Administrateur.

On utilise généralement avec Enter-PSSession le paramètre **-ComputerName** pour spécifier le nom de l'ordinateur distant mais on peut également passer, si on le souhaite, un objet de type PSSession au paramètre **-Session**.

Exemple :

```
PS > Enter-PSSession -ComputerName W2K8R2VM
```

Le résultat de cette ligne de commandes est que l'on obtient un prompt différent. Dans ce dernier, se trouve le nom de l'ordinateur distant comme le montre la copie d'écran ci-après :



Ouverture d'une console PowerShell à distance

Les paramètres de Enter-PSSession sont les suivants :

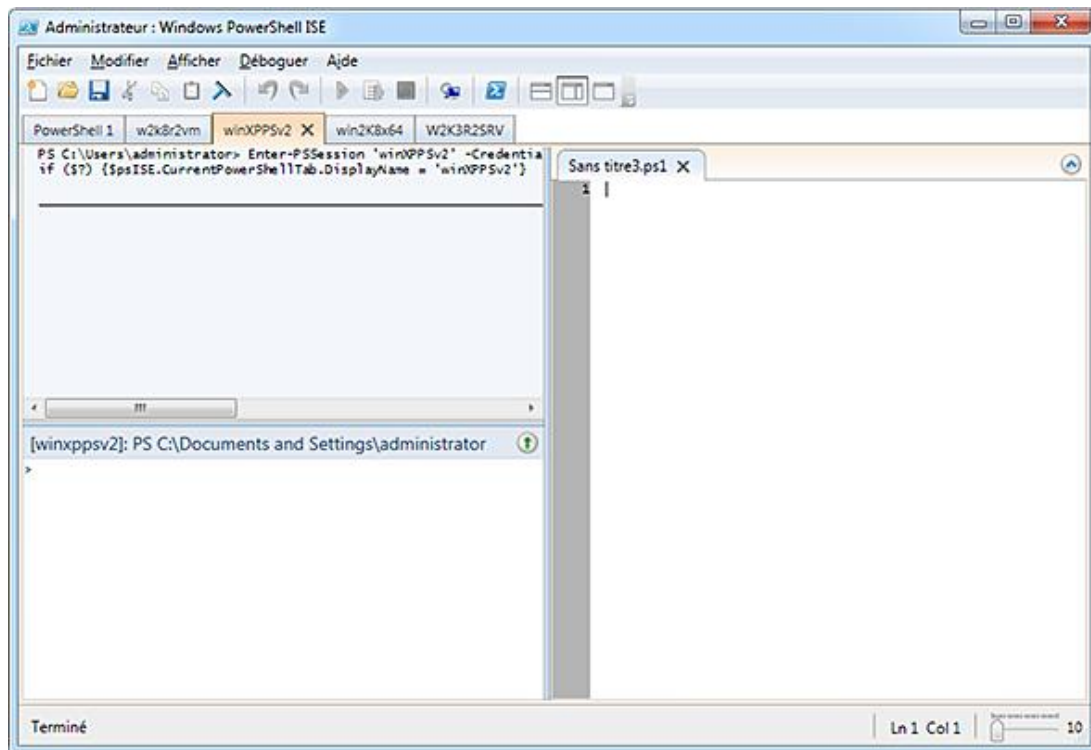
Paramètre	Description
AllowRedirection <Switch>	Autorise la redirection de cette connexion vers un autre URI (<i>Uniform Resource Identifier</i>).
ApplicationName <String>	Spécifie le segment du nom d'application dans l'URI de connexion.
Authentication <AuthenticationMechanism>	Spécifie le mécanisme permettant d'authentifier les informations d'identification de l'utilisateur.
CertificateThumbprint <String>	Spécifie le certificat de clé publique numérique (X509) d'un compte d'utilisateur qui a l'autorisation d'exécuter cette action.
ComputerName <String>	Démarre une session interactive avec l'ordinateur distant spécifié. Entrez un seul nom d'ordinateur. La valeur par défaut est l'ordinateur local.
ConfigurationName <String>	Spécifie la configuration de session utilisée pour la session interactive.
ConnectionURI <Uri[]>	Spécifie un URI qui définit le point de terminaison de connexion de la session interactive.
Credential <PSCredential>	Spécifie un compte d'utilisateur qui a l'autorisation d'exécuter cette action.
Id <int>	Spécifie l'ID d'une session existante. Enter-PSSession utilise la session spécifiée pour la session interactive. Pour rechercher l'ID d'une session, utilisez l'applet de commande Get-PSSession.
InstanceId <Guid>	Spécifie l'ID d'instance d'une session existante. Enter-PSSession utilise la session spécifiée pour la session interactive.
Name <String[]>	Spécifie un nom convivial d'une session existante.
Port <Int>	Spécifie le port réseau sur l'ordinateur distant utilisé pour cette commande. La valeur par défaut est le port 80 (port HTTP).
Session <PSSession[]>	Spécifie une session Windows PowerShell (PSSession) à utiliser

	pour la session interactive. Ce paramètre accepte un objet session. Vous pouvez également utiliser les paramètres Name, InstanceID ou ID pour spécifier une session PSSession.
SessionOption <PSSessionOption>	Définit des options avancées pour la session.
UseSSL <Switch>	Utilise le protocole Secure Socket Layer sur HTTPS pour établir la connexion. Par défaut SSL n'est pas utilisé.

b. Powershell ISE (Integrated Scripting Environment)

Cette nouvelle interface comprend une fonctionnalité qui permet d'ouvrir plusieurs consoles PowerShell à distance. Cela s'avère particulièrement pratique.

Les différentes sessions à distance sont ouvertes chacune dans un onglet propre ; voir ci-après :



PowerShell ISE peut se connecter à différents ordinateurs simultanément

Pour ouvrir une console à distance dans ISE, cela se passe dans le menu **Fichier - Nouvel onglet PowerShell à distance...** Il vous sera alors demandé de vous authentifier, puis un nouvel onglet fera son apparition. Dans l'exemple montré par la figure ci-dessus, nous avons l'onglet **PowerShell 1** qui correspond à la première instance de la console ouverte localement, puis nous avons un onglet par machine distante.

Cet exemple, lancé à partir d'un ordinateur fonctionnant sous Windows 7, montre une console PowerShell ouverte à distance sur quatre machines fonctionnant respectivement avec des systèmes d'exploitation différents : Windows Server 2008 R2, Windows XP, Windows Server 2008 et Windows Server 2003 R2.

8. Importation de commandes à distance

Supposons que sur une machine distante nous ayons des Snap-Ins ou modules que nous aimerions utiliser sur notre machine locale. Jusque là, nous pouvons nous dire que nous n'avons qu'à utiliser la commande vue précédemment Enter-PSSession. Certes cela fonctionne... Cependant, nous avons vu que Enter-PSSession ne charge pas les profils PowerShell qui pourraient se trouver sur la machine distante.

Que faire donc si l'on a plein de fonctions indispensables sur la machine distante et que l'on souhaite les utiliser sur notre machine locale ? Et bien dans ce cas de figure, le plus simple est d'importer dans la session PowerShell courante les commandes de la machine distante. Ainsi nous bénéficierons de notre profil et du jeu de commandes étendu de la machine distante.

C'est pour cela que la commandelette Import-PSSession justifie son existence. Mais il doit y avoir de nombreux autres scénarios d'usage auxquels nous n'avons pas pensé...

La commande Import-PSSession possède les paramètres suivants :

Paramètre	Description
AllowClobber <Switch>	Importe les commandes spécifiées, même si elles portent le même nom que des commandes de la session active.
ArgumentList <Object[]>	Importe la variante de la commande qui résulte de l'utilisation des arguments spécifiés.
CommandName <String[]>	Importe uniquement les commandes possédant les modèles de noms ou les noms spécifiés. Les caractères génériques sont autorisés.
CommandType <CommandTypes>	Importe uniquement les types d'objets de commande spécifiés. La valeur par défaut est Cmdlet.
FormatTypeName <String[]>	Importe les instructions de mise en forme des types Microsoft .NET Framework spécifiés.
Module <String[]>	Importe uniquement les commandes dans les composants logiciels enfichables (Snap-Ins) et les modules Windows PowerShell spécifiés.
Prefix <String>	Ajoute le préfixe spécifié aux noms des commandes importées. Utilisez ce paramètre pour éviter des conflits de nom qui peuvent se produire lorsque différentes commandes de la session ont le même nom.
Session <PSSession>	Spécifie la session PSSession à partir de laquelle les commandelettes sont importées.

Prenons, par exemple, le cas d'un serveur contrôleur de domaine Windows 2008 R2 ayant le rôle installé « Active Directory Domain Services ». Ce dernier a donc la chance de posséder le module ActiveDirectory (cf. Chapitre Module Active Directory). Ce module apporte de nombreuses commandelettes pour la gestion des objets utilisateurs, machines, groupes, OU, etc. Nous voulons donc importer ce module dans la session courante, ou autrement dit dans notre console.

La première chose à faire va consister à établir une session avec l'ordinateur distant, comme ceci :

```
PS > $s = New-PSSession -ComputerName W2K8R2VM
```

À présent, c'est comme si nous venions d'ouvrir une console PowerShell sur la machine distante. Il faut donc, dans cette dernière, charger le module Active Directory ; car pour l'instant elle ne possède que les commandes de base.

```
PS > Invoke-Command -Session $s -ScriptBlock {Import-Module ActiveDirectory}
```

On peut observer que cette commande prend quelques secondes à s'exécuter et que durant ce laps de temps une barre de progression s'affiche dans notre console.

Maintenant que le module est chargé sur la machine distante, nous allons pouvoir importer les commandes qui le composent :

```
PS > Import-PSSession -Session $s -Module ActiveDirectory

ModuleType Name                               ExportedCommands
-----
Script      tmp_3325e847-774e-4891... {Set-ADOrganizationalUnit, Get-...
```

Et voilà, c'est fait ! Pour vérifier que les commandes sont bien disponibles dans la session courante, si nous connaissons leurs noms, nous pouvons effectuer la commande suivante :

```
PS > Get-Command *-AD*

CommandType Name                                     Definition
```


-----	----	-----
Function	Add-ADComputerServiceAccount	...
Function	Add-ADDomainControllerPasswordReplicationPolicy	...
Function	Add-ADFineGrainedPasswordPolicySubject	...
Function	Add-ADGroupMember	...
Function	Add-ADPrincipalGroupMembership	...
...		

Ou cette commande si nous ne connaissons pas les commandes présentes dans le module :

```
PS > Get-Command -Module tmp*
```

Cette dernière commande liste les commandes du module nommé « tmp* ». Lorsque nous importons un module avec `Import-PSSession`, il se crée localement un module temporaire qui contient les commandes importées. Avec cette commande, nous demandons à `Get-Command` qu'elle nous fournisse la liste des commandes du module temporaire commençant par « tmp ».



Lors de l'import de commandes, ces dernières sont transformées en fonctions. De plus, gardez à l'esprit que même si les commandes importées semblent locales de part leur comportement, elles s'exécutent sur la machine distante. Il faut donc bien prendre soin de maintenir la session `PSSession` en vie le temps nécessaire.



Nous avons pris comme exemple le module Active Directory, mais sachez qu'il en aurait été de même avec le module Exchange 2010, par exemple.

Communications à distance WSMAN/WinRM avec WMI

La technologie WMI s'appuyant sur les technologies COM et DCOM, fait que celle-ci n'est pas très *firewall-friendly*. Afin de tenter de la rendre plus amicale auprès des pare-feux, le protocole WinRM se charge de l'acheminement des requêtes WMI en les encapsulant dans des trames transitant sur les ports 5985/ 5986 et 80/443 si l'on utilise la connexion via une adresse URI (voir ci-après).

Les exemples que nous fournissons sont réalisés avec une machine sous Windows 7 x64 qui sert de client et une autre machine sous Windows Server 2008 R2 qui sert de machine gérée, mais nous aurions très bien pu faire l'inverse. Ces deux machines se trouvent dans un même domaine et le pare-feu est activé sur chacune d'elles. Nous n'avons pas configuré de règles particulières sur le pare-feu, nous avons seulement laissé faire la commande Enable-PSRemoting dont nous avons discuté dans la partie précédente de ce chapitre.

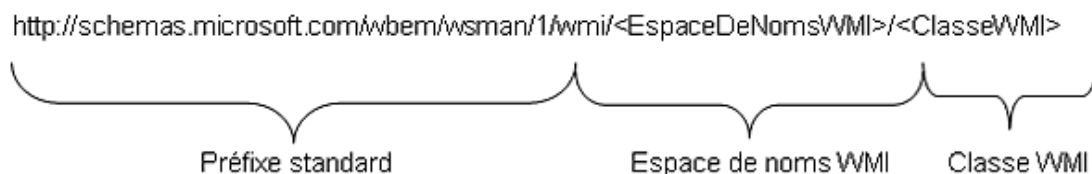
WinRM répond aux standards du Web, et par conséquent toutes les ressources auxquelles il s'adresse doivent se conformer à un certain formalisme. Chaque ressource WMI dans le monde WinRM est représentée par un URI (*Uniform Resource Identifier*).

1. Identifier une ressource WMI avec les URIs

WinRM supporte la plupart des classes WMI et opérations sur celles-ci. WinRM a donc besoin d'un mécanisme qui lui permette d'identifier toutes les ressources du système afin de pouvoir agir sur celles-ci au travers de WMI. En d'autres termes, cela signifie que nous allons pouvoir obtenir des informations ou agir sur des objets tels que les disques, les processus, les services ou les cartes réseaux à travers les classes WMI que nous connaissons déjà.

Ce sont les URIs qui vont nous permettre de faire le lien entre le protocole WS-Management et les classes WMI. Les URIs WMI ont été définis directement dans le schéma WS-Management.

Un URI est la concaténation d'un préfixe et de l'espace de nom WMI, comme ceci :



Ce qu'il faut retenir pour construire un URI :

Un URI WMI commence toujours par : `http://schemas.microsoft.com/wbem/wsman/1`

L'espace de noms WMI est de la forme `wmi/root`, `wmi/root/cimv2` (le plus souvent), `wmi/root/microsoft`, `wmi/root/directory`, etc.

Enfin il faut ajouter à la fin de l'URI le nom d'une classe WMI.

Exemples d'URIs :

```
http://schemas.microsoft.com/wbem/wsman/1/wmi/root/cimv2/Win32_Service
http://schemas.microsoft.com/wbem/wsman/1/wmi/root/cimv2/Win32_CurrentTime
http://schemas.microsoft.com/wbem/wsman/1/wmi/root/cimv2/Win32_Processor
```

Le 1 du préfixe standard de l'URI est là pour indiquer qu'il s'agit de la version 1 du protocole WS-Management.

2. Le jeu de commandes PowerShell

PowerShell v2 est doté d'un jeu de commandes spécifiques qui simplifient l'accès à la gestion avec WSMAN. On peut le scinder en deux catégories : l'une pour réaliser des opérations, l'autre pour la configuration des sessions WSMAN.

Commandes WSMAN orientées opération

Commande	Description

Test-WSMan	Teste si le service WinRM est bien démarré.
Get-WSManInstance	Affiche les informations de gestion pour une instance de ressource spécifiée par un URI de ressource.
Set-WSManInstance	Modifie les informations de gestion qui sont associées à une ressource.
New-WSManInstance	Crée une nouvelle instance d'une ressource de gestion.
Remove-WSManInstance	Supprime une instance de ressource de gestion.
Invoke-WSManAction	Appelle une action sur l'objet spécifié par l'URI de ressource et les sélecteurs.

Commandes WSMan orientées configuration

Commande	Description
Connect-WSMan	Se connecte au service WinRM sur un ordinateur distant.
Disconnect-WSMan	Déconnecte le client du service WinRM sur un ordinateur distant.
New-WSManSessionOption	Crée une table de hachage d'options de session WSMAN à utiliser comme paramètres d'entrée pour les commandes : Get-WSManInstance Set-WSManInstance Invoke-WSManAction Connect-WSMan.
Set-WSManQuickConfig	Configure l'ordinateur local pour l'administration à distance.
Get-WSManCredSSP	Obtient la configuration CredSSP (<i>Credential Security Service Provider</i>) du client.
Enable-WSManCredSSP	Active l'authentification CredSSP sur un ordinateur client.
Disable-WSManCredSSP	Désactive l'authentification CredSSP sur un ordinateur client.

3. Configuration du système

Tout d'abord il convient de tester si le service WinRM est bien en cours d'exécution. Pour ce faire utilisons la commande Test-WSMan :

```
PS > Test-WSMan -Authentication default

wsid      : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 6.1.7600 SP: 0.0 Stack: 2.0
```

Le résultat indique que la version installée de WinRM est la version 2.0.

Pour configurer le système il faut utiliser au minimum la commande Set-WSManQuickConfig. Nous disons au minimum car si vous avez déjà configuré votre système pour utiliser les communications à distance PowerShell (avec la commande Enable-PSRemoting) alors cette étape n'est pas nécessaire.

a. Lister les services d'une machine distante

Comme le résultat sera très verbeux, à l'aide de Select-Object et du paramètre -first 1, nous nous contenterons de la récupération du premier objet.

```
PS > $URI = 'http://schemas.microsoft.com/wbem/wsman/1/wmi/root/cimv2/Win32_Service'
```

```
PS > Get-WSManInstance -ResourceURI $URI -computer w2K8R2VM -Enumerate |
Select-Object -first 1

xsi           : http://www.w3.org/2001/XMLSchema-instance
p             : http://schemas.microsoft.com/wbem/wsman/1/wmi/
               root/cimv2/Win32_Service
cim           : http://schemas.dmtf.org/wbem/wscim/1/common
type          : p:Win32_Service_Type
lang          : fr-FR
AcceptPause   : false
AcceptStop    : true
Caption       : Active Directory Web Services
CheckPoint    : 0
CreationClassName : Win32_Service
Description   : This service provides a Web Service interface
               to instances of the directory service (AD DS
               and AD LDS) that are running locally on this
               server. If this service is stopped or disabled,
               client applications, such as Active Directory
               PowerShell, will not be able to access or
               manage any directory service instances that are
               running locally on this server.
DesktopInteract : false
DisplayName    : Active Directory Web Services
ErrorControl   : Normal
ExitCode       : 0
InstallDate    : InstallDate
Name           : ADWS
PathName       : C:\Windows\ADWS\Microsoft.ActiveDirectory.
               WebServices.exe
ProcessId      : 1372
ServiceSpecificExitCode : 0
ServiceType    : Own Process
Started        : true
StartMode      : Auto
StartName      : LocalSystem
State          : Running
Status         : OK
SystemCreationClassName : Win32_ComputerSystem
SystemName     : W2K8R2VM
TagId          : 0
WaitHint       : 0
```

4. Déterminer la date d'installation d'une machine distante

```
PS > $u=
'http://schemas.microsoft.com/wbem/wsman/1/wmi/root/cimv2/Win32_OperatingSystem'
PS > Get-WSManInstance -ResourceURI $u -computer w2K8R2VM -Enumerate
```

Nous restreindrons volontairement l'affichage à certaines propriétés car elles sont vraiment très nombreuses.

```
...
type          : p:Win32_OperatingSystem_Type
lang          : fr-FR
BootDevice     : \Device\HarddiskVolume1
BuildNumber    : 7600
BuildType      : Multiprocessor Free
Caption        : Microsoft Windows Server 2008 R2 Enterprise
CodeSet        : 1252
CountryCode    : 33
CreationClassName : Win32_OperatingSystem
CSCreationClassName : Win32_ComputerSystem
CSDVersion     : CSDVersion
CSName         : W2K8R2VM
CurrentTimeZone : 60
EncryptionLevel : 256
```

```
FreePhysicalMemory      : 86584
FreeSpaceInPagingFiles  : 781408
FreeVirtualMemory       : 709956
InstallDate             : InstallDate
LastBootUpTime          : LastBootUpTime
LocalDateTime           : LocalDateTime
Locale                  : 040c
Manufacturer            : Microsoft Corporation
MaxNumberOfProcesses    : 4294967295
MaxProcessMemorySize    : 8589934464
MUILanguages            : en-US
SerialNumber            : 55041-507-0304761-12345
ServicePackMajorVersion : 0
ServicePackMinorVersion : 0
SizeStoredInPagingFiles : 1048576
Status                  : OK
Version                 : 6.1.7600
WindowsDirectory        : C:\Windows
...
```

La propriété `InstallDate` n'étant pas encore accessible, nous allons l'atteindre de la façon suivante :

```
PS > $result = Get-WSManInstance -ResourceURI $u -computer w2K8R2VM
-Enumerate
PS > $result.InstallDate

Datetime
-----
2009-10-11T16:38:18+02:00
```

Ce résultat, bien que lisible n'est pas forcément le plus adapté. Mais nous pouvons facilement le transformer dans la mesure où il s'agit d'un format reconnu par le type **DateTime**. Forçons donc la conversion de cette valeur en type **DateTime** et observons le résultat :

```
PS > [DateTime]$result.InstallDate.DateTime

dimanche 11 octobre 2009 16:38:18
```

Voilà, nous avons réussi à envoyer l'équivalent d'une requête WMI à une machine distante en passant à travers les pare-feu sur le port HTTP et à en récupérer le contenu. HTTP, comme nous vous le disions précédemment, n'est pas sécurisé car les échanges passent en clair sur le réseau (excepté l'authentification). Nous vous recommandons donc vivement si vous avez à utiliser WinRM en milieu hostile d'activer le port d'écoute sur le protocole de transport HTTPS et de déployer un certificat serveur.

À la découverte d'ADSI

Active Directory Service Interfaces est une interface de programmation qui permet l'interaction avec un annuaire. ADSI prend en charge différents types d'annuaires : base de comptes locale SAM (*Security Account Manager*), LDAP (*Lightweight Directory Access Protocol*)/Active Directory, ou encore Novell Netware Directory Service (NDS) (liste non exhaustive : IIS, etc).

Cette prise en charge s'effectue par le biais des fournisseurs de services (on trouve dans la littérature anglaise le terme « Moniker » pour les désigner). C'est en spécifiant le fournisseur de services que l'on indique à ADSI avec quel annuaire travailler. ADSI a été conçu afin d'homogénéiser l'accès à tous les types d'annuaires disponibles. Ainsi il n'est pas nécessaire de connaître les spécificités de chacun, mais plutôt de savoir manipuler les interfaces ADSI.

Considérations sur la gestion d'un domaine Active Directory avec ADSI

Si vous disposez dans votre entreprise de la plate-forme de messagerie Exchange 2007 ou 2010 ou d'un contrôleur de domaine Active Directory 2008 R2, alors mieux vaut clairement utiliser leurs jeux de commandes intégrées pour gérer un domaine Active Directory, plutôt qu'ADSI. En effet, vous bénéficierez d'un jeu de commandes plus simple et mieux adapté qui vous fera gagner un temps précieux.

Si par contre, vous ne disposez pas de ces plates-formes, alors vous trouverez dans les prochaines pages tout ce qu'il faut pour gérer votre domaine. Cependant il vous faudra consentir quelques petits efforts supplémentaires car nous dialoguerons directement avec l'API ADSI.

Nous invitons donc ceux d'entre vous qui disposent d'un contrôleur de domaine Active Directory fonctionnant sous Windows Server 2008 R2 à vous reporter au chapitre Module Active Directory de Server 2008 R2 pour découvrir toutes les nouvelles commandes PowerShell. Pour les autres, on continue...

Quelques mots sur Active Directory...

L'acronyme ADSI peut prêter à confusion, car il pourrait laisser penser qu'ADSI ne sert qu'à manipuler des objets d'annuaire Active Directory. Or comme nous venons de le voir, ceci est faux. D'autant plus qu'ADSI était déjà présent dans Windows NT 4.0, donc bien avant l'annuaire Active Directory apparu avec Windows 2000 Server.

Depuis l'arrivée de Windows Server 2008, l'annuaire que l'on avait l'habitude d'appeler Active Directory a été rebaptisé Active Directory Domain Services (ADDS). Nous allons donc devoir laisser tomber nos vieilles habitudes et désormais parler d'AD DS et non plus d'AD tout court. Cette distinction est importante car Microsoft apporte dans Windows Server 2008 de nouveaux rôles comme, entre autres, la gestion des droits numériques avec Active Directory Rights Management Services (AD RMS) ou encore le service de certificats Active Directory Certificate Services (AD CS).

Manipulation de la base de comptes locale

Vous devez savoir que le support d'Active Directory Service Interfaces est perfectible. Il n'a pratiquement pas évolué entre les versions 1 et 2 de PowerShell. Vous devez prendre conscience qu'ADSI fait partie des technologies du passé. Malgré tout, dès lors qu'il s'agit de gérer une base de compte locale, vous n'aurez pas le choix que de manipuler des objets ADSI.

Comme nous allons le voir, il n'existe pas de jeu de commandes PowerShell dédié, et certaines propriétés et méthodes d'objets sont accessibles parfois de façons un petit peu... « inhabituelles ». C'est-à-dire que nous devrons par moment accéder aux objets bruts de PowerShell.

Que ce soit sur un ordinateur exécutant un système d'exploitation client (tel que Windows Vista, XP ou antérieur) ou sur un système d'exploitation serveur (hors contrôleur de domaine) il y a toujours une base de compte locale. Celle-ci, également appelée « base SAM » (*Security Account Manager*), contient des objets de type utilisateurs, et groupes d'utilisateurs, ainsi que leurs identifiants système respectifs : le SID (*Security Identifier*).

Afin d'accéder à un objet d'annuaire de type SAM ou AD DS, nous allons utiliser le raccourci [ADSI]. Ce qui changera, c'est uniquement le fournisseur de services auquel nous ferons appel. Pour la manipulation de la base SAM il s'agira du fournisseur « WinNT: ».

Veuillez noter que les majuscules et les minuscules ont ici une grande importance. Cela est une contrainte imposée non pas par PowerShell mais par ADSI et ce quel que soit le langage de script utilisé.

1. Gestion des groupes

a. Lister les groupes

Pour démarrer, listons les groupes locaux d'une machine locale ou distante, avec le petit script suivant :

```
# Get-LocalGroups.ps1
param([String]$machine='.')
$connexion = [ADSI]"WinNT://$machine"
$connexion.PSBase.Children |
    Where {$_.PSBase.SchemaClassName -eq 'group'} | foreach{$_.Name}
```

Nous récupérons la valeur du paramètre « -machine » dans la variable `$machine`. Si non définie, la valeur de cette dernière sera initialisé à la valeur « . ». Nous créons ensuite une connexion à la base SAM locale grâce à [ADSI] "WinNT://\$machine" puis nous effectuons un filtre pour ne récupérer que les objets de type groupe. Enfin nous affichons la propriété Name des objets filtrés.

Exemple d'utilisation sur une machine Windows Vista :

```
PS > ./Get-LocalGroups.ps1

Administrateurs
Duplicateurs
IIS_IUSRS
Invités
Lecteurs des journaux d'événements
Opérateurs de chiffrement
Opérateurs de configuration réseau
Opérateurs de sauvegarde
Utilisateurs
Utilisateurs avec pouvoir
Utilisateurs de l'Analyseur de performances
Utilisateurs du Bureau à distance
Utilisateurs du journal de performances
Utilisateurs du modèle COM distribué
```



Vous devez être administrateur de la machine pour que le script fonctionne.

b. Lister les membres d'un groupe

Voici le script permettant d'exécuter cette tâche :

```
# Get-LocalGroupMembers.ps1
param ([String]$machine='.',
      [String]$Groupe=$(Throw 'Nom de groupe obligatoire !'))

$connexion = [ADSI]"WinNT://$machine/$groupe,group"
$connexion.PSBase.Invoke('Members') |
    foreach{$_ .GetType().InvokeMember('Name', 'GetProperty', $null, $_, $null)}
```

Exemple d'utilisation :

```
PS > ./Get-LocalGroupMembers.ps1 -groupe Administrateurs

Administrateur
Admins du domaine
Arnaud
Robin
```

Bien que nous arrivions à lister les noms des membres, cela peut s'avérer être dans certains cas insuffisant. En effet, nous pourrions avoir dans notre groupe, des membres appartenant à un domaine.

C'est la raison pour laquelle nous allons modifier notre script afin d'obtenir en plus du nom, la précieuse information que constitue le domaine. Pour cela nous allons interroger la propriété `AdsPath` au lieu de `Name`.

```
# Get-LocalGroupMembersV2.ps1
param ([String]$machine='.',
      [String]$Groupe=$(Throw 'Nom de groupe obligatoire !'))

$connexion = [ADSI]"WinNT://$machine/$groupe,group"
$connexion.PSBase.Invoke('Members') |
%{$_ .GetType().InvokeMember('AdsPath', 'GetProperty', $null, $_, $null)}
```

Testons notre nouveau script :

```
PS > ./Get-LocalGroupMembersV2.ps1 -groupe Administrateurs

WinNT://PCVISTA/Administrateur
WinNT://PCVISTA/Robin
WinNT://PS-SCRIPTING/Admins du domaine
WinNT://PS-SCRIPTING/Arnaud
```

Nous avons bien fait de vérifier car dans cet exemple `Administrateur` et `Robin` sont des utilisateurs locaux de la machine `PCVISTA`, tandis que `Admins du domaine` et `Arnaud` sont des membres du domaine `PS-SCRIPTING`.

Remarquez que l'information retournée n'est pas dans la forme la plus courante. Habituellement on désigne un utilisateur ou un groupe sous la forme `Domaine\Utilisateur`. Qu'à cela ne tienne, PowerShell possède des opérateurs de traitement de chaînes très puissants qui vont nous régler ce petit « soucis » :

```
PS > ./Get-LocalGroupMembersV2.ps1 -groupe Administrateurs |
foreach {($_ -replace 'WinNT://', '') -replace '/', '\'}

PCVISTA\Administrateur
PCVISTA\Robin
PS-SCRIPTING\Admins du domaine
PS-SCRIPTING\Arnaud
```

Il nous a suffi d'envoyer chaque résultat à un `foreach`, qui pour chaque occurrence commence par supprimer la chaîne « `WinNT://` » en la remplaçant par une chaîne vide, puis remplace le caractère « `/` » par « `\` ».

c. Ajouter un membre à un groupe

Nous pouvons ajouter à des groupes des utilisateurs locaux ou des utilisateurs du domaine. Voyons le premier cas :

```
# Add-LocalGroupMember.ps1
param ([String]$machine='.',
```

```
[String]$Groupe=$(Throw 'Nom de groupe obligatoire !'),  
[String]$Utilisateur=$(Throw "Nom d'utilisateur obligatoire !"))
```

```
$connexion = [ADSI]"WinNT://$machine/$groupe,group"  
$connexion.Add("WinNT://$Utilisateur")
```

Veuillez noter que nous n'avons pas besoin de passer le nom de l'ordinateur local en plus du nom d'utilisateur à la méthode Add ; en effet ADSI comprend qu'il s'agit d'un utilisateur local à la machine.

Exemple d'utilisation :

```
PS > ./Add-LocalGroupMember.ps1 -groupe Administrateurs -util Arnaud
```

 Nous ne sommes pas obligés de spécifier les paramètres dans leur totalité tant qu'il n'y a pas d'ambiguïté. La preuve, nous avons utilisé **-util** au lieu de **-utilisateur**. Si par contre, nous avions eu un second paramètre nommé **-utilisation**, alors PowerShell n'aurait pas su quel paramètre associer au raccourci **-util** et nous aurait renvoyé une erreur.

Ajout d'un membre du domaine

Nous pouvons aussi ajouter un compte du domaine dans un groupe local. Pour cela nous devons utiliser la commande suivante :

```
PS > .\Add-LocalGroupMember.ps1 -g Administrateurs -u 'Domaine/Utilisateur'
```

 Attention au sens de la barre du séparateur Domaine/Utilisateur. Notre fonction utilise le fournisseur de service « WinNT:// » et vous aurez remarqué que ce dernier ne travaille qu'avec des slash et non des anti-slash (« \ »).

Libre à vous de modifier le script **Add-LocalGroupMember.ps1** pour changer la façon d'entrer des membres. Un membre pouvant être bien évidemment soit un utilisateur, soit un groupe global.

d. Supprimer un membre d'un groupe

Pour ajouter un membre à un groupe nous avons utilisé la méthode Add, pour supprimer un membre, nous utiliserons simplement la méthode Remove.

Ainsi notre script précédent devient :

```
# Remove-LocalGroupMember.ps1  
param ([String]$machine='.',  
      [String]$Groupe=$(Throw 'Nom de groupe obligatoire !'),  
      [String]$Utilisateur=$(Throw "Nom d'utilisateur obligatoire !"))  
  
$connexion = [ADSI]"WinNT://$machine/$groupe,group"  
$connexion.Remove("WinNT://$Utilisateur")
```

Exemple d'utilisation :

```
PS > ./Remove-LocalGroupMember.ps1 -groupe Administrateurs -util Robin
```

Nous venons de supprimer l'utilisateur local nommé « Robin ». Supprimons à présent un membre du domaine appartenant au groupe Administrateurs :

```
PS > ./Remove-LocalGroupMember.ps1 -g Administrateurs -u 'Domaine/Utilisateur'
```

e. Créer un groupe

Jusqu'à présent nous avons modifié des groupes déjà présents dans le système. Voyons maintenant comment créer un groupe.

```
# New-LocalGroup.ps1
param ([String]$machine='.',
      [String]$Groupe=$(Throw 'Nom de groupe obligatoire !'),
      [String]$Description)

$connexion = [ADSI]"WinNT://$machine"
$objGroupe = $connexion.Create('group', $groupe)
$objGroupe.Put('Description', $Description)
$objGroupe.SetInfo()
```

L'objet groupe possède la propriété Description que nous pouvons modifier.

Exemple d'utilisation :

```
PS > ./New-LocalGroup.ps1 -g GroupeTest -desc 'Groupe de test'
```

Il ne nous reste plus qu'à ajouter des membres grâce au script **Add-LocalGroupember.ps1** que nous avons créé précédemment.

f. Supprimer un groupe

```
# Remove-LocalGroup.ps1
param ([String]$machine='.',
      [String]$Groupe=$(Throw 'Nom de groupe obligatoire !'))

$connexion = [ADSI]"WinNT://$machine"
$connexion.Delete('group', $groupe)
```

Exemple d'utilisation :

```
PS > ./Remove-LocalGroup.ps1 -groupe GroupeTest
```

g. Modifier un groupe

Lorsque nous avons créé un groupe nous lui avons également ajouté une description en modifiant la valeur de l'attribut Description grâce à la méthode Put suivie de SetInfo.

SetInfo permet de valider une modification. Si vous omettez SetInfo lorsque vous modifiez les propriétés d'un objet, vos modifications ne seront pas prises en compte par le système.



Il existe une autre méthode qui produit les mêmes effets que **SetInfo**. Il s'agit de **CommitChanges**. La seule différence est que **CommitChanges** s'applique sur un objet de type **PSObject** et en particulier sur la propriété **PSBase**. Au cours des nombreux exemples à venir nous utiliserons indifféremment l'une ou l'autre méthode.

Sur un groupe, nous pouvons modifier deux choses :

- son nom,
- sa description.

Voyons comment renommer un groupe :

```
# Rename-LocalGroup.ps1
param ([String]$machine='.',
      [String]$OldName=$(Throw 'Nom de groupe obligatoire !'),
      [String]$NewName=$(Throw "Nom d'un groupe obligatoire !"))

$connexion = [ADSI]"WinNT://$machine/$OldName,group"
$connexion.PSBase.Rename($NewName)
$connexion.SetInfo()
```

Et maintenant comment modifier sa description :

```
# Set-LocalGroupDescription.ps1
param ([String]$machine='.',
      [String]$Groupe=$(Throw 'Nom de groupe obligatoire !'),
      [String]$Description)

$connexion = [ADSI]"WinNT://$machine/$groupe,group"
$connexion.Put('Description',$Description)
$connexion.SetInfo()
```

Exemple d'utilisation :

```
PS > ./Set-LocalGroupDescription.ps1 -g GroupeTest -d 'Nouvelle description'
```

2. Gestion des utilisateurs locaux

a. Lister les utilisateurs

L'obtention de la liste des utilisateurs de la base SAM se fait sur le même principe que le script qui nous retourne la liste des groupes. Il suffit de modifier le filtre sur la classe SchemaClassName :

```
# Get-LocalUsers.ps1
param ([String]$machine='.')

$connexion = [ADSI]"WinNT://$machine"
$connexion.PSBase.Children |
  Where {$_.PSBase.SchemaClassName -eq 'user'} | ForEach{$_ .Name}
```

Exemples d'utilisation :

```
PS > ./Get-LocalUsers.ps1
```

```
Administrateur
Arnaud
Invité
Robin
```

Pour lister les comptes d'une machine distante :

```
PS > ./Get-LocalUsers.ps1 -machine MaMachineDistante
```

```
Administrateur
HelpAssistant
Invité
SUPPORT_388945a0
```

b. Créer un utilisateur local

La création d'un utilisateur s'effectue à l'aide de la méthode Create ; à laquelle nous lui passons en premier argument « user », suivi du nom. Nous en profitons ensuite pour définir un mot de passe grâce à SetPassword. Enfin nous terminerons par l'ajout d'une description facultative.

```
# New-LocalUser.ps1
param ([String]$machine='.',
      [String]$Nom=$(Throw "Nom d'utilisateur obligatoire !"),
      [String]$MotDePasse=$(Throw 'Mot de passe obligatoire !'),
      [String]$Description)

$objMachine = [ADSI]"WinNT://$machine"
$objUser = $objMachine.Create('user', $Nom)
$objUser.SetPassword($MotDePasse)
$objUser.PSBase.InvokeSet('Description', $Description)

$objUser.SetInfo()
```

Exemples d'utilisation :

```
PS > ./New-LocalUser.ps1 -nom Jacques -MotDepasse 'P@ssw0rd'
```

Nous aurions pu également ajouter une description à notre utilisateur en faisant ainsi :

```
PS > ./New-LocalUser.ps1 -nom Jacques -MotDepasse 'P@ssw0rd' -Desc 'abcd'
```

c. Modifier un utilisateur local

Il est possible de paramétrer beaucoup plus de propriétés que le mot de passe et la description. Voici un extrait des propriétés les plus courantes pour la gestion des utilisateurs locaux :

Propriété	Type	Description
Description	String	Correspond au champ Description.
FullName	String	Correspond au champ Nom complet.
UserFlags	Integer	Permet d'ajuster l'état du compte : désactivé, le mot de passe n'expire jamais, etc.
HomeDirectory	String	Chemin du répertoire de base.
HomeDirDrive	String	Lettre associée au chemin du répertoire de base.
Profile	String	Emplacement du profil Windows.
LoginScript	String	Nom du script de login.
ObjectSID	String	SID de l'utilisateur.
PasswordAge	Time	Durée d'utilisation du mot de passe en cours en nombre de secondes depuis le dernier changement de mot de passe.
PasswordExpired	Integer	Indique si le mot de passe a expiré. Vaut zéro si le mot de passe n'a pas expiré, ou une autre valeur s'il a expiré.
PrimaryGroupID	Integer	Identifiant du groupe primaire d'appartenance de l'utilisateur.

Pour modifier une propriété de cette liste, utilisez la syntaxe suivante :

```
$objUser.PSBase.InvokeSet('Propriété', $Valeur)
```

N'oubliez pas d'utiliser la méthode SetInfo pour valider la modification.

Exemple :

```
...  
PS > $objUser.PSBase.InvokeSet('Propriété', $Valeur)  
PS > $objUser.SetInfo()
```

Nous venons de voir qu'il y avait un certain nombre de propriétés sur lesquelles nous pouvons agir. Voyons comment les découvrir avec PowerShell :

```
PS > $user=[ADSI]'WinNT://./Robin,user'  
PS > $user  
  
distinguishedName  
-----
```

Malheureusement, les propriétés de notre utilisateur ne sont pas exposées avec l'adaptateur de type ADSI. Ceci étant, il est tout de même possible d'aller regarder une partie des propriétés brutes de notre objet utilisateur grâce à la propriété PSAdapted. À présent, essayons cela :

```
PS > $user.PSAdapted

UserFlags           : {545}
MaxStorage          : {-1}
PasswordAge         : {20}
PasswordExpired     : {0}
LoginHours          : {255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255}
FullName            : {Robin Lemesle}
Description          : {Compte d'utilisateur de Robin.}
BadPasswordAttempts : {0}
LastLogin           : {20/07/2007 15:09:53}
HomeDirectory       : {}
LoginScript         : {}
Profile             : {}
HomeDirDrive        : {}
Parameters          : {}
PrimaryGroupID      : {513}
Name                : {Robin}
MinPasswordLength   : {0}
MaxPasswordAge      : {3628800}
MinPasswordAge      : {0}
PasswordHistoryLength : {0}
AutoUnlockInterval  : {1800}
LockoutObservationInterval : {1800}
MaxBadPasswordsAllowed : {0}
objectSid           : {1 5 0 0 0 0 0 5 21 0 0 0 124 224 91 123
                    165 226 143 247 33 244 133 79 232 3 0 0}
```

Essayons par exemple d'obtenir la date de dernière connexion :

```
PS > $user.PSBase.InvokeGet('lastlogin')

vendredi 20 juillet 2007 15:09:53
```

Juste par curiosité regardons de quel type se trouve cette information :

```
PS > ($user.PSBase.InvokeGet('lastlogin')).GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     DateTime                                     System.ValueType
```

L'information est de type DateTime, ce qui est parfait si l'on souhaite reformater cette information ou bien effectuer des tests de comparaison de dates.

Activer/désactiver un compte

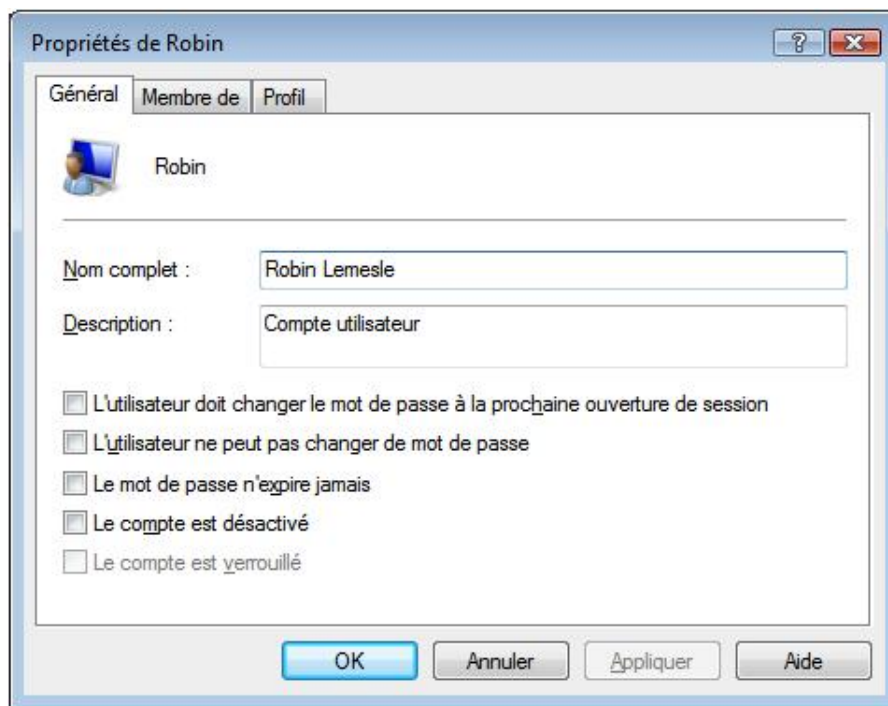
À présent, imaginons que nous voulions désactiver le compte de Robin car il ne s'est pas connecté depuis plus de 30 jours. Comment pourrions-nous faire cela ?

Eh bien nous pourrions le faire d'au moins deux façons différentes : la première consisterait à modifier la propriété invisible AccountDisabled et la seconde à modifier la valeur de la propriété UserFlags.

Première technique :

Modification d'un compte avec la propriété AccountDisabled.

Pour connaître l'état actuel du compte par l'interface graphique, il nous suffit d'aller dans le gestionnaire de l'ordinateur, puis dans **Utilisateurs et groupes locaux/Utilisateurs** et de demander les propriétés sur l'utilisateur. Ce qui devrait donner ceci :



Vérification de l'état d'un compte avec la console de gestion de l'ordinateur

Pour obtenir la même information en PowerShell, allons observer la valeur de la propriété AccountDisabled :

```
PS > $user.PSBase.InvokeGet('AccountDisabled')  
False
```

La valeur False nous est retournée, ce qui correspond à un compte actif.

- Modifions donc cette valeur, comme ceci :

```
PS > $user.PSBase.InvokeSet('AccountDisabled', $True)  
PS > $user.PSBase.CommitChanges()
```

- Vérifions que la valeur a bien été prise en compte :

```
PS > $user=[ADSI]'WinNT://./Robin,user'  
PS > $user.PSBase.InvokeGet('AccountDisabled')  
True
```



Pour être sûr que les modifications ont bien été prises en compte par le système, il faut recharger l'objet.

Deuxième technique :

Modification de la propriété UserFlags.

Tout d'abord, regardons la valeur de cette propriété :

```
PS > $user.PSBase.InvokeGet('UserFlags')  
545
```

Cette valeur est un « flag » ou drapeau auquel correspondent un ou plusieurs états.



La liste complète des flags se trouve ici : <http://msdn2.microsoft.com/en-us/library/aa772300.aspx> (ADS_USER_FLAG_ENUM Enumeration).

Pour désactiver le compte, nous allons devoir effectuer une opération logique de type OR avec le drapeau

ADS_UF_ACCOUNTDISABLE = 2, comme ceci :

```
PS > $UserFlags=$user.PSBase.InvokeGet('UserFlags')
PS > $user.PSBase.InvokeSet('UserFlags', $($UserFlags -bor 2))
PS > $user.PSBase.CommitChanges()
```

Vérifions que la valeur a bien été prise en compte :

```
PS > $user=[ADSI]'WinNT://./Robin,user'
PS > $user.PSBase.InvokeGet('AccountDisabled')
True
```

d. Supprimer un utilisateur local

La suppression d'un utilisateur s'effectue à l'aide de la méthode **Delete**.

```
# Remove-LocalUser.ps1
param ([String]$machine='.',
      [String]$Utilisateur=$(Throw "Nom d'utilisateur obligatoire !"))

$connexion = [ADSI]"WinNT://$machine"
$connexion.Delete('user', $utilisateur)
```

Exemple d'utilisation :

```
PS > ./Remove-LocalUser.ps1 -utilisateur Robin
```


Active Directory Domain Services

Bien que nous puissions quand même gérer les objets contenus dans AD DS avec le fournisseur de services « WinNT » il est néanmoins préférable d'utiliser le fournisseur « LDAP » créé spécialement pour cela.

Nous allons dans cette partie découvrir comment gérer les Unités d'Organisation (UO), les différents types de groupes, et bien sûr les utilisateurs du domaine.

1. Connexion à l'annuaire et à ses objets

Nous devons tout d'abord découvrir comment se connecter aux objets avant de pouvoir les gérer. La connexion à un objet s'effectue à l'aide du protocole LDAP, où nous devons spécifier le DN (*Distinguished Name*) de notre objet cible.

Le DN permet d'identifier de façon unique un objet dans une hiérarchie. Un DN est donc un nom unique qui représente un « chemin » vers un objet situé dans un annuaire.

Connexion à un utilisateur

Voici un exemple de DN qui identifie un objet utilisateur au sein du domaine « PS-Scripting.com » :

```
LDAP://CN=arnaud,OU=utilisateurs,DC=ps-scripting,DC=com
```

Le *Distinguished Name* se compose du CN (*Common Name*) c'est-à-dire le nom de l'objet, suivi de l'UO qui contient l'objet, suivi également du nom du domaine. Quant à ce dernier, chaque composante du domaine doit être spécifiée en utilisant la notation « DC= » (DC, pour *Domain Component*).

Le DN se construit en commençant par le CN de l'objet puis en remontant la hiérarchie du domaine jusqu'à atteindre la racine.

Si notre objet utilisateur avait été contenu dans une hiérarchie d'unités d'organisation, nous pourrions construire le DN de la façon suivante :

```
LDAP://CN=arnaud,OU=Info,OU=France,DC=ps-scripting,DC=com
```

Dans cet exemple, l'utilisateur « arnaud » se situe dans l'UO « Info » qui elle-même se situe dans l'UO « France ».

Maintenant que nous savons créer un DN, nous devons le passer à notre adaptateur de type [ADSI] pour pouvoir nous connecter à un objet. Cela se fait de la façon suivante :

```
PS > $objUser=[ADSI]'LDAP://CN=arnaud,OU=Info,OU=France,DC=ps-scripting,DC=com'
```

À présent que nous sommes connectés à un objet, nous allons pouvoir manipuler ou simplement observer ses propriétés. Pour vérifier la connexion, essayez d'afficher simplement la variable \$objUser, comme ci-dessous :

```
PS > $objUser
distinguishedName
-----
{CN=arnaud,OU=Info,OU=France,DC=ps-scripting,DC=com}
```

Si le DN s'affiche comme ci-dessus, c'est que notre requête LDAP a abouti et qu'elle est valide. Dans le cas contraire, nous obtenons un message d'erreur.

Connexion à la racine du domaine actuel

Nous venons de voir comment nous connecter à un utilisateur dont nous connaissons à l'avance le DN, mais ce n'est pas toujours le cas. Ainsi grâce à une requête LDAP appropriée il nous est possible de rechercher un objet dans l'annuaire. Mais avant d'en arriver là, il faut que l'on se connecte à la racine de l'annuaire.

Pour ce faire, nous avons vu dans l'exemple précédent qu'il nous suffisait de spécifier le nom du domaine sous la forme DC=MonSousDomaine,DC=MonDomaine. Ainsi nous pourrions écrire ceci :

```
$objDomain=[ADSI]'LDAP://DC=ps-scripting,DC=com'
```

En effet ça fonctionne, mais cela implique que nous connaissions une fois de plus le nom du domaine à l'avance. Il existe une autre technique, beaucoup plus triviale qui fait la même chose. À présent écrivons ceci :

```
$objDomaine=[ADSI] ''
```

Pour vérifier que nous sommes bien connectés, tentons d'afficher le contenu de \$objDomaine :

```
PS > $objDomaine  
  
distinguishedName  
-----  
{DC=ps-scripting,DC=com}
```

Cette syntaxe a l'avantage non seulement de la concision, mais surtout elle apporte une certaine indépendance vis-à-vis de l'AD DS ; elle va donc assurer une meilleure portabilité de nos scripts.

Connexion à un catalogue global

Afin d'étendre le périmètre de recherche à tous les utilisateurs, groupes globaux et universels de la forêt, nous pouvons aussi nous connecter à un catalogue global grâce au préfixe « GC:// ».

```
$objGC=[ADSI] 'GC://DC=ps-scripting,DC=com'
```

Vous remarquerez que tout ce que nous avons à faire est de remplacer « LDAP » par « GC » dans notre requête de connexion. Celle-ci nous permettra de nous connecter à un catalogue global et de dialoguer avec ce dernier sur le port TCP 3268 au lieu de 389 (utilisé pour LDAP).



N'oubliez pas qu'en vous connectant à un catalogue global vous aurez accès à davantage d'objets de l'annuaire (tous ceux de la forêt) ; mais ces objets peuvent avoir moins d'attributs que lorsque vous interrogez le contrôleur de domaine qui les héberge. Ceci est vrai, en particulier, pour les comptes utilisateurs.

Connexion à un domaine distant sous une autre identité

Lorsque nous utilisons le raccourci [ADSI], PowerShell nous simplifie en réalité l'accès aux classes du Framework .NET. Ainsi si nous regardions de plus près le type de notre variable \$objDomaine, nous constaterions qu'il s'agit d'un objet de type System.DirectoryServices.DirectoryEntry.

En passant par la notation [ADSI] nous ne pouvons pas tirer parti de toutes les fonctionnalités de la classe DirectoryEntry, dont celle qui permet « l'impersonation ». Nous ne traduirons pas ce terme car il n'y a pas vraiment de terme français qui pourrait le définir. L'*impersonation* est un mécanisme qui permet l'exécution de code sous l'identité d'un autre utilisateur. Nous y aurons accès en faisant directement appel au Framework .NET.

Par ce biais, nous allons pouvoir faire les deux opérations suivantes :

- nous connecter à un domaine autre que celui dans lequel se trouve notre machine,
- nous connecter dans le domaine de notre machine sous une autre identité.

Le premier point est très important car vous pouvez disposer de plusieurs domaines dans votre société, et vouloir effectuer des tâches d'administration dans le domaine A alors que votre machine se trouve dans le domaine B. Cela est également vrai si votre machine se trouve dans un *workgroup*, autrement dit à l'extérieur de tout domaine.

Pour ce faire, utilisons le morceau de code suivant :

```
$objDomaine = New-Object System.DirectoryServices.DirectoryEntry(  
'LDAP://Nom_ou_IP_Domaine', 'CompteAdmin', 'motdepasse')
```

Nom_ou_IP_Domaine : vous devez spécifier ici, soit un nom de domaine complet (FQDN (*Fully Qualified Domain Name*)) (par exemple ps-scripting.com), soit l'adresse IP d'un contrôleur de domaine.

Exemple :

```
PS > $objDomaine = New-Object System.DirectoryServices.DirectoryEntry(  
'LDAP://192.160.1.1', 'administrateur', 'P@ssw0rd')
```

À présent que nous sommes connectés, il ne nous reste plus qu'à définir les actions que l'on souhaite faire sur les objets de l'AD DS. C'est ce que nous allons voir dans la partie suivante...

Nous venons de découvrir les différentes façons de se connecter à AD DS : soit au travers du raccourci [ADSI], soit en utilisant la classe DirectoryEntry du Framework .NET. À vous de choisir celle qui vous correspond le mieux en

fonction de votre contexte de travail et de ce que vous voulez faire.



Pour tous les exemples qui suivront, nous ferons toujours comme si nous administrions le domaine courant. C'est-à-dire le domaine dans lequel se trouve notre machine et à partir de laquelle nous exécutons nos commandes. Nous supposons aussi que nous sommes connectés avec un compte disposant des droits administrateur du domaine.

2. Recherche d'objets

Toujours grâce au Framework .NET et plus particulièrement à la classe `DirectorySearcher` nous allons pouvoir effectuer de puissantes recherches dans le service d'annuaire Active Directory. Lorsque l'on instancie un objet `DirectorySearcher`, nous devons obligatoirement lui passer en paramètre le chemin à partir duquel la recherche doit avoir lieu. En option et de façon à restreindre la recherche, nous pouvons préciser un filtre de recherche ainsi qu'une liste de propriétés à rechercher.

Nous avons également à disposition, un grand jeu de propriétés dont voici un extrait des plus usuelles :

Propriété	Type	Description
CacheResults	Boolean	Spécifie si le résultat de la recherche doit être ou non stocké dans le cache de l'ordinateur. Valeur par défaut True . Il est conseillé de mettre la valeur à False si la recherche retourne un grand nombre d'objets.
ServerTimeLimit	TimeSpan	Temps maximum alloué à la recherche. Par défaut 120 secondes (valeur -1). Lorsque la limite de temps est atteinte, la recherche s'interrompt et le serveur retourne les résultats trouvés.
Filter	String	Définit un filtre au format LDAP tel que « (objectClass=user) ». Le filtre appliqué par défaut est « (objectClass=*) ». Il est possible de créer des filtres élaborés à base des opérateurs & (ET) et (OU). Exemple : (&(objectClass= user) (lastName=dupont)). Les parenthèses sont obligatoires.
SearchScope	String	Domaine de recherche. La recherche peut se restreindre à l'UO spécifiée (valeur Base), à un seul niveau de profondeur (valeur OneLevel), à tous les sous conteneurs (valeur SubTree). Défaut : SubTree .
PageSize	Integer	Nombre de résultats à retourner par pages de recherche. Par défaut , <code>PageSize</code> vaut zéro , ce qui signifie que ce mécanisme n'est pas actif. Par conséquent, le nombre de résultats retournés ne peut excéder 1000 . Pour dépasser cette limite vous devez indiquer une valeur supérieure à zéro pour cette propriété.
PropertiesToLoad	String	Jeu de propriétés à récupérer. Par défaut toutes les propriétés sont récupérées.

Enfin, nous disposons des deux méthodes suivantes :

`FindOne` : exécute la requête et ne retourne que le premier résultat trouvé.

`FindAll` : exécute la requête et retourne une collection contenant les objets trouvés.

À présent, essayons nous à une petite recherche. Testons ces quelques lignes :

```
PS > $objDomaine = [ADSI]''
PS > $objRecherche =
    New-Object System.DirectoryServices.DirectorySearcher($objDomaine)

PS > $objRecherche

CacheResults           : True
ClientTimeout          : -00:00:01
PropertyNamesOnly      : False
```

```

Filter                : (objectClass=*)
PageSize              : 0
PropertiesToLoad       : {}
ReferralChasing       : External
SearchScope           : Subtree
ServerPageTimeLimit   : -00:00:01
ServerTimeLimit       : -00:00:01
SizeLimit             : 0
SearchRoot            : System.DirectoryServices.DirectoryEntry
Sort                  : System.DirectoryServices.SortOption
Asynchronous          : False
Tombstone             : False
AttributeScopeQuery   :
DerefAlias            : Never
SecurityMasks         : None
ExtendedDN            : None
DirectorySynchronization :
VirtualListView       :
Site                  :
Container             :

```

Nous venons de créer un objet DirectorySearcher et nous avons affiché ses propriétés. Nous retrouvons bien les propriétés et leurs valeurs par défaut que nous avons détaillées dans le tableau précédent.

À présent, pour exécuter notre requête, nous devons utiliser l'une des deux méthodes FindOne ou FindAll.

Essayons la méthode FindOne :

```

PS > $objRecherche.FindOne() | Format-List

Path          : LDAP://ps-scripting.com/DC=powershell-scripting,DC=com
Properties     : {fsmoroleowner, minpwdlength, adspath, msds-perusertrustto...}

```

a. Obtenir la liste des unités d'organisation

Désormais nous maîtrisons parfaitement la recherche, ainsi obtenir la liste des UO est presque un jeu d'enfant. Comme d'habitude, nous devons d'abord nous connecter à Active Directory. La connexion doit s'effectuer en fonction de la requête que l'on souhaite faire. Devons-nous lister toutes les UO de l'AD DS ou seulement celles contenues dans une UO particulière ? C'est la réponse à cette question qui va déterminer la constitution de la requête de connexion à l'annuaire.

Listons toutes les UO à partir de la racine :

```

PS > $objDomaine = [ADSI]''
PS > $objRecherche =
    New-Object System.DirectoryServices.DirectorySearcher($objDomaine)

PS > $objRecherche.Filter='(objectCategory=organizationalUnit)'
PS > $objRecherche.FindAll() | Format-List

Path          : LDAP://ps-scripting.com/OU=Domain Controllers,
                DC=ps-scripting,DC=com
Properties     : {ou, name, whencreated, iscriticalsystemobject...}

Path          : LDAP://ps-scripting.com/OU=Test,DC=ps-scripting,DC=com
Properties     : {usncreated, objectclass, distinguishedname,objectguid...}

Path          : LDAP://ps-scripting.com/OU=HR,DC=ps-scripting,DC=com
Properties     : {usncreated, objectclass, distinguishedname,objectguid...}

Path          : LDAP://ps-scripting.com/OU=SousOU1,OU=Test,
                DC=ps-scripting,DC=com
Properties     : {usncreated, objectclass, distinguishedname,objectguid...}
Path          : LDAP://ps-scripting.com/OU=SousOU2,OU=Test,
                DC=ps-scripting,DC=com
Properties     : {usncreated, objectclass, distinguishedname,objectguid...}

```

Envie de les compter ? Rien de plus facile.

```
PS > $objRecherche.FindAll().count  
5
```

Maintenant supposons que nous voulions ne retrouver que les UO qui commencent par la lettre « T ». Pour ce faire, nous allons juste devoir apporter une modification à la propriété Filter en lui adjoignant un autre critère ; comme ceci :

```
PS > $objRecherche.Filter='(&(objectCategory=organizationalUnit)(ou=t*))'  
PS > $objRecherche.FindAll()
```

b. Obtenir la liste des utilisateurs

L'extraction de la liste des utilisateurs se fait exactement de la même façon que pour les UO, à la nuance près du filtre. En effet, seule la classe d'objet diffère.

Liste de tous les utilisateurs de l'AD DS à partir de la racine :

```
PS > $objDomaine = [ADSI]''  
PS > $objRecherche =  
New-Object System.DirectoryServices.DirectorySearcher($objDomaine)  
PS > $objRecherche.Filter='(&(objectCategory=person)(objectClass=user))'  
PS > $objRecherche.FindAll()
```

Path	Properties
----	-----
LDAP://CN=Administrateur,CN=Users,DC=ps-scriptin...	{samaccounttype, lastlogon, lastlogontimestamp, objectsi...
LDAP://CN=Invité,CN=Users,DC=ps-scripting,DC=com	{samaccounttype, lastlogon, objectsid, whencreated...}
LDAP://CN=SUPPORT_388945a0,CN=Users,DC=ps-script...	{samaccounttype, lastlogon, objectsid, whencreated...}
LDAP://CN=arnaud,OU=HR,DC=ps-scripting,DC=com	{samaccounttype, countrycode, cn, lastlogoff...}
LDAP://CN=krbtgt,CN=Users,DC=ps-scripting,DC=com	{samaccounttype, lastlogon, objectsid, whencreated...}
LDAP://CN=MyerKen,OU=Test,DC=ps-scripting,DC=com	{lastlogon, objectsid, whencreated, badpasswordtime...}
LDAP://CN=Utilisateur tititoto,OU=Test,DC=ps-scr...	{lastlogon, objectsid, whencreated, primarygroupid...}
LDAP://CN=Utilisateur tititoto2,OU=Test,DC=ps-sc...	{lastlogon, objectsid, whencreated, badpasswordtime...}



Pour découvrir en détail comment réaliser des filtres performants, nous vous recommandons vivement de consulter les pages suivantes du site MSDN : <http://msdn2.microsoft.com/en-us/library/ms808539.aspx> (Creating More Efficient...) <http://msdn2.microsoft.com/en-us/library/aa746475.aspx> (Search Filter Syntax)

Si vous n'êtes pas très à l'aise avec la définition de filtres, nous vous conseillons d'utiliser l'outil d'administration « Utilisateurs et Ordinateurs Active Directory ». Ainsi grâce à lui vous allez pouvoir créer graphiquement des requêtes personnalisées en quelques clics, les tester et récupérer la valeur du champ « Chaîne de recherche ». Ce champ correspond en réalité à la propriété Filter de l'objet DirectorySearcher. Il ne nous restera donc plus qu'à copier/coller cette valeur et notre requête sera créée.

Création d'un filtre de recherche LDAP avec les outils Windows

c. Obtenir la liste des groupes

Toujours sur le même principe, nous allons pouvoir récupérer la liste des groupes. En fait toute la difficulté réside dans la définition du filtre. Il s'agit néanmoins d'une difficulté toute relative dans la mesure où le plus « compliqué » est finalement de connaître les nombreuses classes et les attributs qui composent le schéma d'Active Directory.



Pour découvrir les attributs et les classes qui composent le schéma ainsi que leurs valeurs, nous vous conseillons l'outil du Support Tools nommé ADSIEdit.msc. Mais prenez garde de ne rien modifier si vous n'êtes pas sûr de vous !

Pour lister les groupes nous allons effectuer un filtre sur tous les objets dont l'attribut objectCategory est un groupe. Notre filtre sera donc de la forme suivante :

```
PS > $objDomaine = [ADSI]''
PS > $objRecherche = New-Object `
System.DirectoryServices.DirectorySearcher($objDomaine)
PS > $objRecherche.Filter='(objectCategory=group)'
```

Ces quelques instructions nous retournent tous les groupes, sans distinctions, contenus dans notre Active Directory.

Comme vous le savez, il existe plusieurs sortes de groupes : les locaux, les globaux et les universels. Pour ne récupérer que les uns ou que les autres, nous allons devoir peaufiner notre filtre.

Il existe un attribut qui spécifie justement quel est le type d'un groupe ; il s'agit de l'attribut GroupType. Celui-ci peut contenir une valeur qui est une combinaison des valeurs suivantes :

Valeur	Description
2	Groupe global.
4	Groupe local.
8	Groupe universel.
2147483648 (0x80000000)	Groupe de sécurité. Si cette valeur n'est pas spécifiée, c'est qu'il s'agit d'un groupe de distribution.

En résumé, si nous recherchons tous les objets groupes dont l'attribut GroupType vaut 0x80000002 (en hexadécimal) ou 2147483650 (en décimal), nous trouverons tous les groupes globaux de sécurité. Et c'est justement ce que nous allons faire :

```
PS > $objDomaine = [ADSI]''
PS > $objRecherche = New-Object `
    System.DirectoryServices.DirectorySearcher($objDomaine)
PS > $objRecherche.Filter='(&(objectCategory=group)(GroupType= 2147483650))'
PS > $objRecherche.FindAll()
```

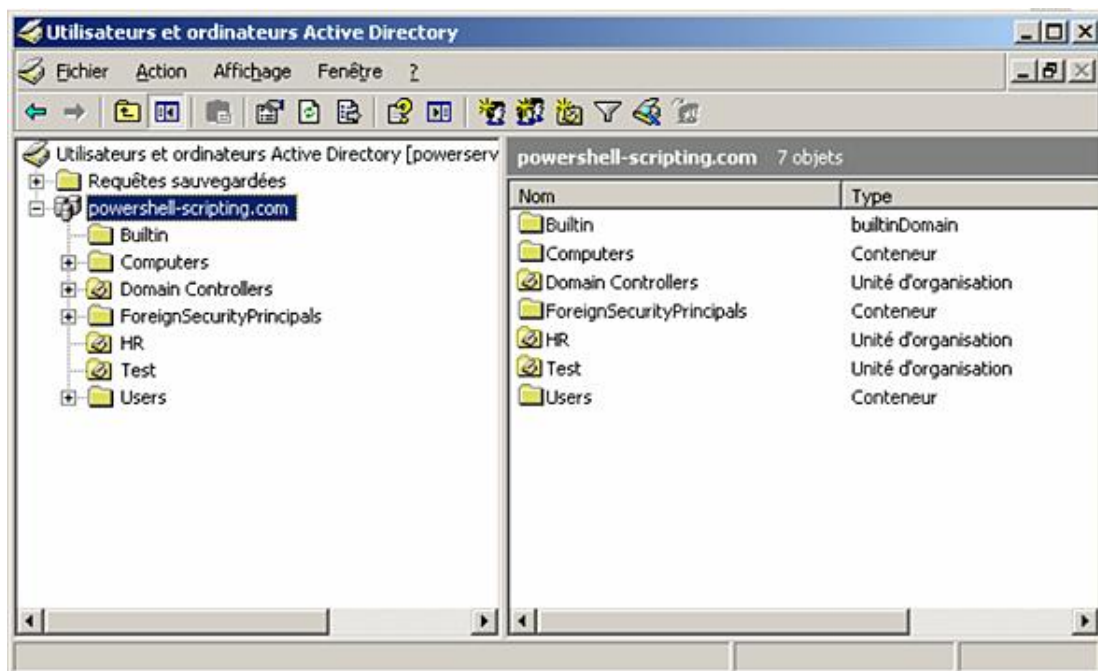
Path	Properties
----	-----
LDAP://ps-scripting.com/CN=Ordinateurs du domain...	{objectguid, grouptype, obj...}
LDAP://ps-scripting.com/CN=Contrôleurs de domain...	{objectguid, grouptype, obj...}
LDAP://ps-scripting.com/CN=Admins du domaine,CN=...	{samaccounttype, objectguid...}
LDAP://ps-scripting.com/CN=Utilisa. du domaine,C...	{objectguid, grouptype, obj...}
LDAP://ps-scripting.com/CN=Invités du domaine,CN...	{objectguid, grouptype, obj...}
LDAP://ps-scripting.com/CN=Propriétaires créateu...	{objectguid, grouptype, obj...}
LDAP://ps-scripting.com/CN=DnsUpdateProxy,CN=Use...	{objectguid, grouptype, obj...}
LDAP://ps-scripting.com/CN=MonGroupe,OU=Test,DC=...	{objectguid, grouptype, obj...}
LDAP://ps-scripting.com/CN=test,OU=HR,DC=powersh...	{usncreated, cn, grouptype, ...}

3. Gestion des unités d'organisation (UO)

Pour ceux qui ne connaissent pas, une unité d'organisation est un container dans lequel nous allons pouvoir ranger des objets. Il est intéressant de créer des UO lorsqu'il y a beaucoup d'objets à ranger, et ce pour des questions d'organisation. En effet, lorsque l'on a plusieurs milliers de comptes utilisateurs et de groupes dans une même UO on a vite fait de s'y perdre. De plus, l'autre intérêt immédiat des UO est qu'il est possible de leur appliquer (on dit aussi « lier ») des stratégies de groupes ou GPO (*Group Policy Object*).

Une stratégie de groupe est un ensemble de paramètres définis de façon centralisée par l'administrateur du domaine. Ces paramètres, qui correspondent généralement à des clés de registre, s'appliquent sur les objets contenus à l'intérieur de l'UO ; ces objets sont la plupart du temps de type utilisateur ou ordinateur.

Voici une arborescence de domaine typique :



Arborescence de domaine vue avec « Utilisateurs et Ordinateurs Active Directory »

Vous constaterez que certaines UO possèdent une icône représentant un dossier avec un petit livre à l'intérieur. Cela signifie que ces unités d'organisation peuvent être liées à des stratégies de groupes. Il faut savoir que les UO dites « built-in », c'est-à-dire créées par défaut par le système ne peuvent être associées à des GPO ; exception faite de l'UO « Domain Controllers ».

Un autre avantage de créer une UO, est qu'il est possible de déléguer sa gestion à un utilisateur non membre du groupe « Admins du domaine ». Cela peut permettre, par exemple, à un gestionnaire d'UO de gérer les membres des groupes (à condition que les groupes et les utilisateurs qu'il gère soient membres de son UO). Il pourrait, en outre et si l'administrateur lui en a donné la permission, réinitialiser les mots de passe de ses collaborateurs.

Enfin, sachez qu'il est possible d'imbriquer plusieurs unités d'organisation.

a. Créer une unité d'organisation

Comme d'habitude tout commence par la connexion à l'annuaire. Nous devons nous connecter à l'endroit où nous voulons créer l'UO. Par exemple, créons une UO à la racine d'AD DS :

```
PS > $objDomaine = [ADSI]''
PS > $objOU = $objDomaine.Create('organizationalUnit','ou=Finance')
PS > $objOU.Put('description', 'Services financiers')
PS > $objOU.SetInfo()
```

Nous venons de créer l'UO nommée « Finance » et lui avons donné une description. À présent, créons une nouvelle UO à l'intérieur de celle-ci qui contiendra les objets relatifs à la comptabilité :

```
PS > $adsPath = 'LDAP://OU=Finance,' + ([ADSI]'' ).distinguishedName
PS > $objomaine = [ADSI]$adsPath
PS > $objOU = $objDomain.Create('organizationalUnit','ou=Comptabilité')
PS > $objOU.Put('description','Bureau comptable')
PS > $objOU.SetInfo()
```

b. Renommer une unité d'organisation

Il y a une restructuration dans votre entreprise qui fait que le service financier a pris de l'ampleur, il englobe désormais le service contrôle de gestion. En tant qu'administrateur système consciencieux, vous allez renommer l'UO « finance » en « finance et gestion ».

Pour ce faire vous allez utiliser les quelques lignes suivantes :

```
PS > $objOU = [ADSI]''
PS > $objOU.MoveHere('LDAP://OU=Finance,DC=ps-scripting,DC=com',
```



```
'OU=Finance et gestion')
```

Le renommage d'objets dans Active Directory s'effectue grâce à la méthode MoveHere. Celle-ci peut également servir au déplacement d'objets.

c. Déplacement d'objets dans une unité d'organisation

Déplacement d'un groupe

Une fois encore la méthode MoveHere va nous être utile. Nous allons pour cet exemple déplacer le groupe « Chargé de clients » présent dans l'OU « informatique » dans l'OU « Utilisateurs ».

Pour ce faire, il nous faut d'abord nous connecter à l'OU de destination car la méthode MoveHere appartient à cette dernière. Nous devons ensuite donner à MoveHere le DN du groupe à déplacer, ainsi que son nouveau nom (nom qui peut être identique à son nom d'origine si vous ne souhaitez pas le changer).

```
PS > $objUODest = [ADSI]'LDAP://OU=Utilisateurs,DC=ps-scripting,DC=com'  
PS > $objUODest.MoveHere('LDAP://CN=Chargés de clients,OU=Informatique,  
DC=ps-scripting,DC=com', 'CN=Chargés de clients')
```

Vous remarquerez que l'on commence par spécifier l'OU de destination au lieu de l'OU source comme habituellement. Bien que cela puisse être déroutant, vous devez raisonner à l'envers en vous posant la question suivante : « Où est-ce que doit aller mon objet ? ».

Enfin, dans le second argument de MoveHere, vous pouvez spécifier le nouveau nom de l'objet si vous voulez le renommer en même temps que vous le déplacez.

Déplacement d'un utilisateur

Dans cet exemple nous déplaçons l'utilisateur Arnaud, de l'OU « Utilisateurs » à l'OU « Users ».

```
PS > $objUODest = [ADSI]'LDAP://CN=Users,DC=ps-scripting,DC=com'  
PS > $objUODest.MoveHere('LDAP://CN=Arnaud,OU=Utilisateurs,  
DC=ps-scripting,DC=com', 'CN=Arnaud')
```

Déplacement d'une OU

Pour illustrer le déplacement d'une unité d'organisation, déplaçons l'OU nommée « Utilisateurs » située à la racine d'AD DS dans l'OU « Informatique ».

```
PS > $objUODest = [ADSI]'LDAP://OU=Informatique,DC=ps-scripting,DC=com'  
PS > $objUODest.MoveHere('LDAP://OU=Utilisateurs,  
DC=ps-scripting,DC=com', 'OU=Utilisateurs')
```

Notez que nous pouvons également renommer l'OU que nous déplaçons en spécifiant le nouveau nom en tant que second argument dans la méthode MoveHere.

d. Supprimer une unité d'organisation

La suppression d'une OU s'effectue en deux temps :

- Connexion au conteneur dans lequel se trouve l'OU.
- Suppression de l'OU.

Imaginons cette fois que notre entreprise ait externalisé la gestion et la finance. Dans ces conditions, l'OU « Finance et gestion » n'a plus de raison d'être. C'est la raison pour laquelle nous allons la supprimer, avec le script suivant :

```
PS > $objOU = [ADSI]''  
PS > $objOU.Delete('organizationalUnit','OU=Finance et gestion')
```

Pour que le script fonctionne, il faut au préalable que nous ayons complètement vidé l'OU.

4. Gestion des groupes

a. Créer un groupe

Avant de créer un groupe, vous devez connaître son étendue. Elle peut être globale, locale ou universelle. Si vous ne spécifiez pas d'étendue, alors par défaut, il sera créé un groupe global. Pour nos exemples, nous avons créé une UO nommée « Informatique » et nous créerons des groupes à l'intérieur.



Pour plus d'informations sur les groupes de sécurité, leurs rôles et leurs étendues, rendez-vous sur le site Microsoft TechNet, à l'adresse suivante : <http://www.microsoft.com/technet/prodtechnol/windowsserver2003/fr/library/ServerHelp/79d93e46-ecab-4165-8001-7adc3c9f804e.mspx?mfr=true>

Création d'un groupe global

Créons un groupe nommé « Responsables projets ». Notez que si vous ne renseignez pas de SamAccountName, Windows en attribuera un par défaut, et ce dernier n'aura pas un nom très compréhensible. Vous pouvez spécifier n'importe quelle valeur mais nous vous conseillons de mettre la même que le CN.

```
PS > $objOU = [ADSI]'LDAP://OU=Informatique, DC=ps-scripting, DC=com'
PS > $objGroupe = $objOU.Create('group', 'cn=Responsables projets')
PS > $objGroupe.Put('samaccountname', 'Responsables projets')
PS > $objGroupe.Put('description', 'Responsables des projets informatiques')
PS > $objGroupe.SetInfo()
```

Création d'un groupe local

Pour créer un groupe d'une autre étendue qu'un groupe global nous allons devoir spécifier une valeur pour l'attribut GroupType. La liste des valeurs est la même que celle que nous avons utilisée pour effectuer un filtre de recherche.

Pour un groupe local, la valeur de GroupType sera **2147483652**.

Créons un groupe local, toujours dans l'UO « Informatique », nommé « Chargés de clients », comme ceci :

```
PS > $objOU = [ADSI]'LDAP://OU=Informatique, DC=ps-scripting, DC=com'
PS > $objGroupe = $objOU.Create('group', 'cn=Chargés de clients')
PS > $objGroupe.Put('samaccountname', 'Chargés de clients')
PS > $objGroupe.Put('groupType', '2147483652')
PS > $objGroupe.Put('description', 'Chargés de clients informatique')
PS > $objGroupe.SetInfo()
```

Création d'un groupe universel

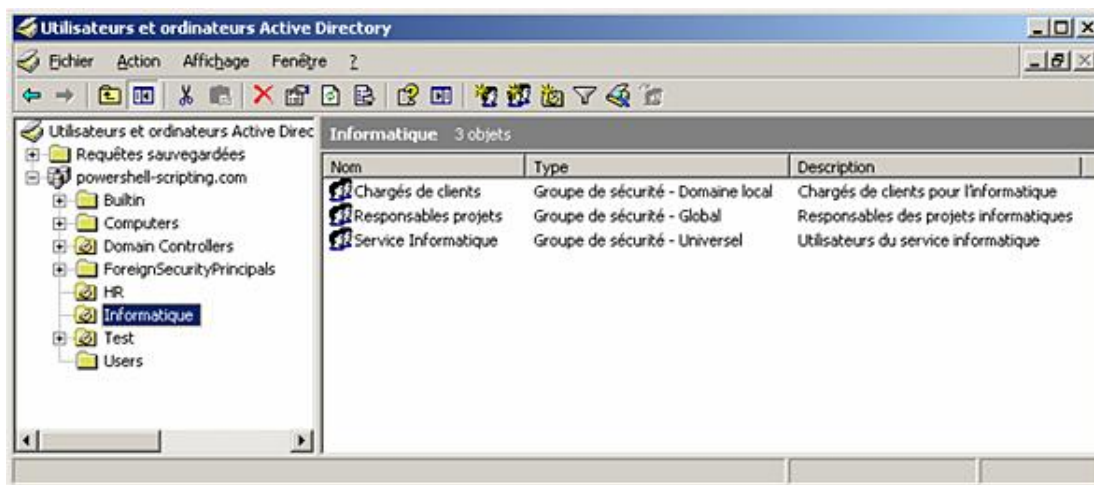
Pour créer un groupe universel de sécurité, rien de plus facile ; il suffit de modifier une fois encore la valeur de l'attribut GroupType.

Pour un groupe local, la valeur de GroupType sera **2147483656**.

Créons le groupe universel nommé « Service Informatique », comme ceci :

```
PS > $objOU = [ADSI]'LDAP://OU=Informatique, DC=ps-scripting, DC=com'
PS > $objGroupe = $objOU.Create('group', 'cn=Service Informatique')
PS > $objGroupe.Put('samaccountname', 'Service Informatique')
PS > $objGroupe.Put('groupType', '2147483656')
PS > $objGroupe.Put('description', 'Utilisateurs du service informatique')
PS > $objGroupe.SetInfo()
```

Vérifions la création de tous nos groupes :



Vérification de la création des groupes

b. Affecter un ou plusieurs membres à un groupe

Ajout d'un membre

Pour mettre un objet dans un groupe, il suffit de se connecter au groupe puis d'utiliser la méthode Add. Cette méthode conserve le contenu existant du groupe et ajoute l'objet spécifié en argument. L'objet peut être un utilisateur ou un autre groupe si l'on veut faire une imbrication de groupes. Enfin pour valider la modification, il ne faut pas oublier d'utiliser la méthode SetInfo.

Supposons que nous voulions ajouter l'utilisateur « Jacques » au groupe « Chargés de clients » ; ce dernier étant situé dans l'UO « Informatique ».

```
PS > $objGroupe = [ADSI]'LDAP://CN=Chargés de clients,
OU=Informatique,DC=ps-scripting,DC=com'
PS > $objGroupe.Add('LDAP://CN=Jacques,OU=Utilisateurs,
DC=powershell-scripting,DC=com')
PS > $objGroupe.SetInfo()
```

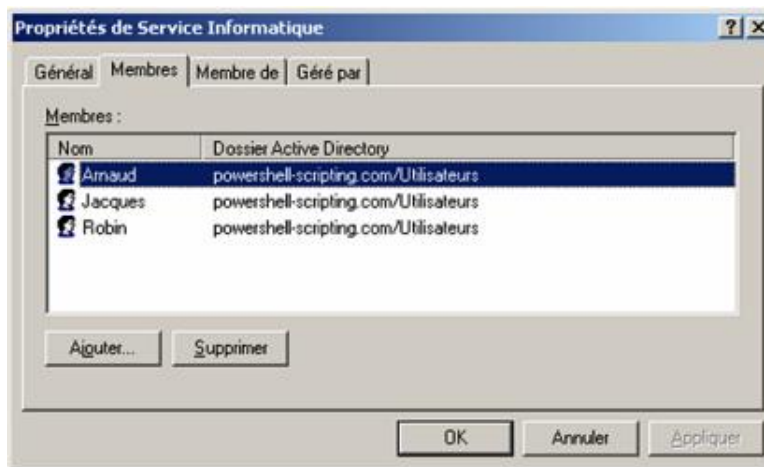
Ajout de plusieurs membres

Pour ajouter plusieurs membres à un groupe, rien de plus facile. Il suffit d'employer la méthode Add comme pour l'ajout d'un utilisateur, et ce autant de fois que d'objets à ajouter au groupe.

Par exemple, pour ajouter les trois utilisateurs Jacques, Robin et Arnaud au groupe « Service Informatique », utilisons le petit morceau de code suivant :

```
PS > $objGroupe = [ADSI]'LDAP://CN=Service Informatique,OU=Informatique,
DC=ps-scripting,DC=com'
PS > $objGroupe.Add('LDAP://CN=Jacques,OU=Utilisateurs,
DC=powershell-scripting,DC=com')
PS > $objGroupe.Add('LDAP://CN=Robin,OU=Utilisateurs,
DC=powershell-scripting,DC=com')
PS > $objGroupe.Add('LDAP://CN=Arnaud,OU=Utilisateurs,
DC=powershell-scripting,DC=com')
PS > $objGroupe.SetInfo()
```

Vérifions le résultat avec la console graphique :



Vérification des membres d'un groupe

c. Renommer un groupe

Tout comme le renommage d'une unité d'organisation, nous allons faire appel à la méthode `MoveHere`.

Dans l'exemple suivant, nous renommons le groupe « Service Informatique » en « Service Informatique de gestion » :

```
PS > $objGroupe = [ADSI]'LDAP://OU=Informatique,DC=ps-scripting,DC=com'
PS > $objGroupe.MoveHere('LDAP://CN=Service Informatique,OU=Informatique,DC=ps-scripting,DC=com', 'CN=Service Informatique de gestion')
```

La première ligne nous connecte à l'UO contenant le groupe. La seconde fait appel à la méthode `MoveHere` où nous indiquons en premier argument le Distinguished Name du groupe (le nom complet LDAP), puis le nouveau nom sous la forme raccourcie « CN=Nouveau nom ». Cette notation s'appelle le RDN pour *Relative Distinguished Name*.



Pour déplacer un groupe, référez-vous à la section Déplacement d'objets dans une unité d'organisation.

d. Supprimer un groupe

Pour supprimer un groupe, nous utiliserons la méthode `Delete`. Il est possible de supprimer un groupe sans avoir à supprimer au préalable son contenu. Veuillez noter également que la suppression d'un groupe ne supprime pas les objets contenus à l'intérieur de celui-ci.

Supprimons, par exemple, le groupe « Chargés de clients » situé dans l'UO « Utilisateurs » :

```
PS > $objGroupe = [ADSI]'LDAP://OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objGroupe.Delete('group', 'CN=Chargés de clients')
```

5. Gestion des utilisateurs

a. Créer un compte utilisateur

La création des comptes utilisateurs fait partie du quotidien des administrateurs système de tous poils. Nous verrons dans le prochain chapitre comment automatiser cette tâche souvent fastidieuse et répétitive. En attendant, nous allons découvrir les principaux attributs à définir lorsque l'on crée un compte.

Afin de mieux visualiser ces attributs et découvrir leurs « vrais » noms, nous entendons par là leurs noms définis dans le Schéma Active Directory, nous allons découvrir les différents champs d'un utilisateur tels qu'ils sont vus lorsque nous utilisons la console **Utilisateurs et Ordinateurs Active Directory**.

name → Edouard

givenName → Prénom : Edouard Initials : → initials

sn → Nom : Bracame

displayName → Nom affiché : Edouard Bracame

description → Description :

physicalDeliveryOfficeName → Bureau :

telephoneNumber → Numéro de téléphone : Autre... → otherTelephone

mail → Adresse de messagerie :

WWWHomePage → Page Web : Autre... → url

OK Annuler Appliquer

Propriétés d'un compte utilisateur - onglet **Général**

userPrincipalName → Nom d'ouverture de session de l'utilisateur : Bracame @powershell-scripting.com

sAMAccountName → Nom d'ouverture de session de l'utilisateur (antérieur à Windows 2000) : PS-SCRIPTING\ Bracame

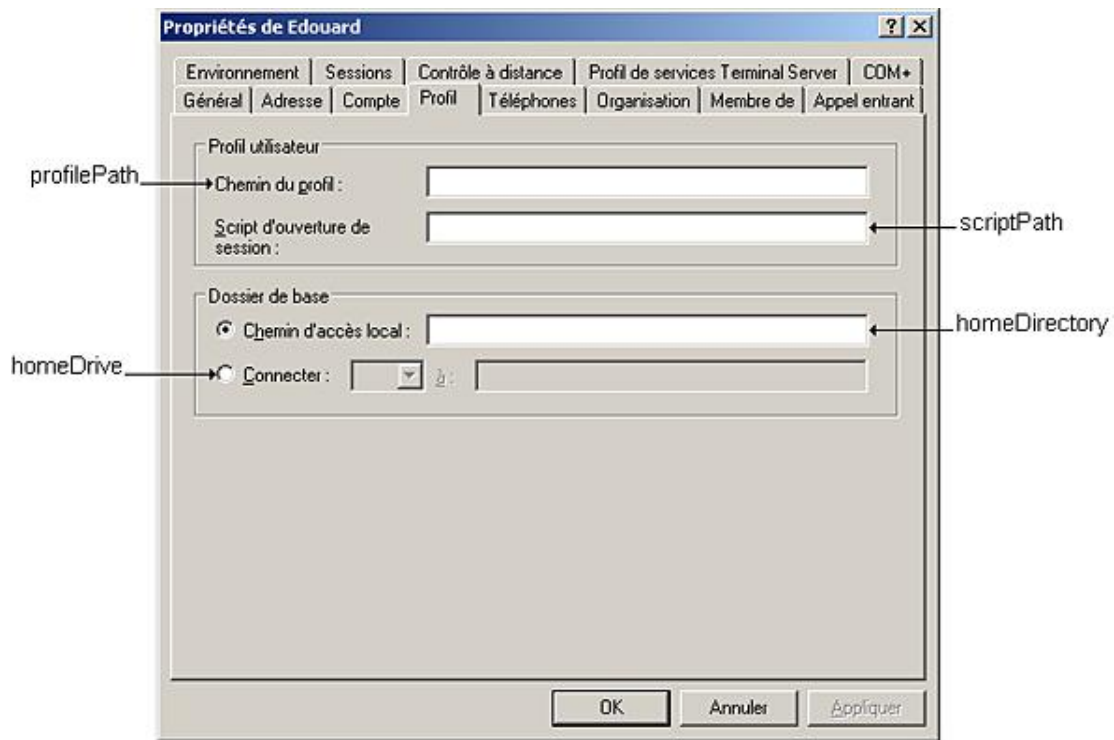
logonHours → Horaires d'accès... Se connecter à... → userWorkstations

userAccountControl → Options de compte : L'utilisateur devra changer le mot de passe L'utilisateur ne peut pas changer de mot de passe Le mot de passe n'expire jamais Enregistrer le mot de passe en utilisant un cryptage réversible

accountExpires → Date d'expiration du compte : Jamais En de : lundi 17 décembre 2007

OK Annuler Appliquer

Propriétés d'un compte utilisateur - onglet **Compte**



Propriétés d'un compte utilisateur - onglet **Profil**

Pour créer un utilisateur, il faut renseigner au minimum les attributs suivants :

- cn,
- SAMAccountName.

Exemple 1 :

Création de l'utilisateur « Edouard Bracame » dans l'UO « Utilisateurs ».

```
PS > $objOU=[ADSI]'LDAP://OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser=$objOU.Create('user', 'cn=Edouard Bracame')
PS > $objUser.Put('SamAccountName', 'Bracame')
PS > $objUser.SetInfo()
```

➤ Bien que ces deux attributs suffisent pour créer un utilisateur, nous vous recommandons fortement d'ajouter l'attribut UserPrincipalName. Celui-ci est arrivé avec Windows 2000/ Active Directory ; il permet de « gagner en sécurité » lors de l'authentification d'un utilisateur en activant l'authentification Kerberos au lieu de NTLM.

Exemple 2 :

Création d'un utilisateur avec UserPrincipalName.

```
PS > $objOU=[ADSI]'LDAP://OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser=$objOU.Create('user', 'cn=Edouard Bracame')
PS > $objUser.Put('SamAccountName', 'Bracame')
PS > $objUser.Put('UserPrincipalName', 'Bracame@powershell-scripting.com')
PS > $objUser.SetInfo()
```

b. Affectation d'un mot de passe

Nous venons de créer un utilisateur mais nous ne lui avons pas encore défini de mot de passe. Pour ce faire, et

comme pour la gestion des utilisateurs locaux, nous allons utiliser la propriété SetPassword.

 Contrairement aux utilisateurs locaux, la définition d'un mot de passe n'est pas obligatoire pour la création d'un compte dans l'annuaire Active Directory. Ceci étant si l'on veut qu'un utilisateur puisse se connecter avec son compte, cette étape est incontournable.

Notez également que pour affecter un mot de passe à un utilisateur, il faut que ce dernier soit déjà créé. En d'autres termes, vous ne pouvez pas en une seule passe créer un utilisateur avec ses différents attributs, et lui affecter un mot de passe. Ceci n'est pas bien gênant mais il faut le savoir !

Vous serez donc obligés d'utiliser SetInfo une première fois pour la création de l'utilisateur, puis une seconde fois, pour valider la définition de son mot de passe.

Exemple 1 :

Affectation d'un mot de passe à un utilisateur déjà existant.

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'  
PS > $objUser.SetPassword('P@ssw0rd')  
PS > $objUser.SetInfo()
```

 Veillez à saisir un mot de passe qui corresponde aux règles de sécurité que vous avez définies dans Active Directory Domain Services, sinon vous obtiendrez un message d'erreur lorsque vous tenterez d'affecter le mot de passe.

Exemple 2 :

Création d'un utilisateur et affectation d'un mot de passe.

```
PS > $objOU=[ADSI]'LDAP://OU=Utilisateurs,DC=ps-scripting,DC=com'  
PS > $objUser=$objOU.Create('user', 'cn=Edouard Bracame')  
PS > $objUser.Put('SamAccountName', 'Bracame')  
PS > $objUser.Put('UserPrincipalName', 'Bracame@powershell-scripting.com')  
PS > $objUser.SetInfo()  
PS > $objUser.SetPassword('P@ssw0rd')  
PS > $objUser.SetInfo()
```

c. Activation d'un compte utilisateur

Lorsque nous créons des comptes utilisateurs avec ADSI dans l'AD DS, ceux-ci sont créés désactivés. Nous allons donc devoir les activer afin que les utilisateurs puissent se servir de leurs comptes.

L'activation d'un compte de l'AD DS s'effectue comme pour un compte utilisateur local. Nous pouvons donc utiliser deux techniques : la première consiste à modifier la propriété AccountDisabled, et la seconde consiste à modifier la valeur de l'attribut UserAccountControl.

Première technique :

Modification d'un compte avec la propriété AccountDisabled.

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'  
PS > $objUser.PSBase.InvokeSet('AccountDisabled', $False)  
PS > $objUser.SetInfo()
```

Pour vérifier par script que la propriété AccountDisabled a bien été prise en compte, faites comme ceci :

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'  
PS > $objUser.PSBase.InvokeGet('AccountDisabled')
```


False

La valeur False nous est retournée, ce qui signifie que le compte n'est pas désactivé ; il est donc actif.

Deuxième technique :

Modification de la propriété UserAccountControl.

Cette technique est très intéressante car elle pourrait nous permettre de faire bien plus que simplement activer ou désactiver un compte. Cette propriété définit également, entre autres, si un utilisateur doit changer son mot de passe à la prochaine connexion ou s'il ne peut pas le changer, etc.

Pour plus d'informations sur ce que l'on peut faire avec cette propriété, reportez-vous au lien suivant : <http://support.microsoft.com/kb/305144>.

Pour activer un compte, il faut effectuer un **ET** logique sur le complément de **2** sur la valeur courante de la propriété UserAccountControl, comme ceci :

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser.userAccountControl[0] =
$objUser.userAccountControl[0] -band (-bnot 2)
PS > $objUser.SetInfo()
```

d. Lecture/définition d'attributs

La première chose à faire pour lire ou définir des attributs d'un utilisateur consiste à se connecter à ce dernier par le biais d'une requête ADSI.

Imaginons que nous voulions définir l'attribut **Description** de l'utilisateur Edouard Bracame, créé dans l'exemple précédent :

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser.Put('Description', 'Compte Utilisateur')
PS > $objUser.SetInfo()
```

À présent, allons lire la valeur de la propriété pour vérifier que celle-ci a bien été prise en compte par le système :

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser.Get('Description')

Compte Utilisateur
```

Nous aurions pu également faire :

```
PS > $objUser.Description

Compte Utilisateur
```

e. Suppression d'attributs

Il est tentant d'imaginer que l'on peut affecter à un attribut une chaîne vide ou une chaîne contenant un espace pour effacer la valeur qu'il contient. Mais la réalité est toute autre : l'affectation d'une chaîne vide n'est pas autorisée, et l'affectation d'une chaîne contenant un espace ne supprime pas l'attribut mais ne fait qu'affecter une nouvelle chaîne. Pour vous en convaincre il suffit de lancer ADSIEdit et de vous positionner sur l'utilisateur que vous manipulez. Ainsi vous pourrez voir facilement les attributs qui ont une valeur et ceux qui n'en ont pas.

Bref, en d'autres termes, nous n'utiliserons pas pour cette tâche la méthode Put mais la méthode PutEx.

Cette méthode est habituellement utilisée avec les attributs multivalués. Elle peut être une alternative à la méthode Add que nous avons utilisé pour ajouter des membres à un groupe.

La méthode PutEx prend trois arguments. Le premier indique ce que l'on souhaite faire :

Valeur	Opération
--------	-----------

1	supprime le contenu complet de l'attribut.
2	remplace la ou les valeurs courantes par la ou les valeurs spécifiées.
3	ajoute des valeurs à la valeur courante.
4	efface la valeur spécifiée uniquement.

Le second argument est le nom de l'attribut sur laquelle porte l'opération. Enfin le troisième argument est la ou les valeurs à définir.

Pour ce qui nous concerne (la suppression), nous utiliserons la valeur 1 en tant que premier argument. Supposons maintenant que nous voulions supprimer la description de l'utilisateur « Edouard Bracame ».

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser.PutEx(1, 'Description', $null)
PS > $objUser.SetInfo()
```

Si vous désirez utiliser la méthode PutEx pour mettre à jour des attributs multivalués ou non, vous devez toujours passer comme valeur un objet de type tableau. Par exemple, si nous voulons définir l'attribut Téléphone d'Edouard Bracame nous devons passer le numéro de téléphone sous la forme d'un tableau contenant un seul élément :

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser.PutEx(2, 'telephoneNumber', @('0556102030'))
PS > $objUser.SetInfo()
```

Et pour spécifier une liste de valeurs, comme pour définir d'autres numéros de téléphone, nous utiliserons toujours un tableau de valeurs, comme dans cet exemple :

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser.PutEx(2, 'otherTelephone', @('123', '456', '789'))
PS > $objUser.SetInfo()
```

Vous avez sûrement remarqué qu'il existait une autre façon de supprimer une valeur d'attribut. Celle-ci consiste à supprimer une valeur parmi un ensemble de valeurs contenues dans un attribut multivalué tel que **otherTelephone** (voir remarque ci-dessus).

Si nous voulons supprimer uniquement la valeur « 456 » nous devons écrire ceci :

```
PS > $objUser=[ADSI]'LDAP://CN=Edouard Bracame,OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objUser.PutEx(4, 'otherTelephone', @('456'))
PS > $objUser.SetInfo()
```

f. Supprimer un utilisateur

La suppression d'utilisateurs est aisée mais gardez bien à l'esprit que c'est une opération dangereuse pour laquelle vous ne pouvez revenir en arrière. Pour cette raison, il peut être prudent avant de supprimer un compte de le désactiver quelque temps auparavant.

Pour supprimer un compte, nous utiliserons la méthode Delete. Nous lui donnerons deux arguments : en premier la classe de l'objet à supprimer, soit « user » et en second, le RDN de l'objet à supprimer sous la forme « CN=nom ».

Par contre, cette fois nous ne nous connecterons pas à l'utilisateur lui-même mais à l'unité organisationnelle qui contient l'utilisateur.

Supprimons pour l'exemple le compte d'Edouard Bracame :

```
PS > $objOU=[ADSI]'LDAP://OU=Utilisateurs,DC=ps-scripting,DC=com'
PS > $objOU.Delete('user', 'CN=Edouard Bracame')
```

Si pour une raison quelconque nous ne savons pas où se situe l'utilisateur que nous voulons détruire, nous pouvons faire appel pour le localiser à un DirectorySearcher. Puis une fois l'objet trouvé, nous nous connecterons à l'OU qui le contient, puis nous passerons son RDN à la méthode Delete.

```
PS > $UserCN = 'Jean Raoul Ducable'
PS > $objDomaine = [ADSI]''
PS > $objRecherche = New-Object `
System.DirectoryServices.DirectorySearcher($objDomaine)
PS > $objRecherche.Filter="(&(objectCategory=user)(cn=$UserCN))"
PS > $Recherche=$objRecherche.FindOne()
PS > $UO = $($($objRecherche.FindOne().path) -replace "CN=$UserCN,", '')
PS > $objOU=[ADSI]"$UO" # Connexion à l'UO
PS > $objOU.Delete('user',"CN=$UserCN") # Suppression de l'utilisateur
```



Pour affecter des groupes à un utilisateur, référez-vous à la section Affecter un ou plusieurs membres à un groupe. Pour déplacer un utilisateur dans une UO, référez-vous à la section Déplacement d'objets dans une unité d'organisation.

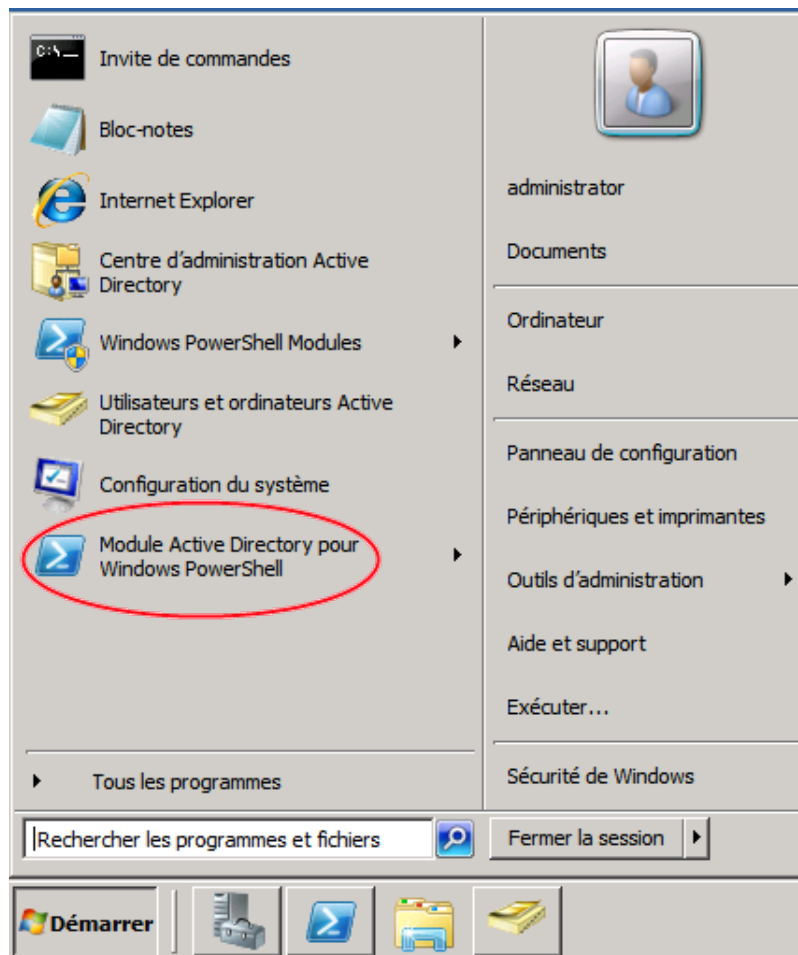
Présentation

Ce module a fait son apparition dans Windows Server 2008 R2. Il est installé en même temps que le rôle « Contrôleur de domaine Active Directory ». Le module Active Directory apporte deux choses : un fournisseur (aussi appelé « Provider ») ainsi qu'un ensemble de commandelettes. Grâce à ce module, il est désormais possible d'administrer en ligne de commandes PowerShell les rôles « *Active Directory Domain Services* (AD DS) » et « *Active Directory Lightweight Domain Services* (AD LDS) ».

AD LDS est le nouveau nom donné à « Active Directory Application Mode (ADAM) ». Ce rôle permet d'utiliser l'annuaire Active Directory en tant qu'annuaire applicatif, contrairement à AD DS qui lui a une vocation système.

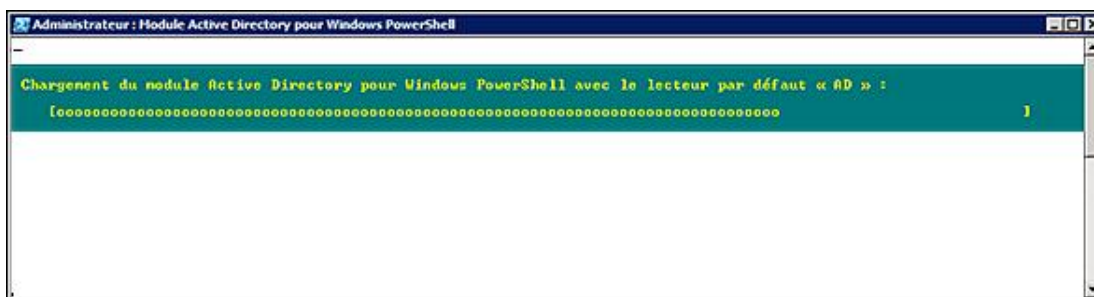
Mise en route du module

Afin de pouvoir utiliser les nouvelles fonctionnalités apportées par ce module il faut tout d'abord le charger. Pour ce faire, il y a plusieurs façons de procéder. La première consiste à passer par le menu **Démarrer - Module Active Directory pour Windows PowerShell**, comme dans l'image suivante.



Menu démarrer de Windows Server 2008 R2

Cela aura pour effet d'ouvrir une console PowerShell et d'importer automatiquement le module Active Directory. L'état de chargement du module s'affiche sous la forme d'une barre de progression comme dans l'image suivante :



Chargement du module Active Directory

L'autre possibilité consiste à ouvrir une console PowerShell de façon traditionnelle, puis de lancer la commande suivante :

```
PS > Import-Module ActiveDirectory
```

En faisant cela, vous verrez également apparaître furtivement la barre de progression indiquant l'état de chargement du module.

Ensuite pour vérifier le bon chargement du module vous pouvez appeler la commande Get-Module, qui vous retournera normalement ceci :

```
PS > Get-Module

ModuleType Name          ExportedCommands
-----
Manifest   ActiveDirectory {Set-ADOrganizationalUnit, Get-ADDomainControl...
```

Vous pouvez également demander la liste des nouvelles commandes apportées par le module en tapant ceci :

```
PS > Get-Command -Module ActiveDirectory
```

Ou vous amuser à les compter, ainsi :

```
PS > Get-Command -Module ActiveDirectory | Measure-Object

Count      : 76
Average    :
Sum        :
Maximum    :
Minimum    :
Property   :
```

Et oui, il y en a soixante-seize ! Soixante-seize commandes de pur bonheur qui vont vous réconcilier avec la ligne de commandes tellement elles sont pratiques...

Le fournisseur Active Directory

Le fournisseur est une des multiples façons de gérer un annuaire Active Directory. Ce n'est clairement pas notre manière de faire préférée, mais elle a le mérite d'exister...

Le fournisseur AD, et les fournisseurs en général, rendent l'exploration des objets similaire à l'exploration d'une arborescence de fichiers et de répertoires. En effet, à la place d'utiliser le jeu de commandes dédié que nous verrons un peu plus loin, pour manipuler les objets d'un fournisseur nous allons devoir utiliser des commandes plus génériques qui sont les suivantes :

Get-PSProvider	Remove-PSDrive	New-Item	Rename-Item	Clear-ItemProperty
Get-PSDrive	Get-ChildItem	Remove-Item	Get-ItemProperty	Get-ACL
New-PSDrive	Get-Item	Move-Item	Set-ItemProperty	Set-ACL

Le point d'entrée du fournisseur ActiveDirectory est le lecteur AD:. Souvenez-vous, pour lister les lecteurs associés aux fournisseurs, il faut utiliser la commandelette Get-PSDrive.

```
PS > Get-PSDrive
```

Name	Used (GB)	Free (GB)	Provider	Root	CurrentLocation
----	-----	-----	-----	----	-----
A			FileSystem	A:\	
AD			ActiveDirectory	//RootDSE/	
Alias			Alias		
C	9,11	117,79	FileSystem	C:\	Users\Administrator
cert			Certificate	\	
D	2,32		FileSystem	D:\	
Env			Environment		
Function			Function		
HKCU			Registry	HKEY_CURRENT_USER	
HKLM			Registry	HKEY_LOCAL_MACHINE	
Variable			Variable		
WSMan			WSMan		

1. Exploration du fournisseur

À présent pour commencer l'exploration, le point de départ se situe au niveau du lecteur AD:.

```
PS C:\Users\Administrator > cd AD :
PS AD:\> Get-ChildItem
```

Name	ObjectClass	DistinguishedName
----	-----	-----
powershell-scripting	domainDNS	DC=powershell-scripting,DC=com
Configuration	configuration	CN=Configuration,DC=powershell-script...
Schema	dMD	CN=Schema,CN=Configuration,DC=pow...
DomainDnsZones	domainDNS	DC=DomainDnsZones,DC=powershell...
ForestDnsZones	domainDNS	DC=ForestDnsZones,DC=powershell...

```
PS AD:\> cd '.\DC=powershell-scripting,DC=com'
```

Name	ObjectClass	DistinguishedName
----	-----	-----
Builtin	builtinDomain	CN=Builtin,DC=powershell...
Computers	container	CN=Computers,DC=powershell...
Domain Controllers	organizationalUnit	OU=Domain Controllers,DC...
ForeignSecurityPrincipals	container	CN=ForeignSecurityPrincipals,...
Infrastructure	infrastructureUpdate	CN=Infrastructure,DC=powershe...
LostAndFound	lostAndFound	CN=LostAndFound,DC=powershell...
Managed Service Accounts	container	CN=Managed Service Accounts,D...
NTDS Quotas	msDS-QuotaContainer	CN=NTDS Quotas,DC=powershell...
Program Data	container	CN=Program Data,DC=powershell...

System	container	CN=System,DC=powershell-scrip...
test	organizationalUnit	OU=test,DC=powershell-scripti...
Users	container	CN=Users,DC=powershell-script...

L'exploration s'effectue comme dans un système de fichiers. Notez que nous avons une « unité d'organisation » nommé « test ». Allons voir ce qu'elle contient :

```
PS AD:\DC=powershell-scripting,DC=com> cd .\OU=test
PS AD:\OU=test,DC=powershell-scripting,DC=com> Get-ChildItem
```

Name	ObjectClass	DistinguishedName
----	-----	-----
Arnaud Petitjean	user	CN=Arnaud Petitjean,OU=test,DC=powershell...

Cette unité d'organisation contient un utilisateur nommé « Arnaud Petitjean ».



Pour vous simplifier la tâche lors de la navigation dans ce fournisseur, utilisez la touche [Tab] (fonction « d'autocomplétion ») car il faut préciser à chaque fois le DistinguishedName des objets à visiter ; ce qui peut vite devenir ennuyeux à taper...

2. Modification d'un objet d'annuaire

Admettons que cet utilisateur ne soit pas à sa place et que nous voulions le déplacer dans l'unité d'organisation Users. Comme pour déplacer un fichier, nous allons utiliser la commande Move-Item :

```
PS > Move-Item '.\CN=Arnaud Petitjean' '\CN=Users,DC=powershell-scripting,DC=com'
```

À présent modifions la propriété EmployeeID d'un objet « user » avec la commande Set-ItemProperty :

```
PS AD:\> Set-ItemProperty `
-path 'AD:\CN=Arnaud Petitjean,CN=Users,DC=powershell-scripting,DC=com' `
-name 'EmployeeID' -value '123456'
```

Vérifions si la valeur a bien été prise en compte :

```
PS AD:\> Get-ItemProperty `
-path 'AD:\CN=Arnaud Petitjean,CN=Users,DC=powershell-scripting,DC=com' `
-name 'EmployeeID'
```

```
PSPath      : ActiveDirectory:///RootDSE/CN=Arnaud Petitjean,CN=Users,DC=powershell-scripting,DC=com
PSParentPath : ActiveDirectory:///RootDSE/CN=Users,DC=powershell-scripting,DC=com
PSChildName  : CN=Arnaud Petitjean
PSDrive      : AD
PSProvider   : ActiveDirectory
EmployeeID   : 123456
```

Comme vous pouvez le remarquer le résultat est un peu verbeux, si nous voulons le restreindre pour n'obtenir que la propriété EmployeeID nous pouvons écrire ceci :

```
PS AD:\> Get-ItemProperty `
-path 'AD:\CN=Arnaud Petitjean,CN=Users,DC=powershell-scripting,DC=com' `
-name 'EmployeeID' | Select-Object EmployeeID -ExpandProperty EmployeeID
```

```
123456
```



Nous faisons ici appel au paramètre **-ExpandProperty** de la commande Select-Object pour nous éviter l'étape qui consiste à extraire une seule propriété. En effet, la commande Select-Object ne récupère qu'une seule propriété de l'objet mais elle nous retourne un objet de type PSObject.

Pour mieux comprendre regardons la valeur que nous aurions obtenue si nous n'avions pas spécifié le paramètre

ExpandProperty :

```
PS AD:\> Get-ItemProperty `
-path 'AD:\CN=Arnaud Petitjean,CN=Users,DC=powershell-scripting,DC=com' `
-name 'EmployeeID' | Select-Object EmployeeID

EmployeeID
-----
123456
```

Pour obtenir la valeur en elle-même, il nous faut une fois encore demander la propriété EmployeeID, comme ceci :

```
PS AD:\> (Get-ItemProperty `
-path 'AD:\CN=Arnaud Petitjean,CN=Users,DC=powershell-scripting,DC=com' `
-name 'EmployeeID' | Select-Object EmployeeID).EmployeeID

123456
```

Si l'on procède de la sorte, l'usage du Select-Object est même superflu car on pourrait simplifier l'écriture de la façon suivante :

```
PS AD:\> (Get-ItemProperty `
-path 'AD:\CN=Arnaud Petitjean,CN=Users,DC=powershell-scripting,DC=com' `
-name 'EmployeeID').EmployeeID

123456
```



Cela prouve une fois de plus qu'il y a de nombreuses façons d'écrire un script et qu'il n'y pas toujours de meilleure façon de faire. La meilleure étant celle qui vous plaît le plus et qui vous permet de reprendre vos scripts quelques mois plus tard sans vous arracher les cheveux pour essayer de comprendre ce que vous avez écrit.

Le jeu de commandes du module Active Directory

Avant d'entrer dans le vif du sujet, nous devons parler de la recherche des objets. En effet, avant d'agir sur un objet, il faut d'abord arriver à l'identifier pour s'y connecter et ainsi pouvoir lui appliquer des commandes.

1. Recherche d'objets

La recherche d'objets avec le jeu de commandes Active Directory peut s'effectuer d'au moins trois façons différentes, qui sont :

- Recherche basée sur la réalisation d'un filtre LDAP (paramètre `LDAPFilter`).
- Recherche basée sur la réalisation d'un filtre générique (paramètre `Filter`).
- Recherche basée sur une identité connue à l'avance (paramètre `Identity`).

La plupart des commandelettes AD possède ces trois paramètres, c'est la raison pour laquelle il est important de les connaître un minimum. Nous verrons que le fait de disposer de ces trois modes de recherche offre un maximum de souplesse.

a. Création d'un filtre LDAP

Si vous avez des requêtes LDAP déjà prêtes à l'emploi ou que vous connaissez parfaitement bien la syntaxe LDAP alors le paramètre **-LDAPFilter** vous sera très utile.

Dans les exemples qui vont suivre, veuillez vous focaliser davantage sur le filtre plutôt que sur la commandelette utilisée.

Exemple :

Obtenir les objets dont la propriété « name » contient la chaîne de caractères « jean ».

-LDAPFilter '(name=*jean*)'

```
PS > Get-ADObject -LDAPFilter '(name=*jean*)'

DistinguishedName : CN=Arnaud Petitjean,CN=Users,DC=powershell-scripting,
                  DC=com
Name              : Arnaud Petitjean
ObjectClass       : user
ObjectGUID        : 7c67b7a3-2415-42c2-8e7a-77a435b3054c
```

Cet exemple basique définit un filtre sur la propriété « name ». Associé à une commande de type `Get-*`, ce filtre nous retournera tous les objets qui contiennent « jean » dans leur nom. Le résultat sera susceptible de contenir tous les types d'objets possibles de l'Active Directory (user, computer, organizationalUnit, etc.) car nous n'avons pas précisé de type en particulier.

Observons à présent un autre exemple :

-LDAPFilter '(&(objectCategory=computer)(name=win*))'

```
PS > Get-ADObject -LDAPFilter '(&(objectCategory=computer)(name=win*))'

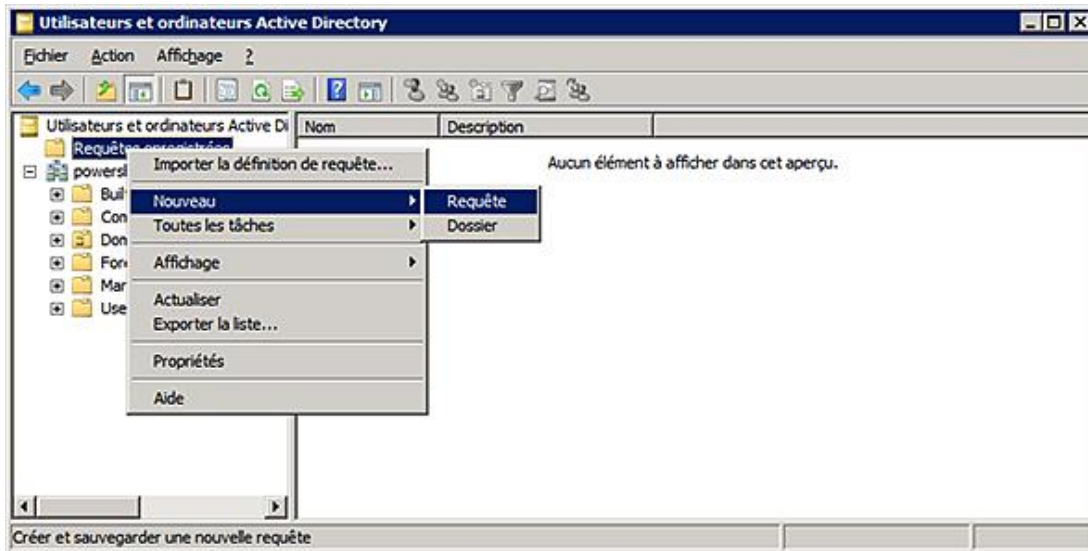
DistinguishedName      Name
-----
CN=WIN7_US_X64,CN=Computers,DC=powershell-scripting,DC=com  WIN7_US_X64
CN=WINXP,CN=Computers,DC=powershell-scripting,DC=com         WINXP
CN=WIN2K8X64,CN=Computers,DC=powershell-scripting,DC=com     WIN2K8X64
CN=WINXPSPV2,CN=Computers,DC=powershell-scripting,DC=com     WINXPSPV2
```

Ce filtre nous retourne tous les objets de la catégorie « computer » dont leur nom commence par la chaîne de caractère « win ».

b. Création avancée d'un filtre LDAP

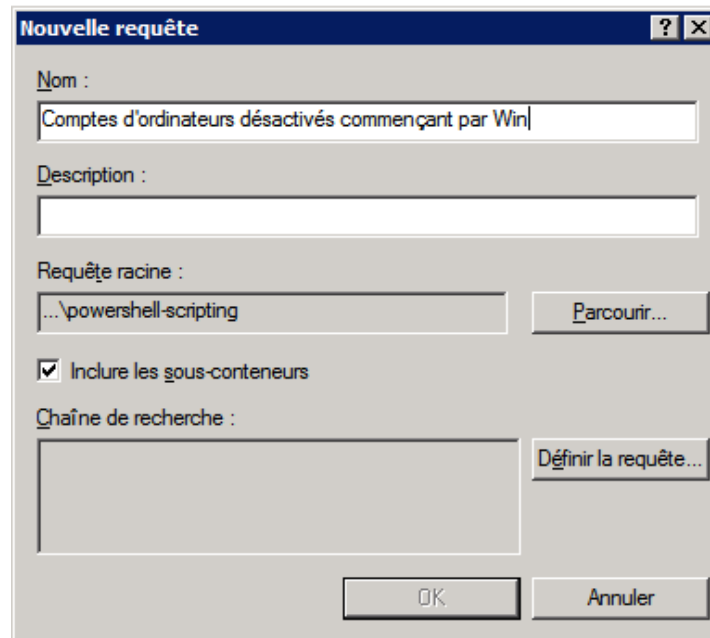
Ce que nous apprécions tout particulièrement avec les filtres de type LDAP c'est qu'il existe un fabuleux outil installé sur tous les contrôleurs de domaine Windows qui va grandement nous faciliter la vie. Il s'agit d'une fonctionnalité assez peu connue mais pourtant fort utile de la console « Utilisateurs et ordinateurs Active Directory ».

Cette console se trouve être un redoutable générateur de requêtes LDAP. Celui-ci est accessible dans l'arborescence « Requêtes enregistrées », voir ci-après :



Éditeur de requêtes LDAP dans la console de gestion Active Directory

Pour ouvrir l'éditeur de requêtes, cliquez droit sur **Requêtes enregistrées** et sur **Nouveau - Requête**. Puis donnez un nom à votre requête, comme ceci :



Éditeur de requêtes LDAP - définition du nom de la requête

Ensuite, cliquez sur le bouton **Définir la requête....** Vous arrivez alors sur une fenêtre qui comprend un certain nombre de requêtes prédéfinies, et vous n'avez plus qu'à renseigner les champs qui vous intéressent. Dans cet exemple, nous avons choisi l'onglet **Ordinateurs**, renseigné le premier champ vide par le début du nom de la recherche, choisi **Commence par** et coché la case **Comptes désactivés**.

Rechercher : Requêtes communes

Utilisateurs Ordinateurs Groupes

Définir les variables de votre requête.

Nom : Commence par win

Description :

☒ Comptes désactivés

OK Annuler

Éditeur de requêtes LDAP - définition de la requête

Validez la requête en cliquant sur le bouton **OK** ; cela nous ramène au menu principal. Maintenant nous allons récolter le fruit de notre travail en copiant dans le Presse-papiers la requête LDAP présente dans le champ **Chaîne de recherche** :

Nouvelle requête

Nom : Comptes d'ordinateurs désactivés commençant par Win

Description :

Requête racine : ... \powershell-scripting Parcourir...

☒ Inclure les sous-conteneurs

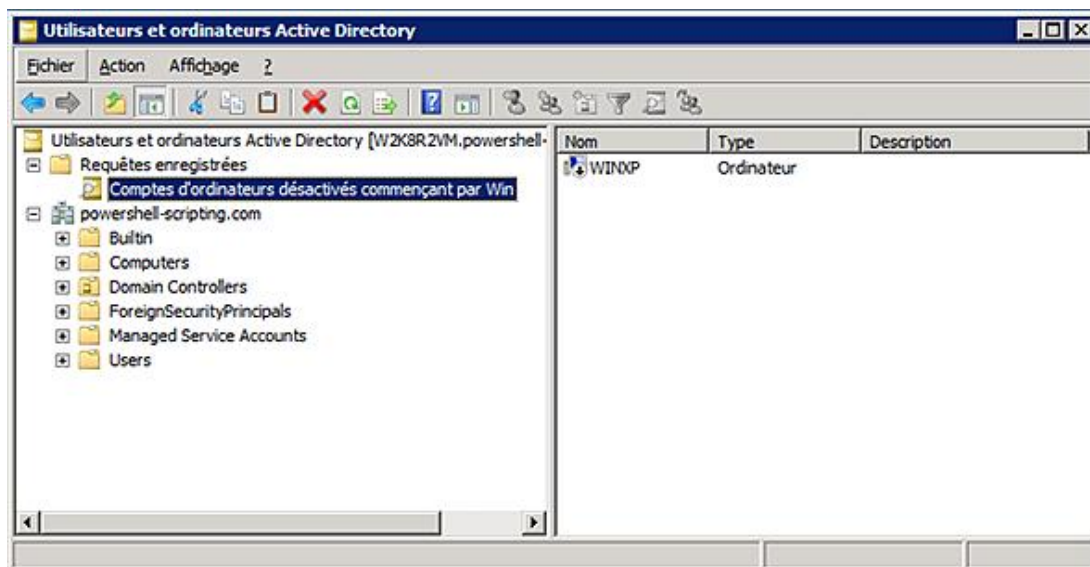
Chaîne de recherche : (&(objectCategory=computer)(name=win*)(objectCategory=computer)(userAccountControl:1.2.840.113556.1.4.803:=2)) Définir la requête...

OK Annuler

Éditeur de requêtes LDAP - récupération de la requête

Et oui, nous avons coché quelques cases, rempli quelques champs et la requête LDAP a été générée automatiquement. N'est-ce pas un outil fantastique !?

Cerise sur le gâteau, lorsque l'on referme cette fenêtre, nous retournons dans l'interface principale de la console **Utilisateurs et ordinateurs Active Directory** et nous pouvons voir visuellement le résultat de notre requête. Si celui-ci n'est pas conforme à nos attentes, nous pouvons la modifier et la tester jusqu'à ce que le résultat nous convienne.



Visualisation du résultat de la requête définie graphiquement

Nous n'avons plus qu'à coller dans notre filtre le résultat issu de notre requête et le tour est joué !

- LDAPFilter

```
'(&(objectCategory=computer)(name=win*)(objectCategory=computer)
(userAccountControl:1.2.840.113556.1.4.803:=2))'
```

```
PS > Get-ADObject -LDAPFilter
'(&(objectCategory=computer)(name=win*)(objectCategory=computer)
(userAccountControl:1.2.840.113556.1.4.803:=2))'

DistinguishedName : CN=WINXP,CN=Computers,DC=powershell-scripting,DC=com
Name              : WINXP
ObjectClass       : computer
ObjectGUID        : 483cc19d-2bb5-4582-b053-c32095f9cfd2
```

c. Création d'un filtre générique

Tout comme le paramètre **-LDAPFilter**, le paramètre **-Filter** prend en entrée une chaîne de caractères. La particularité de cette chaîne qui constituera le filtre est qu'elle peut être très sophistiquée. En effet, sa syntaxe est celle du langage des expressions PowerShell. Elle utilise la notation BNF (*Backus-Naur Form*). Cette notation permet de décrire les règles syntaxiques des langages de programmation ; c'est ce que l'on appelle un métalangage.

Comme il serait trop long de rentrer dans les détails, nous illustrerons les filtres génériques uniquement par le biais d'exemples.



Si vous voulez en apprendre davantage sur la constitution des filtres génériques, rendez-vous dans la rubrique d'aide **about_ActiveDirectory_Filter**.

Exemple 1 :

Obtenir tous les objets de l'Active Directory.

```
PS > Get-ADObject -Filter *
```

Exemple 2 :

Obtenir tous les comptes d'ordinateur de l'AD.

Nous avons spécifié le contenu du filtre entre accolades pour nous conformer à la forme BNF, mais les guillemets fonctionnent également.

```
PS > Get-ADObject -Filter {objectClass -eq 'computer'}
```

name	ObjectClass	ObjectGUID
W2K8R2VM	computer	1f9b3e06-9610-4f9e-8e62-b448f09e40f9
WIN7_US_X64	computer	0558ba9c-a181-44af-a993-1ab56770efd9
WINXP	computer	483cc19d-2bb5-4582-b053-c32095f9cfd2
WIN2K8X64	computer	58052f95-9e16-4dbc-bdcb-bce0e7ee63f1
W2K3R2SRV	computer	46652166-eeb4-4313-b178-3e9c3863653d
WINXPPSV2	computer	2a2bdd5a-6ccd-4cce-8c05-6a00cc01a8c7

Pour des raisons de présentation, nous avons volontairement enlevé la propriété DistinguishedName.

Exemple 3 :

Obtenir la liste des comptes d'ordinateurs commençant par Win.

```
PS > Get-ADObject -Filter {(objectClass -eq 'computer')  
-and (name -like 'Win*')}
```

name	ObjectClass	ObjectGUID
WIN7_US_X64	computer	0558ba9c-a181-44af-a993-1ab56770efd9
WINXP	computer	483cc19d-2bb5-4582-b053-c32095f9cfd2
WIN2K8X64	computer	58052f95-9e16-4dbc-bdcb-bce0e7ee63f1
WINXPPSV2	computer	2a2bdd5a-6ccd-4cce-8c05-6a00cc01a8c7

Exemple 4 :

Obtenir la liste des comptes d'ordinateurs désactivés commençant par Win.

```
PS > $f = {(objectClass -eq 'computer') -and  
           (name -like 'win*') -and (enabled -eq 'False')}  
PS > Get-ADComputer -Filter $f
```

DistinguishedName : CN=WINXP,CN=Computers,DC=powershell-scripting,DC=com
DNSHostName : winxp.powershell-scripting.com
Enabled : False
Name : WINXP
ObjectClass : computer
ObjectGUID : 483cc19d-2bb5-4582-b053-c32095f9cfd2
SamAccountName : WINXP\$
SID : S-1-5-21-1005862844-2131066483-1759542542-1106
UserPrincipalName :

Veuillez noter que nous avons cette fois-ci utilisé commandelette Get-ADComputer à la place de Get-ADObject. Get-ADComputer en étant plus spécifique apporte des propriétés spécifiques aux objets de type Computer, dont la propriété Enabled.

Par conséquent, cela nous permet de simplifier notre requête car nous n'avons pas besoin de spécifier la classe d'objet :

```
PS > $f = {(name -like 'Win*') -and (enabled -eq 'False')}  
PS > Get-ADComputer -Filter $f
```

d. Création d'un filtre basé sur une identité

Les filtres basés sur une identité permettent de s'adresser directement à un objet individuel dont l'identité est déjà connue. C'est très utile car cela évite d'avoir à effectuer une large requête qui renverrait plein de résultats qu'on serait obligé de filtrer. Il s'agit donc d'une manière efficace pour établir une connexion à un objet que l'on connaît. Ou autrement dit à un objet dont l'identité est déjà connue.

Il faut savoir que chaque type d'objet défini dans Active Directory possède des attributs d'identités qui lui sont propres. Par exemple, un objet de type « compte d'ordinateur (ADComputer) » est caractérisé par les attributs suivants :

Attribut d'identification	Description
DistinguishedName	Identifiant LDAP, de la forme : CN=nom,CN=Unité_Organisation,DC=Domaine
ObjectGUID	Identifiant Active Directory, de la forme : c829051e-dea9-4245-b373-d381b9181cc9
SID	Identifiant de sécurité, de la forme : S-1-5-21-1005862844-2131066483-1759542542-1137
SAMAccountName	Identifiant de compte, de la forme : NomDeCompte\$

Mais il possède également d'autres attributs car un objet compte d'ordinateur hérite de la classe ADAccount, qui elle hérite de la classe ADPrincipal, qui à son tour hérite de ADObject, etc.

Voici la hiérarchie de classe définie dans le modèle objet du module Active Directory :

```

ADEntity
  ADRootDSE
  ADObject
    ADFineGrainedPasswordPolicy
    ADOptionalFeature
    ADOrganizationalUnit
    ADPartition
      ADDomain
    ADPrincipal
      ADAccount
        ADComputer
        ADServiceAccount
        ADUser
      ADGroup
    ADDefaultDomainPasswordPolicy
    ADForest
    ADDirectoryServer
      ADDomainController

```

Commençons par observer les attributs d'identité d'un compte d'ordinateur :

```

PS > Get-ADComputer -Identity 'c829051e-dea9-4245-b373-d381b9181cc9'

DistinguishedName : CN=EXCH2K10SRV,CN=Computers,DC=powershell-scripting,
                  DC=com
DNSHostName       : EXCH2K10SRV.powershell-scripting.com
Enabled           : True
Name              : EXCH2K10SRV
ObjectClass       : computer
ObjectGUID        : c829051e-dea9-4245-b373-d381b9181cc9
SamAccountName    : EXCH2K10SRV$
SID               : S-1-5-21-1005862844-2131066483-1759542542-1137
UserPrincipalName :

```

Les attributs constituent les propriétés de l'objet retourné. Nous pouvons remarquer que les attributs DistinguishedName, ObjectGUID, SID et SamAccountName sont bien présents. Les autres étant donc des attributs hérités. Quoi qu'il en soit, chacun de ces attributs non hérité peut être utilisé pour identifier un objet de type ADComputer.

Par exemple, les formes suivantes nous amènent exactement au même résultat que précédemment :

```

Get-ADComputer -Identity 'EXCH2K10SRV'
Get-ADComputer -Identity 'S-1-5-21-1005862844-2131066483-1759542542-1137'
Get-ADComputer -Identity 'CN=EXCH2K10SRV,CN=Computers,DC=powershell-
scripting,DC=com'

```

2. Gestion des utilisateurs

Maintenant que nous savons comment effectuer des recherches basées sur des filtres ou basées sur des identités connues, nous allons entrer dans le vif du sujet.

Pour la gestion des utilisateurs, nous disposons d'un jeu de quelques commandes que nous pouvons obtenir de la manière suivante :

```
PS > Get-Command -Module ActiveDirectory -Name *user*
```

Commande	Description
Get-ADUser	Obtient un ou plusieurs utilisateurs Active Directory.
Set-ADUser	Modifie un utilisateur Active Directory.
New-ADUser	Crée un utilisateur Active Directory.
Remove-ADUser	Supprime un utilisateur Active Directory.
Get-ADUserResultantPasswordPolicy	Obtient la stratégie de mot de passe résultante pour un utilisateur Active Directory.

L'obtention d'un ou de plusieurs utilisateurs s'effectue avec la commande Get-ADUser. Celle-ci possède les paramètres suivants :

Paramètre	Description
Filter <String>	Spécifie une chaîne de requête de type « filtre générique » qui récupère des objets Active Directory.
LDAPFilter <String>	Spécifie une chaîne de requête LDAP utilisée pour filtrer des objets Active Directory.
Identity <ADUser>	Spécifie un objet Active Directory en fournissant l'une de ses valeurs de propriété permettant de l'identifier.
ResultPageSize <Int>	Spécifie le nombre d'objets à inclure dans une page pour une requête. Par défaut, 256 objets par page.
ResultSetSize <Int>	Spécifie le nombre maximal d'objets à retourner pour une requête. La valeur par défaut est \$null. Ce qui correspond à la valeur « tous les objets ».
SearchBase <String>	Spécifie un chemin d'accès Active Directory dans lequel effectuer les recherches.
SearchScope {Base OneLevel Subtree}	Spécifie la portée d'une recherche Active Directory. Une requête <i>Base</i> n'effectue des recherches que dans le chemin d'accès ou l'objet actuel. Une requête <i>OneLevel</i> effectue des recherches dans les enfants directs de ce chemin d'accès ou de cet objet. Une requête <i>Subtree</i> effectue des recherches dans le chemin d'accès ou l'objet actuel et tous les enfants de ce chemin d'accès ou de cet objet. La portée par défaut est <i>Subtree</i> .
Partition <String>	Spécifie le nom unique d'une partition Active Directory.
Properties <String[]>	Spécifie les propriétés de l'objet de sortie à récupérer. Utilisez ce paramètre pour récupérer des propriétés qui ne sont pas incluses dans le jeu par défaut. Spécifiez les propriétés de ce paramètre sous la forme d'une liste de noms séparés par des virgules. Pour afficher tous les attributs définis sur l'objet, spécifiez « * ».
Server <String>	Spécifie l'instance des services Active Directory à laquelle se connecter. Celle-ci peut être de type AD DS, AD LDS ou AD Snapshot. C'est le domaine de l'ordinateur exécutant la requête qui est choisi par défaut.

Credential <PSCredential>	Spécifie les informations d'identification de compte d'utilisateur à utiliser pour effectuer cette tâche.
AuthType {Negotiate Basic}	Spécifie la méthode d'authentification à utiliser. Le type par défaut est Negotiate.

a. Obtenir la liste des utilisateurs

La forme la plus simple pour effectuer une recherche d'utilisateurs à travers tout l'annuaire Active Directory est la suivante :

```
PS > Get-ADUser -Filter *
```

Si nous voulons seulement lister les utilisateurs d'une unité d'organisation, alors nous préférons la ligne de commande suivante :

```
PS > Get-ADUser -Filter * -SearchBase 'OU=Finance,DC=POWERSHELL-SCRIPTING,DC=COM'
```

Pour obtenir une liste facilement interprétable d'utilisateurs, il est utile de formater le résultat sous forme de tableau, comme ci-dessous :

```
PS > Get-ADUser -Filter * | Format-Table GivenName, Surname, Name, Sam*
```

GivenName	Surname	Name	SamAccountName
-----	-----	----	-----
		Administrator	Administrator
		Guest	Guest
		krbtgt	krbtgt
Arnaud	Petitjean	Arnaud Petitjean	PetitjeanA

b. Création d'utilisateurs

En ce qui concerne la création d'utilisateurs, la commande à utiliser est New-ADUser. Celle-ci possède de très nombreux paramètres. Elle possède quasiment autant de paramètres que de propriétés que possède un objet utilisateur (type ADUser).

Pour connaître précisément tous les paramètres de New-ADUser et leur rôle associé, tapez la commande :

```
PS > Help New-ADUser -Full
```

Vu le nombre impressionnant de paramètres que peut prendre cette commande, nous ne mentionnerons que les plus couramment utilisés :

Paramètre	Description
SamAccountName <String>	Spécifie le nom du compte de sécurité SAM (nom de groupe antérieur à Windows 2000).
Name <String>	Spécifie le nom de l'objet.
Surname <String>	Spécifie le nom de famille de l'utilisateur.
DisplayName <String>	Spécifie le nom d'affichage de l'objet.
GivenName <String>	Spécifie le prénom de l'utilisateur.
Description <String>	Spécifie une description de l'objet.
EmailAddress <String>	Spécifie l'adresse de messagerie de l'utilisateur.
Enabled {\$true \$false}	Spécifie si le compte doit être activé. Un compte activé requiert

	un mot de passe.
AccountPassword <SecureString>	Spécifie une nouvelle valeur de mot de passe pour un compte. Cette valeur est stockée sous forme d'une chaîne sécurisée.
CannotChangePassword {\$true \$false}	Spécifie si le mot de passe du compte peut être modifié.
ChangePasswordAtLogon {\$true \$false}	Spécifie si un mot de passe doit être modifié lors de la prochaine tentative d'ouverture de session.
PasswordNeverExpires {\$true \$false}	Spécifie si le mot de passe du compte peut expirer.
PasswordNotRequired {\$true \$false}	Spécifie si le compte requiert un mot de passe. Un mot de passe n'est pas nécessaire pour un nouveau compte.
HomeDirectory <String>	Spécifie le répertoire de base d'un utilisateur.
HomeDrive <String>	Spécifie un lecteur associé au chemin UNC défini par la propriété HomeDirectory.
ProfilePath <String>	Spécifie un chemin d'accès au profil de l'utilisateur.
Path <String>	Spécifie le chemin d'accès X.500 de l'unité d'organisation ou du conteneur où le nouvel objet est créé. Si cette valeur n'est pas précisée, avec AD DS les objets utilisateurs seront rangés dans l'unité d'organisation Users.
PassThru <Switch>	Retourne l'objet nouvellement créé. Par défaut (c'est-à-dire si - PassThru n'est pas spécifié), cette commande ne génère aucune sortie.
ScriptPath <String>	Spécifie un chemin d'accès au script d'ouverture de session de l'utilisateur.
Instance <ADUser>	Spécifie une instance d'un objet utilisateur à utiliser comme modèle pour un nouvel objet utilisateur.
Credential <PSCredential>	Spécifie les informations d'identification de compte d'utilisateur à utiliser pour effectuer cette tâche.

Pour créer un compte utilisateur, il faut au minimum préciser la propriété Name.

Exemple :

Création minimaliste d'un utilisateur.

```
PS > New-ADUser -Name JoeBar
```

Cette ligne de commandes crée un utilisateur dans le conteneur « Users » (car nous n'avons rien précisé) qui apparaîtra sous le nom de « JoeBar ». Il aura également par défaut un attribut SamAccountName qui prendra la valeur « JoeBar ».

Pour vérifier la création et retrouver notre utilisateur, nous pouvons créer un filtre sur son nom :

```
PS > Get-ADUser -Filter {(name -like 'joe*')} | Format-Table Name, Sam*, SID

Name      SamAccountName SID
----      -
JoeBar    JoeBar          S-1-5-21-1005862844-2131066483-1759542542-1138
```

Nous venons d'effectuer une recherche sur tous les comptes dont le nom commence par « joe » et nous en avons profité au passage pour afficher son SID.

Si maintenant nous voulons créer un compte avec davantage d'attributs et que nous voulons le placer dans l'unité d'organisation « Finance », nous pouvons le faire ainsi :

```
PS > New-ADUser -SamAccountName Posichon -Name 'Paul Posichon' `
    -GivenName Paul -Surname Posichon `
    -DisplayName 'Paul Posichon' -description 'Utilisateur terrible !' `
    -HomeDrive 'H:' -HomeDirectory '\\W2K8R2SRV\users' `
    -Path 'OU=Finance,dc=Powershell-scripting,dc=com'
```

Voilà le résultat obtenu lorsque l'on observe les propriétés de notre nouvel utilisateur à travers la console « Utilisateurs et ordinateurs Active Directory » :

The screenshot shows the 'Propriétés de : Paul Posichon' dialog box with the 'Général' tab selected. The fields are as follows:

Field	Value
Prénom :	Paul
Initiales :	
Nom :	Posichon
Nom complet :	Paul Posichon
Description :	Utilisateur terrible !
Bureau :	
Numéro de téléphone :	
Adresse de messagerie :	
Page Web :	

Buttons at the bottom: OK, Annuler, Appliquer, Aide.

Propriétés d'un compte utilisateur, onglet **Général**

Propriétés d'un compte utilisateur, onglet **Profil**

c. Affecter un mot de passe à la création

Nous pouvons affecter un mot de passe à la création d'un compte, avec la commande `New-ADUser`. Si par contre, nous voulons modifier le mot de passe d'un compte existant, alors nous devons utiliser plutôt la commande `Set-ADAccountPassword`.

Quoi qu'il en soit, comme la valeur attendue pour le paramètre **-AccountPassword** est de type chaîne sécurisée (`SecureString`), nous devons effectuer quelques petites manipulations supplémentaires :

```
PS > $passwd = 'Passw0rd123*!'
PS > $passwd = ConvertTo-SecureString $passwd -AsPlainText -Force
PS > New-ADUser -SamAccountName DucableJR -Name 'JR Ducable' `
-AccountPassword $passwd
```

d. Affecter un mot de passe à un compte existant

Si le compte existe déjà, alors dans ce cas il faudra faire comme ceci :

```
PS > Set-ADAccountPassword -Identity DucableJR -NewPassword $passwd -Reset
```

Alors que l'on aurait pu imaginer logiquement que la commande `Set-ADUser` allait faire l'affaire, celle-ci n'autorise pas le changement de mot de passe. Pour ce faire, il faut utiliser la commande `Set-ADAccountPassword`. Celle-ci permet de définir un mot de passe pour un utilisateur, un ordinateur ou un compte de service.

`Set-ADAccountPassword` possède les paramètres suivants :

Paramètre	Description
Identity <ADAccount>	Spécifie un objet Active Directory en fournissant l'une de ses valeurs de propriété permettant de l'identifier.
AuthType {Negotiate Basic}	Spécifie la méthode d'authentification à utiliser. Le type par défaut est Negotiate.

Credential <PSCredential>	Spécifie les informations d'identification de compte d'utilisateur à utiliser pour effectuer cette tâche.
NewPassword <SecureString>	Spécifie une valeur de nouveau mot de passe.
OldPassword <SecureString>	Spécifie la valeur de mot de passe la plus récente.
Partition <String>	Spécifie le nom unique d'une partition Active Directory.
PassThru <Switch>	Retourne l'objet modifié. Par défaut (c'est-à-dire si -PassThru n'est pas spécifié), cette commande ne génère aucune sortie.
Reset <Switch>	Spécifie la réinitialisation du mot de passe sur un compte. Lorsque vous utilisez ce paramètre, vous devez définir le paramètre NewPassword. Il n'est pas obligatoire de spécifier le paramètre OldPassword.
Server <String>	Spécifie l'instance des services Active Directory à laquelle se connecter. Celle-ci peut être de type AD DS, AD LDS ou AD Snapshot. C'est le domaine de l'ordinateur exécutant la requête qui est choisi par défaut.

e. Activer un compte à la création

Pour activer un compte à la création, il est nécessaire de lui affecter un mot de passe. Ce mot de passe doit bien entendu être en adéquation avec la politique de sécurité de votre domaine.

Pour créer un compte qui soit actif, suivez l'exemple ci-après :

```
PS > $passwd = 'Passw0rd123*!'
PS > $passwd = ConvertTo-SecureString $passwd -AsPlainText -Force

PS > New-ADUser -SamAccountName Posichon -Name 'Paul Posichon' `
    -GivenName Paul -Surname Posichon `
    -DisplayName 'Paul Posichon' -description 'Utilisateur terrible !' `
    -HomeDrive 'H:' -HomeDirectory '\\W2K8R2SRV\users' `
    -Path 'OU=Finance,DC=Powershell-scripting,dc=com' `
    -AccountPassword $passwd -Enabled $true
```

f. Activer un compte existant

Pour activer un compte existant, il faut procéder en deux étapes. La première va consister à lui affecter un mot de passe, et la seconde à activer le compte.

```
PS > Set-ADAccountPassword -Identity BracameE -NewPassword $passwd
PS > Set-ADUser -Identity BracameE -Enabled $true
```

g. Lire un ou plusieurs attributs

Pour lire un attribut ou propriété d'un utilisateur, il faut tout d'abord se connecter à l'objet d'annuaire correspondant. Pour ce faire, nous utilisons la commande Get-ADUser avec le paramètre **-Identity**. Ensuite, il n'y a plus qu'à demander la ou les propriétés de notre choix en les passant au paramètre **-Properties**.

Exemple :

```
PS > Get-ADUser -Identity Posichon `
    -Properties Name,WhenCreated,PasswordLastSet

DistinguishedName : CN=Paul Posichon,OU=Finance,DC=powershell-
                    scripting,DC=com
Enabled           : True
GivenName         : Paul
```

```

Name           : Paul Posichon
ObjectClass    : user
ObjectGUID     : 27fe799d-d759-464e-b253-20847a6e7b6a
PasswordLastSet : 31/10/2009 17:22:14
SamAccountName : Posichon
SID            : S-1-5-21-1005862844-2131066483-1759542542-1141
Surname        : Posichon
UserPrincipalName :
WhenCreated    : 31/10/2009 17:22:14

```

Nous nous retrouvons avec tous les autres attributs par défaut. Pour filtrer, nous pouvons formater notre résultat de façon à n'obtenir que les propriétés qui nous intéressent.

```

PS > Get-ADUser -Identity Posichon -Properties * |
FT Name,WhenCreated,PasswordLastSet

```



Pour un gain de place sur la ligne de commandes, notez que nous avons utilisé l'alias « FT » au lieu de la commande Format-Table.

Si cette fois nous ne voulons ne récupérer qu'une seule propriété, par exemple WhenCreated, et la stocker dans une variable nous pouvons écrire ceci :

```

PS > $dateCreation =
(Get-ADUser -Identity Posichon -Properties WhenCreated).WhenCreated

```

Observons le contenu de notre variable \$dateCreation :

```

PS > $dateCreation
samedi 31 octobre 2009 17:22:14

```

Nous pouvons constater que la date obtenue a été formatée différemment que précédemment ; cela montre bien que PowerShell manipule des objets et non du texte. Si le format de date ne vous convient pas, n'hésitez à pas vous reporter à la partie traitant de la manipulation des dates au chapitre Maîtrise du Shell pour la remettre en forme.

h. Obtenir tous les attributs

Pour obtenir toutes les propriétés qu'un objet Utilisateur possède, c'est aussi simple que cela :

```

PS > Get-ADUser -Identity Posichon -Properties *

AccountExpirationDate      :
accountExpires             : 9223372036854775807
AccountLockoutTime         :
AccountNotDelegated        : False
AllowReversiblePasswordEncryption : False
BadLogonCount              : 0
badPasswordTime            : 0
badPwdCount                : 0
CannotChangePassword       : False
CanonicalName              : powershell-scripting.com/Finance/
                             Paul Posichon
Certificates               : {}
City                      :
CN                        : Paul Posichon
codePage                  : 0
Company                   :
Country                   :
countryCode               : 0
Created                   : 31/10/2009 17:22:14
createTimeStamp            : 31/10/2009 17:22:14
Deleted                   :
Department                :
Description                : Utilisateur terrible !
DisplayName                : Paul Posichon
DistinguishedName          : CN=Paul Posichon,OU=Finance,
                             DC=powershell-scripting,DC=com
Division                  :

```

```

DoesNotRequirePreAuth      : False
dScorePropagationData      : {01/01/1601 01:00:00}
EmailAddress               :
EmployeeID                 :
EmployeeNumber             :
Enabled                    : True
Fax                        :
GivenName                  : Paul
HomeDirectory               : \\W2K8R2SRV\users
HomedirRequired            : False
HomeDrive                  : H:
HomePage                   :
HomePhone                  :
Initials                   :
instanceType               : 4
isDeleted                  :
LastBadPasswordAttempt     :
LastKnownParent            :
lastLogoff                 : 0
lastLogon                  : 0
LastLogonDate              :
LockedOut                  : False
logonCount                 : 0
LogonWorkstations         :
Manager                    :
MemberOf                   : {}
MNSLogonAccount            : False
MobilePhone                :
Modified                   : 31/10/2009 17:22:14
modifyTimeStamp             : 31/10/2009 17:22:14
msDS-User-Account-Control-Computed : 0
Name                       : Paul Posichon
nTSecurityDescriptor       : System.DirectoryServices.
                           ActiveDirectorySecurity
ObjectCategory              : CN=Person,CN=Schema,CN=Configuration,
                           DC=powershell-scripting,DC=com
ObjectClass                 : user
ObjectGUID                 : 27fe799d-d759-464e-b253-20847a6e7b6a
objectSid                   : S-1-5-21-1005862844-2131066483-
                           1759542542-1141
Office                     :
OfficePhone                :
Organization                :
OtherName                  :
PasswordExpired             : False
PasswordLastSet            : 31/10/2009 17:22:14
PasswordNeverExpires       : False
PasswordNotRequired        : False
POBox                      :
PostalCode                 :
PrimaryGroup                : CN=Domain Users,CN=Users,
                           DC=powershell-scripting,DC=com
primaryGroupID              : 513
ProfilePath                 :
ProtectedFromAccidentalDeletion : False
pwdLastSet                 : 129014797346100371
SamAccountName              : Posichon
sAMAccountType              : 805306368
ScriptPath                 :
sDRightsEffective           : 15
ServicePrincipalNames       : {}
SID                        : S-1-5-21-1005862844-2131066483-
                           1759542542-1141
SIDHistory                  : {}
SmartcardLogonRequired     : False
sn                          : Posichon
State                      :
StreetAddress               :
Surname                     : Posichon
Title                      :

```

TrustedForDelegation	: False
TrustedToAuthForDelegation	: False
UseDESKeyOnly	: False
userAccountControl	: 512
userCertificate	: {}
UserPrincipalName	:
uSNChanged	: 17991
uSNCreated	: 17987
whenChanged	: 31/10/2009 17:22:14
whenCreated	: 31/10/2009 17:22:14

i. Modifier un attribut

La modification d'un attribut s'effectue avec la commande Set-ADUser. Par exemple, si nous voulons modifier l'attribut SamAccountName, nous utiliserons le paramètre du même nom. Comme ceci :

```
PS > Set-ADUser -Identity Posichon -SamAccountName Cornichon
```

Nous pouvons aussi utiliser le paramètre **-Replace** pour modifier une ou plusieurs propriétés en une seule opération.

Exemple :

Modification de la description, du numéro de téléphone principal et des autres numéros de téléphone

```
PS > Set-ADUser -Identity Posichon -Replace @{
    Description = 'Utilisateur sympathique' ;
    TelephoneNumber = '0110203040';
    OtherTelephone = @('0250403020','0340506070')}
```

Remarquez que nous avons passé une valeur de type tableau à la propriété OtherTelephone ; ainsi nous pouvons lui affecter différentes valeurs en une seule fois.

j. Effacer un attribut

Cette fois c'est à la propriété Clear d'entrer en scène. Supprimons les numéros de téléphone secondaires de l'utilisateur « Posichon », comme ceci :

```
PS > Set-ADUser -Identity Posichon -Clear OtherTelephone
```

k. Supprimer un utilisateur

Nous avons vu que dans le jeu de commandes pour la gestion des utilisateurs existait la commande Remove-ADUser.

Son utilisation est très simple, elle nécessite seulement qu'on passe à son paramètre Identity un objet de type ADUser ; comme dans l'exemple suivant :

```
PS > Remove-ADUser -Identity Posichon

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « Remove » en cours sur la cible « CN=Paul Posichon,OU=Finance,
DC=powershell-scripting,DC=com ».
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S] Suspendre [?]
Aide (la valeur par défaut est « 0 ») :
```

Le fonctionnement par défaut de cette commande est de demander une confirmation. Si vous souhaitez outrepasser la confirmation, vous devez faire comme ceci :

```
PS > Remove-ADUser -Identity Posichon -Confirm:$false
```

3. Gestion des groupes

Pour la gestion des groupes et de leur contenu, nous disposons d'un jeu d'une dizaine de commandes. Nous pouvons les obtenir de la manière suivante :

```
PS > Get-Command -Module ActiveDirectory -Name *group*
```

Commande	Description
Add-ADGroupMember	Ajoute un ou plusieurs membres à un groupe Active Directory.
Add-ADPrincipalGroupMembership	Ajoute un membre à un ou plusieurs groupes Active Directory.
Get-ADAccountAuthorizationGroup	Obtient les groupes de sécurité par l'utilisateur, l'ordinateur ou le jeton de comptes de service spécifié.
Get-ADGroup	Obtient un ou plusieurs groupes Active Directory.
Get-ADGroupMember	Obtient les membres d'un groupe Active Directory.
Get-ADPrincipalGroupMembership	Obtient les groupes Active Directory qui possèdent un utilisateur, un ordinateur, un groupe ou un compte de service spécifié.
New-ADGroup	Crée un groupe Active Directory.
Remove-ADGroup	Supprime un groupe Active Directory.
Remove-ADGroupMember	Supprime un ou plusieurs membres d'un groupe Active Directory.
Remove-ADPrincipalGroupMembership	Supprime un membre d'un ou plusieurs groupes Active Directory.
Set-ADGroup	Modifie un groupe Active Directory.

a. Lister les groupes

Pour obtenir la liste des groupes présents dans l'Active Directory, la commandelette Get-ADGroup est celle qu'il nous faut. Elle possède les mêmes paramètres que la commande Get-ADUser qui sont les suivants :

Paramètre	Description
Filter <String>	Spécifie une chaîne de requête de type « filtre générique » qui récupère des objets Active Directory.
LDAPFilter <String>	Spécifie une chaîne de requête LDAP utilisée pour filtrer des objets Active Directory.
Identity <ADUser>	Spécifie un objet Active Directory en fournissant l'une de ses valeurs de propriété permettant de l'identifier.
ResultPageSize <Int>	Spécifie le nombre d'objets à inclure dans une page pour une requête. 256 objets par page par défaut.
ResultSetSize <Int>	Spécifie le nombre maximal d'objets à retourner pour une requête. La valeur par défaut est \$null. Ce qui correspond à la valeur « tous les objets ».
SearchBase <String>	Spécifie un chemin d'accès Active Directory dans lequel effectuer les recherches.
SearchScope {Base OneLevel}	Spécifie la portée d'une recherche Active Directory. Une requête <i>Base</i>

Subtree}	n'effectue des recherches que dans le chemin d'accès ou l'objet actuel. Une requête <i>OneLevel</i> effectue des recherches dans les enfants directs de ce chemin d'accès ou de cet objet. Une requête <i>Subtree</i> effectue des recherches dans le chemin d'accès ou l'objet actuel et tous les enfants de ce chemin d'accès ou de cet objet. la portée par défaut est <i>Subtree</i> .
Partition <String>	Spécifie le nom unique d'une partition Active Directory.
Properties <String[]>	Spécifie les propriétés de l'objet de sortie à récupérer. Utilisez ce paramètre pour récupérer des propriétés qui ne sont pas incluses dans le jeu par défaut. Spécifiez les propriétés de ce paramètre sous la forme d'une liste de noms séparés par des virgules. Pour afficher tous les attributs définis sur l'objet, spécifiez « * ».
Server <String>	Spécifie l'instance des services Active Directory à laquelle se connecter. Celle-ci peut être de type AD DS, AD LDS ou AD Snapshot. C'est le domaine de l'ordinateur exécutant la requête qui est choisi par défaut.
Credential <PSCredential>]	Spécifie les informations d'identification de compte d'utilisateur à utiliser pour effectuer cette tâche.
AuthType {Negotiate Basic}	Spécifie la méthode d'authentification à utiliser. La valeur par défaut est Negotiate.

Obtenir tous les groupes du domaine

La liste intégrale des groupes peut s'obtenir de la façon suivante :

```
PS > Get-ADGroup -Filter *
```

Pour rendre cette liste un peu plus exploitable visuellement il est intéressant de l'afficher sous forme de table ; auquel cas il nous faut indiquer quelques noms de propriétés :

```
PS > Get-ADGroup -Filter * | Format-Table Name, GroupScope
```

Name	GroupScope
----	-----
Administrators	DomainLocal
Users	DomainLocal
Guests	DomainLocal
Print Operators	DomainLocal
Backup Operators	DomainLocal
Remote Desktop Users	DomainLocal
IIS_IUSRS	DomainLocal
...	
Event Log Readers	DomainLocal
Domain Computers	Global
Domain Controllers	Global
Schema Admins	Universal
Enterprise Admins	Universal
Domain Admins	Global
Domain Users	Global
Domain Guests	Global
Group Policy Creator Owners	Global
...	

Obtenir les groupes d'un certain type

De même, pour n'obtenir que les groupes universels, la création d'un filtre amélioré fait parfaitement l'affaire :

```
PS > Get-ADGroup -Filter {GroupScope -eq 'Universal'} |
Format-Table Name, GroupScope
```

Name	Groupscope
----	-----
Schema Admins	Universal

Enterprise Admins	Universal
Enterprise Read-only Domain Controllers	Universal

Obtenir les groupes d'une unité d'organisation particulière

Pour ne pas récupérer tous les groupes d'un domaine, en spécifiant le paramètre SearchBase nous pouvons restreindre l'étendue de la recherche à une unité d'organisation particulière, comme ci-dessous :

```
PS > Get-ADGroup -Filter * -SearchBase 'OU=finance,DC=powershell-scripting,DC=com'
```

Obtenir un groupe particulier

Et bien sûr, si nous voulons ne récupérer qu'un groupe connu, il est pratique d'utiliser le paramètre Identity, comme ci-après :

```
PS > Get-ADGroup -Identity Administrators

DistinguishedName : CN=Administrators,CN=Builtin,DC=powershell-scripting,DC=com
GroupCategory      : Security
GroupScope         : DomainLocal
Name               : Administrators
ObjectClass        : group
ObjectGUID         : 9cabb43d-ecfa-4cb0-910d-33c9d9490127
SamAccountName     : Administrators
SID                : S-1-5-32-544
```

b. Création de groupes

La création de groupes s'effectue à l'aide de la commande New-ADGroup. Celle-ci comprend les paramètres suivants :

Paramètre	Description
Name <String>	Spécifie le nom de l'objet.
Description <String>	Spécifie une description de l'objet.
DisplayName <String>	Spécifie le nom d'affichage de l'objet.
GroupCategory {Distribution Security}	Spécifie la catégorie du groupe.
GroupScope {DomainLocal Global Universal}	Spécifie l'étendue de groupe du groupe.
SamAccountName <String>	Spécifie le nom du compte de sécurité SAM (nom de groupe antérieur à Windows 2000).
HomePage <String>	Spécifie l'URL de la page d'accueil de l'objet.
Instance <ADGroup>	Spécifie une instance d'un objet groupe à utiliser comme modèle pour un nouvel objet groupe.
ManagedBy <ADPrincipal>	Spécifie l'utilisateur ou le groupe qui gère l'objet.
OtherAttributes <hashtable>	Spécifie les valeurs d'attribut d'objet pour les attributs qui ne sont pas représentés par des paramètres.
PassThru <Switch>	Retourne l'objet nouvellement créé. Par défaut (c'est-à-dire si -PassThru n'est pas spécifié), cette commande ne génère aucune sortie.
Path <String>	Spécifie le chemin d'accès X.500 de l'unité d'organisation ou du

	conteneur où le nouvel objet est créé.
Server <String>	Spécifie l'instance des services Active Directory à laquelle se connecter. Celle-ci peut être de type AD DS, AD LDS ou AD Snapshot. C'est le domaine de l'ordinateur exécutant la requête qui est choisi par défaut.
AuthType {Negotiate Basic}	Spécifie la méthode d'authentification à utiliser. Le type par défaut est Negotiate.
Credential <PSCredential>	Spécifie les informations d'identification de compte d'utilisateur à utiliser pour effectuer cette tâche.

Pour créer un groupe, il faut au minimum lui donner un nom (propriété Name) et une étendue (GroupScope).

Si aucune catégorie (GroupCategory) n'est spécifiée alors par défaut il sera créé en tant que groupe de sécurité. Si vous ne spécifiez pas de SamAccountName alors cette propriété prendra le même nom que la propriété Name du groupe.

Exemple :

Création d'un groupe local de sécurité

```
PS > New-ADGroup -Name UtilisateursVDI -GroupScope DomainLocal `
-GroupCategory Security
```

Création d'un groupe global de distribution

```
PS > New-ADGroup -Name UtilisateursTS -GroupScope Global `
-GroupCategory Distribution
```

Création d'un groupe universel de sécurité dans l'unité d'organisation « Finance » et délégation de la gestion du groupe à un utilisateur :

```
PS > New-ADGroup -Name GrpDelegate1 -GroupScope universal `
-GroupCategory security `
-ManagedBy BracameE -Path 'OU=finance,DC=powershell-scripting,DC=com'
```

c. Énumérer les membres d'un groupe

L'obtention du contenu d'un groupe s'effectue avec la commande Get-ADGroupMember. Voici les paramètres qu'elle possède :

Paramètre	Description
Identity <ADGroup>	Spécifie un objet groupe en fournissant l'une de ses valeurs de propriété permettant de l'identifier.
Recursive <Switch>	Obtient tous les membres du groupe spécifié ainsi que les membres de tout groupe enfant.
Partition <String>	Spécifie le nom unique d'une partition Active Directory.
Server <String>	Spécifie l'instance des services Active Directory à laquelle se connecter. Celle-ci peut être de type AD DS, AD LDS ou AD Snapshot. C'est le domaine de l'ordinateur exécutant la requête qui est choisi par défaut.
Credential <PSCredential>	Spécifie les informations d'identification de compte d'utilisateur à utiliser pour effectuer cette tâche
AuthType {Negotiate Basic}	Spécifie la méthode d'authentification à utiliser. La valeur par

défaut est Negotiate.

Exemple : récupération du contenu du groupe « Administrators »

```
PS > Get-ADGroupMember -Identity Administrators |  
Format-Table Name, ObjectClass
```

Name	ObjectClass
-----	-----
Domain Admins	group
Enterprise Admins	group
Administrator	user

Nous pouvons constater que nous récupérons aussi bien des objets groupes que des objets utilisateurs. Si maintenant nous utilisons le paramètre **-Recursive**, le résultat obtenu sera le développé du contenu de chaque groupe retourné.

Exemple :

```
PS > Get-ADGroupMember -Identity Administrators -Recursive |  
Format-Table Name, ObjectClass
```

Name	ObjectClass
-----	-----
Administrator	user
AdminBracame	user

Dans cet exemple nous récupérons l'utilisateur « AdminBracame » car il est membre du groupe « Domain Admins ».



Si des objets de type « compte d'ordinateur » avaient été présents dans l'un des groupes, « Domain Admins » ou « Entreprise Admins » ils auraient aussi été récupérés. Le paramètre Recursive fait en sorte d'extraire le contenu de tous les groupes.

d. Ajout de membres à un groupe (1 vers 1 ou n vers 1)

L'ajout de membres à un groupe s'effectue avec la commande `Add-ADGroupMember`. Les paramètres disponibles sont les suivants :

Paramètre	Description
Identity <ADGroup>	Spécifie un objet groupe en fournissant l'une de ses valeurs de propriété permettant de l'identifier.
Members <ADPrincipal[]>	Spécifie un jeu d'objets (utilisateur, groupe et ordinateur), dans une liste séparée par des virgules, à ajouter à un groupe.
Partition <String>	Spécifie le nom unique d'une partition Active Directory.
Server <String>	Spécifie l'instance des services Active Directory à laquelle se connecter. Celle-ci peut être de type AD DS, AD LDS ou AD Snapshot. C'est le domaine de l'ordinateur exécutant la requête qui est choisi par défaut.
PassThru <Switch>	Retourne l'objet nouvellement créé. Par défaut (c'est-à-dire si -PassThru n'est pas spécifié), cette commande ne génère aucune sortie.
Credential <PSCredential>	Spécifie les informations d'identification de compte d'utilisateur à utiliser pour effectuer cette tâche
AuthType {Negotiate Basic}	Spécifie la méthode d'authentification à utiliser. La valeur par défaut est Negotiate.

Exemple : ajout de plusieurs utilisateurs à un groupe.

```
PS > Add-ADGroupMember -Identity UtilisateursVDI `
-Members BracameE,DucableJR
```

Vérifions si l'ajout de membres au groupe « UtilisateursVDI » a bien fonctionné :

```
PS > Get-ADGroupMember UtilisateursVDI

distinguishedName : CN=Edouard Bracame,OU=Finance,DC=powershell-scripting,
                  DC=com
name              : Edouard Bracame
objectClass       : user
objectGUID        : eadb78c7-0aa7-4c6b-8496-388f46f3cfea
SamAccountName    : BracameE
SID               : S-1-5-21-1005862844-2131066483-1759542542-1142

distinguishedName : CN=Jean Raoul Ducable,CN=Users,DC=powershell-scripting,
                  DC=com
name              : Jean Raoul Ducable
objectClass       : user
objectGUID        : 1a33625b-14bc-4ec2-b4d9-e53116980350
SamAccountName    : DucableJR
SID               : S-1-5-21-1005862844-2131066483-1759542542-1139
```

Aucun problème, ça a fonctionné ! Remarquez que nous avons omis de préciser le paramètre Identity. Cela ne constitue pas un problème pour PowerShell car ce paramètre est le premier attendu par la commande.

e. Ajout d'un membre à un ou plusieurs groupes (1 vers 1 ou 1 vers n)

L'approche de cette opération est un peu différente de la précédente. Pour l'imager, c'est comme si vous ouvriez les propriétés d'un utilisateur avec l'interface graphique **Utilisateurs et ordinateurs Active Directory** et que vous alliez dans l'onglet **Membre de** pour lui ajouter des groupes.

Par ce moyen vous pouvez mettre un utilisateur dans de nombreux groupes en une seule opération. Pour y parvenir en ligne de commandes, il faut utiliser la commande `Add-ADPrincipalGroupMembership`. Celle-ci accepte les paramètres suivants :

Paramètre	Description
Identity <ADPrincipal>	Spécifie un objet principal Active Directory en fournissant l'une de ses valeurs de propriété permettant de l'identifier.
MemberOf <ADGroup[]>	Spécifie les groupes Active Directory auxquels ajouter un utilisateur, ordinateur ou groupe comme membre.
Partition <String>	Spécifie le nom unique d'une partition Active Directory.
Server <String>	Spécifie l'instance des services Active Directory à laquelle se connecter. Celle-ci peut être de type AD DS, AD LDS ou AD Snapshot. C'est le domaine de l'ordinateur exécutant la requête qui est choisi par défaut.
PassThru <Switch>	Retourne l'objet nouvellement créé. Par défaut (c'est-à-dire si -PassThru n'est pas spécifié), cette commande ne génère aucune sortie.
Credential <PSCredential>	Spécifie les informations d'identification de compte d'utilisateur à utiliser pour effectuer cette tâche.
AuthType {Negotiate Basic}	Spécifie la méthode d'authentification à utiliser. La valeur par défaut est Negotiate.

Cette commande porte un nom générique car elle permet d'ajouter à un groupe non seulement un compte d'utilisateur mais aussi un compte d'ordinateur ou un groupe. Les entités de sécurité Active Directory portent le nom

de « principal » dans le jargon du protocole Kerberos.

Exemple : affectation d'un utilisateur à groupe

```
PS > Add-ADPrincipalGroupMembership AdminBracame `
    -MemberOf 'Enterprise Admins'
```

Exemple : affectation d'un utilisateur à plusieurs groupes

```
PS > Add-ADPrincipalGroupMembership -Identity DucableJR `
    -MemberOf Groupe1, Groupe2, Groupe3
```

f. Suppression d'un ou plusieurs membres d'un groupe

Alors que nous avons utilisé la commande Add-ADGroupMember pour ajouter un ou plusieurs membres à un groupe, cette fois-ci nous allons utiliser son opposée, la commande Remove-ADGroupMember.

Celle-ci possède exactement les mêmes paramètres que sa sœur Add-ADGroupMember. On peut l'utiliser de la façon suivante :

```
PS > Remove-ADGroupMember -Identity UtilisateursVDI `
    -Members DucableJR, BracameE

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « Set » en cours sur la cible « CN=UtilisateursVDI,CN=Users,
DC=powershell-scripting,DC=com ».
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S] Suspendre
[?] Aide (la valeur par défaut est « 0 ») :
```

On peut remarquer que cette commande demande une confirmation pour s'exécuter correctement. Pour outrepasser la confirmation nous pouvons utiliser le paramètre **-Confirm** de la façon suivante :

```
PS > Remove-ADGroupMember -Identity UtilisateursVDI `
    -Members DucableJR, BracameE -Confirm:$false
```

g. Suppression d'un membre d'un ou de plusieurs groupes

La suppression d'un membre d'un ou de plusieurs groupes en une seule opération s'effectue exactement de la même façon que pour l'ajout à un ou plusieurs groupes (ce que nous avons fait avec la commande Add-ADPrincipalGroupMembership). Pour ce faire, il faut utiliser la commande Remove-ADPrincipalGroupMembership. Elle comprend exactement les mêmes paramètres que ceux de sa sœur jumelle.

Exemple : suppression d'un utilisateur d'un groupe

```
PS > Remove-ADPrincipalGroupMembership -Identity AdminBracame `
    -MemberOf 'Enterprise Admins'

Supprimer des membres du groupe
Voulez-vous supprimer tous les membres spécifiés des groupes spécifiés ?
[0] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S] Suspendre
[?] Aide (la valeur par défaut est « 0 ») :
```

Comme à chaque fois que l'on supprime un objet de l'Active Directory, une confirmation est demandée. Pour l'outrepasser, il faut passer la valeur booléenne « false » au paramètre **-Confirm**, comme ci-après :

```
PS > Remove-ADPrincipalGroupMembership -Identity AdminBracame `
    -MemberOf 'Enterprise Admins' -Confirm:$false
```

Exemple : suppression d'un utilisateur de plusieurs groupes

```
PS > Remove-ADPrincipalGroupMembership -Identity DucableJR `
```

h. Supprimer un groupe

La suppression d'un groupe se réalise avec la commande Remove-ADGroup.

Son utilisation est très simple, elle nécessite seulement qu'on passe au paramètre **-Identity** un objet de type ADGroup ; comme dans l'exemple suivant :

```
PS > Remove-ADGroup -Identity GrpDelegue1
```

Confirmer

Êtes-vous sûr de vouloir effectuer cette action ?

Opération « Remove » en cours sur la cible « CN=GrpDelegue1,OU=Finance,DC=powershell-scripting,DC=com ».

[O] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S] Suspendre

[?] Aide (la valeur par défaut est « 0 ») :

Le fonctionnement par défaut de cette commandelette, comme toutes celles de la famille Remove-AD* est de demander une confirmation. Si vous souhaitez outrepasser la confirmation, vous devez faire comme ceci :

```
PS > Remove-ADGroup -Identity GrpDelegue1 -Confirm:$false
```

Trouver les comptes d'ordinateurs périmés dans AD DS

Problématique

Il existe bien souvent un grand nombre de comptes d'ordinateurs inutiles qui sont présents dans le service d'annuaire Active Directory. La raison est simple : dans bien des entreprises il manque une procédure de mise au rebut du matériel, ou si elle existe pour la gestion du matériel physique rien n'est prévu en revanche pour supprimer les comptes d'ordinateurs. Ainsi au bout de quelques années, il n'est pas rare d'avoir 50 % de comptes machine en trop. Par conséquent, il peut devenir difficile pour un administrateur de bien gérer son parc de machines. Pour tenter de remettre un peu d'ordre dans l'AD DS nous allons développer un script qui va se connecter sur un contrôleur de domaine et récupérer la liste des comptes machine. Pour chaque compte machine nous regarderons quelle a été la date de dernier logon ou autrement dit la date de dernière ouverture de session. Car oui, même les comptes machine ouvrent des sessions ! Un compte machine ouvre une session sur un domaine en s'authentifiant auprès d'un contrôleur de domaine, tout comme un utilisateur. À la différence près que les mots de passe des comptes d'ordinateurs s'autogèrent. Un mot de passe est généré aléatoirement la première fois lorsqu'un ordinateur adhère à un domaine, puis il change automatiquement tous les trente jours.

Quelques difficultés à surmonter

Les propriétés des comptes machine sont disponibles auprès des contrôleurs de domaine, mais pas toutes ! En effet, l'information de dernier logon n'est pas répliquée ; elle reste locale à chaque contrôleur de domaine. Pour avoir la dernière information à jour il faut aller interroger tous contrôleurs et ne garder que la donnée la plus à jour. Il en aurait été trop simple autrement ! Cela constitue la première difficulté.

La bonne nouvelle, c'est que dans un domaine Windows Server 2003 ou 2008 nous avons à notre disposition un attribut nommé « LastLogonTimeStamp ». Celui-ci est répliqué entre tous les contrôleurs de domaine, mais (car il y a un mais !) cette réplication n'a lieu que tous les quatorze jours. Pour répondre à notre problématique nous ne sommes pas à deux semaines près ; nous adopterons donc cette technique.

La seconde difficulté provient de la valeur de l'attribut "LastLogonTimeStamp" car cette valeur est un entier codé sur 64 bits et non une simple date du genre « 10/02/2010 ». Nous vous en parlons justement dans le chapitre Maîtrise du Shell, la valeur retournée est le nombre d'intervalles de dix millionnièmes de secondes écoulées depuis le 1^{er} Janvier 1601, rien que cela... Nous devons donc convertir cette valeur en date, mais cela nous l'avons déjà fait.

➤ Pour une prise en charge optimale du « LastLogonTimeStamp », il est recommandé d'avoir des contrôleurs de domaine sous Windows Server 2003 SP1 minimum. De plus, le niveau fonctionnel du domaine doit être en mode natif Windows Server 2003. La fréquence de mise à jour de cet attribut peut être réglée en modifiant l'attribut suivant dans le schéma :

- Object: DC=DomainName
- Attribute: msDS-LogonTimeSyncInterval
- Default value: 14 days

La solution :

```
# Get-MachineAccounts.ps1 - v1.0
[datetime]$date = '01/01/1601'

$ObjDomaine = [ADSI]''
$ObjRecherche = New-Object System.DirectoryServices.DirectorySearcher($ObjDomaine)
$requete = '(&(sAMAccountType=805306369)(name=*))'
$ObjRecherche.Filter=$requete
$comptes = $ObjRecherche.FindAll()

$comptes | select-object @{e={$_.properties.cn};n='Nom commun'},
@{e={$date.AddTicks($_.properties.lastlogontimestamp)};n='Dernière connexion'},
@{e={$_.properties.operatingsystem};n='OS'},
@{e={$_.properties.operatingsystemversion};n='Version'},
@{e={$_.properties.operatingsystemsvicepack};n='Service Pack'}
```


Résultat :

```
PS > ./Get-MachineAccounts.ps1 | Format-Table -AutoSize
```

Nom commun	Dernière connexion	OS	Version	Service Pack
POWERSERVER	29/10/2007 21:02:38	Windows Server 2003	5.2 (3790)	Service Pack 2
PCXPBIS	05/11/2007 23:28:33	Windows XP Professional	5.1 (2600)	Service Pack 2
PCXP1	29/05/2007 22:51:31	Windows XP Professional	5.1 (2600)	Service Pack 2
PCVISTA	03/11/2007 13:37:16	Windows Vista Édition...	6.0 (6000)	
SERVEUR2008	03/11/2007 11:54:49	Windows Server® 2008	6.0 (6001)	Service Pack 1

Quelques explications :

Nous commençons par créer une variable de type date à laquelle nous donnons la valeur du 1^{er} Janvier 1601. Ensuite nous lui ajoutons la valeur de « LastLogonTimeStamp » pour obtenir une date dans un format compréhensible. Nous créons ensuite une requête de recherche dans AD DS. La valeur donnée à « sAMAccountType » correspond à la valeur hexadécimale « 0x30000001 » ; celle-ci représentant un compte machine. Le résultat de la requête est envoyé dans la variable \$comptes. Puis, grâce à **Select-Object** nous créons un nouvel objet à partir du contenu récupéré via le pipe, objet qui contiendra les propriétés « Nom commun », « Dernière connexion », « OS », « Version » et « Service Pack ». Enfin, nous appelons le script en lui demandant d'afficher son contenu sous forme de tableau.

➤ Pour vous montrer qu'il existe un grand nombre d'informations intéressantes en plus du « LastLogonTimeStamp » nous en avons profité pour les retourner en même temps. Celles-ci peuvent se montrer utiles pour, par exemple, faire des inventaires rapides des systèmes d'exploitation installés ainsi que leurs versions de Service Pack associé.

➤ Afin que ce script puisse fonctionner sur un maximum de plates-formes nous avons effectué notre recherche dans l'Active Directory Domain Services en se basant uniquement sur ADSI. Ce script fonctionne aussi avec la version 1 de PowerShell.

Voici ce que pourrait être le script si l'on utilisait les nouvelles commandelettes fournies dans le module Active Directory de Windows Server 2008 R2 :

```
# Get-MachineAccounts.ps1 - v2.0
[datetime]$date = '01/01/1601'

$comptes = Get-ADComputer -Filter * `
    -Properties name,LastLogonTimeStamp,OperatingSystem,
        OperatingSystemVersion,OperatingSystemServicePack

$comptes | Select-Object @{e={$_.Name};n='Nom commun'},
    @{e={$date.AddTicks($_.lastlogontimestamp)};n='Dernière connexion'},
    @{e={$_.operatingsystem};n='OS'},
    @{e={$_.operatingsystemversion};n='Version'},
    @{e={$_.operatingsystemsvicepack};n='Service Pack'}
```

Amélioration possible :

Améliorons notre script afin que nous puissions lui passer un paramètre de péremption du compte. Ainsi grâce à ce paramètre, notre script ne nous retournera que les comptes d'ordinateurs qui ne se sont pas connectés au domaine depuis plus de n jours.

Pour cela, il nous suffit de définir un filtre avec la clause **where-Object**. Si l'expression contenue dans le bloc de script est vraie alors l'objet traité est passé le long du pipeline. En d'autres termes, si l'évaluation est vraie alors c'est que nous avons obtenu un résultat.

Voici le script modifié :

```
# Get-MachineAccounts.ps1 - v1.1
```

```

param ($NonConnecteDepuisNbJours=0)

[datetime]$date = '01/01/1601'

$objDomaine = [ADSI]''
$objRecherche =
    New-Object System.DirectoryServices.DirectorySearcher($objDomaine)
$requete = '(&(sAMAccountType=805306369)(name=*))'
$objRecherche.Filter=$requete
$scomptes = $objRecherche.FindAll()

$scomptes | select-object @{e={$_.properties.cn};n='Nom commun'},
    @{e={$date.AddTicks($_.properties.lastlogontimestamp)};n='Dernière connexion'},
    @{e={$_.properties.operatingsystem};n='OS'},
    @{e={$_.properties.operatingsystemversion};n='Version'},
    @{e={$_.properties.operatingsystemsvicepack};n='Service Pack'} |
    Where-object {(new-timespan $_."Dernière connexion"
$(get-date)).days -ge $NonConnecteDepuisNbJours
}

```

Résultat :

```
PS > ./Get-MachineAccounts.ps1 60 | Format-Table -Autosize
```

Nom commun	Dernière connexion	OS	Version	Service Pack
-----	-----	--	-----	-----
PCXP1	29/05/2007 22:51:31	Windows XP Professional	5.1 (2600)	Service Pack 2

Nous obtenons un résultat ! En effet par rapport à la date du jour, le compte de machine « PCXP1 » ne s'est pas connecté depuis au moins soixante jours. Mais étant donné qu'il existe un delta de mise à jour de quatorze jours, en réalité cela fait peut-être plus de soixante-dix jours que le compte est inactif.



Nous aurions également pu, pour traiter ce problème, nous baser sur la date de dernier changement de mot de passe, mais le delta aurait cette fois été de trente jours au lieu de quatorze. En effet, les mots de passe des comptes machines changent automatiquement tous les trente jours.

Lister les comptes d'utilisateurs inactifs dans AD DS

Problématique

Le constat est assez similaire aux comptes machines : les administrateurs sont toujours au courant dès lors qu'il s'agit de créer un compte utilisateur mais jamais lorsqu'il s'agit de le fermer. Cela entraîne forcément des dérives qui peuvent finir par coûter cher. En effet, le nombre d'objets dans l'annuaire Active Directory ne cesse de croître et des ressources restent monopolisées pour rien (espace disque hébergeant les « homes directory », boîtes aux lettres, etc.).

D'autre part, une mauvaise gestion des comptes utilisateurs peut causer des problèmes de sécurité car nous savons tous qu'un mot de passe qui ne change jamais peut facilement se faire « casser »...

Une seule solution : faire du ménage !

Faire du ménage, l'intention est louable mais sans script point de salut ! Dans l'étude de cas précédente, nous avons focalisé notre attention sur les comptes d'ordinateurs. Et bien sachez que la gestion des comptes utilisateurs s'effectue sur le même principe, et nos explications autour des attributs « LastLogon » et « LastLogonTimeStamp » restent vraies. À savoir que l'attribut « LastLogon » n'est pas répliqué entre les contrôleurs de domaine et que « LastLogonTimeStamp » l'est mais se met à jour que tous les quatorze jours environ. Pour nous simplifier la vie, comme pour les comptes machines, nous nous contenterons d'utiliser « LastLogonTimeStamp » ; nous pourrions donc avoir au pire des cas une différence maximum de quatorze jours par rapport à la réalité du terrain. Mais est-ce vraiment important ?

« LastLogonTimeStamp » n'est disponible qu'à partir d'un domaine Windows Server 2003. Si votre domaine est un domaine Windows 2000 vous n'aurez d'autre choix que d'interroger chaque contrôleur de domaine et prendre l'information « LastLogon » la plus à jour. Ceci étant, avec PowerShell, ce n'est pas une tâche insurmontable ! Remarquez que c'est ce qu'il faudrait faire si vous ne souhaitez pas avoir de delta de date de quatorze jours.

Le script que nous allons développer ensemble va nous permettre de trouver les comptes inactifs depuis un certain nombre de jours. Ensuite libre à vous de l'adapter pour qu'il colle au mieux à votre besoin. Vous pourriez désactiver les comptes ou pourquoi pas, les supprimer (mais ce ne serait pas très prudent !) ou mieux, archiver les données utilisateurs avant toute chose.

```
# Get-userAccounts.ps1 - v1.0
[datetime]$date = '01/01/1601'

$sadsPath = 'LDAP://OU=Utilisateurs,' + ([ADSI]'').distinguishedName
$objDomaine = [ADSI]$sadsPath
$objRecherche = New-Object `
    System.DirectoryServices.DirectorySearcher($objDomaine)
$requeste = '(&(objectCategory=person)(objectClass=user))'
$objRecherche.Filter=$requeste
$comptes = $objRecherche.FindAll()

$comptes |
    select-object @{e={$_.properties.cn};n='Nom commun'},
                 @{e={$_.properties.whencreated};n='Date de création'},
                 @{e={$_.properties.homedrive};n='HD'},
                 @{e={$_.properties.homedirectory};n='HomeDirectory'},
                 @{e={$date.AddTicks($_.properties.lastlogontimestamp)}};
                 n='Dernière connexion'
```

Résultat :

```
./Get-userAccounts.ps1 | Format-Table
```

Nom commun	Date de création	HD	HomeDirectory	Dernière connexion
Joe Bar	12/01/2007 10:52:33	F:	\\Srv3\JoeB	14/01/2007 14:28:05
Edouard Bracame	24/05/2007 19:53:35	M:	\\Srv1\Ed	01/01/1601 00:00:00
Joe L'arsouille	30/06/2007 21:53:58	E:	\\Srv2\Joel	01/01/1601 00:00:00
Jérémie				

Lapurée	01/07/2007 11:57:26 E: \\Srv2\Jérémie	06/10/2007 15:53:38
Jean Raoul		
Ducable	05/08/2007 22:54:33 M: \\Srv1\Jean	01/01/1601 00:00:00
Paul Posichon	30/09/2007 23:11:02 M: \\Srv1\Paul	12/11/2007 18:05:19

Notez que nous en avons profité pour remonter quelques informations qui peuvent être utiles, comme la date de création, le chemin du « home directory » ainsi que sa lettre associé.

Maintenant il ne nous reste plus qu'à gérer un paramètre qui définira le nombre de jours à partir duquel un utilisateur verra son compte désactivé. Par exemple, si l'on définit ce nombre à quatre-vingt dix, nous désactiverons les comptes dont les utilisateurs n'ont pas ouvert de session durant cette période (à la date d'aujourd'hui, soit J-90).

Voici le script modifié :

```
# Get-userAccounts.ps1 - v1.1
param ($NonConnectesDepuisNbJours)

[datetime]$date = '01/01/1601'

$sadsPath = 'LDAP://OU=Utilisateurs,' + ([ADSI]'').distinguishedName
$objDomaine = [ADSI]$sadsPath
$objRecherche = New-Object `
    System.DirectoryServices.DirectorySearcher($objDomaine)
$requete = '(&(objectCategory=person)(objectClass=user))'
$objRecherche.Filter=$requete
$comptes = $objRecherche.FindAll()

if ($NonConnectesDepuisNbJours -eq $null) {
    $comptes |
        select-object @{e={$_.properties.cn};n='Nom commun'},
            @{e={$_.properties.whencreated};n='Date de création'},
            @{e={$_.properties.homedrive};n='HD'},
            @{e={$_.properties.homedirectory};n='Home Directory'},
            @{e={$date.AddTicks($_.properties.lastlogontimestamp)}};
            n='Dernière connexion'}
}
else {
    $comptes |
        select-object @{e={$_.properties.cn};n='Nom commun'},
            @{e={$_.properties.whencreated};n='Date de création'},
            @{e={$_.properties.homedrive};n='HD'},
            @{e={$_.properties.homedirectory};n='Home Directory'},
            @{e={$date.AddTicks($_.properties.lastlogontimestamp)}};
            n='Dernière connexion'} |
        where-object {
            (new-timespan $_.'Dernière connexion' $(get-date)).days -ge
            $NonConnectesDepuisNbJours
        }
}
```

Résultat :

```
./Get-userAccounts.ps1 90 | Format-Table
```

Nom commun	Date de création	HD	HomeDirectory	Dernière connexion
Joe Bar	12/01/2007 10:52:33 F:	\\Srv3\JoeB		14/01/2007 14:28:05
Edouard Bracame	24/05/2007 19:53:35 M:	\\Srv1\Ed		01/01/1601 00:00:00
Joe L'arsouille	30/06/2007 21:53:58 E:	\\Srv2\JoeL		01/01/1601 00:00:00
Jean Raoul				
Ducable	05/08/2007 22:54:33 M:	\\Srv1\Jean		01/01/1601 00:00:00



Une date de dernière connexion au « 01/01/1601 » indique que l'utilisateur n'a jamais ouvert de session. Pour plus d'informations sur l'attribut « LastLogonTimeStamp » veuillez-vous reporter aux documentations Microsoft Technet disponibles ici : <http://www.microsoft.com/technet/scriptcenter/topics/win2003/lastlogon.mspx> et <http://technet2.microsoft.com/windowsserver/en/library/54094485-71f6-4be8-8ebf-faa45bc5db4c1033.mspx?mfr=true>

Voici ce que pourrait être le script si l'on utilisait les nouvelles commandelettes fournies dans le module Active Directory de Windows Server 2008 R2 :

```
# Get-userAccounts.ps1 - v2.1
param ($NonConnectesDepuisNbJours)

[datetime]$date = '01/01/1601'

$comptes = Get-ADUser -Filter * `
    -Properties Name,WhenCreated,HomeDrive,HomeDirectory,LastLogonTimestamp

if ($NonConnectesDepuisNbJours -eq $null) {
    $comptes |
        select-object @{e={$_.Name};n='Nom commun'},
            @{e={$_.whencreated};n='Date de création'},
            @{e={$_.homedrive};n='HD'},
            @{e={$_.homedirectory};n='Home Directory'},
            @{e={$date.AddTicks($_.lastlogontimestamp)};n='Dernière connexion'}
}
else {
    $comptes |
        select-object @{e={$_.Name};n='Nom commun'},
            @{e={$_.whencreated};n='Date de création'},
            @{e={$_.homedrive};n='HD'},
            @{e={$_.homedirectory};n='HomeDirectory'},
            @{e={$date.AddTicks($_.lastlogontimestamp)};n='Dernière connexion'} |
        where-object {
            (new-timespan $_.'Dernière connexion' $(get-date)).days -ge
            $NonConnectesDepuisNbJours
        }
}
```

Changer le mot de passe Administrateur local à distance

Problématique

Lorsque l'on a un important parc de serveurs à administrer et que l'on souhaite garantir à son entreprise un degré de sécurité minimum, il faut s'astreindre à changer régulièrement les mots de passe des comptes ayant des privilèges d'administration. Lorsqu'il s'agit de changer le mot de passe d'un compte du domaine, c'est facile car on ne le change qu'une fois en un seul endroit. Mais lorsqu'il s'agit de changer le mot de passe du compte Administrateur local de chaque serveur membre du domaine, c'est une toute autre paire de manche !

Solution

Nous allons commencer par écrire une petite fonction qui comprendra deux paramètres : un nom de machine et un mot de passe. Cette fonction se connectera à la base de compte locale d'une machine distante et lui changera son mot de passe.

```
Function Set-AdminPassword
{
    param ($machine = $(Throw "Nom d'ordinateur obligatoire !"),
          $password = $(Throw "Mot de passe obligatoire !"))

    $objAdminUser = [ADSI]"WinNT://$machine/Administrateur,user"
    $objAdminUser.SetPassword($password)
    if ($?)
    {
        Write-Host "$machine : Le mot de passe a été changé"
    }
}
```

➤ Ce script fonctionne pour des systèmes d'exploitation français car nous avons indiqué le nom d'un compte utilisateur nommé « Administrateur » dans la requête ADSI. Si vous avez des systèmes d'exploitation américains pensez à changer « Administrateur » en « Administrator ».

À présent, créons un fichier texte qui contiendra la liste des serveurs dont nous voulons changer les mots de passe. Supposons que nous ayons une unité organisationnelle contenant les serveurs membres, nommée « Serveurs membres ». Dans ce cas, il sera très simple de lister les machines et de générer un fichier texte grâce au script suivant :

```
# Get-ServerList.ps1
# Liste les machines contenues dans l'UO Serveurs membres
$sadsPath = 'LDAP://OU=Serveurs membres,' + ([ADSI]'').distinguishedName
$objDomaine = [ADSI]$sadsPath
$objRecherche = New-Object `
    System.DirectoryServices.DirectorySearcher($objDomaine)
$requete = '(&(sAMAccountType=805306369)(name=*))'
$objRecherche.Filter=$requete
$machines = $objRecherche.FindAll()

foreach ($machine in $machines) {
    $machine.Properties.cn >> .\serveurs_membres.txt
}

PS > Get-Content .\serveurs_membres.txt

ServeurFic1
ServeurFic2
ServeurWeb
ServeursSQL
ServeurImpression
```

➤ C'est une bonne pratique de créer une unité d'organisation pour ranger vos serveurs membres car si vous ne le faites pas, ils seront perdus au milieu de tous vos postes clients dans le conteneur Computers. De plus, vous ne pouvez appliquer de stratégies de groupe sur un conteneur « builtin » comme c'est le cas avec l'UO Computers. Par conséquent, nous vous recommandons aussi de déplacer toutes vos machines clientes dans un conteneur que vous aurez créé. Comme par exemple « Ordinateurs du domaine ».

Voici tout ce qu'il y a à faire pour changer le mot de passe administrateur local des machines présentes dans le fichier créé précédemment :

```
PS > $password = 'Azerty123*'
PS > $machines = Get-Content ./serveurs_membres.txt
PS > $machines | Foreach {Set-AdminPassword -Machine $_ -Password
$password}
```

```
ServeurFic1 : Le mot de passe été changé.
ServeurFic2 : Le mot de passe été changé.
ServeurWeb : Le mot de passe été changé.
ServeursQL : Le mot de passe été changé.
ServeurImpression : Le mot de passe été changé.
```

Il est envisageable avec ce script de modifier également le mot de passe Administrateur local de tous vos postes clients. Cependant, vous aurez à faire face à une autre difficulté : les postes peuvent être éteints ou avoir disparus du réseau (d'où la nécessité d'une bonne gestion des comptes machines, voir étude de cas n°1). Nous ne traiterons pas cette problématique dans cette étude de cas, mais n'oubliez pas la fonctionnalité « Wake On Lan » que nous avons pris comme exemple au cours du chapitre .NET, section Manipuler les objets .NET.

Surveiller l'arrivée d'un événement dans le journal

Problématique

Nous avons des troubles du sommeil car des problèmes de sécurité informatique nous empêchent de dormir, parfois des cauchemars nous envahissent durant la nuit et nous réveillent en sursaut. Si cela vous arrive aussi, alors lisez attentivement ce qui va suivre...

Dans l'espoir de détecter une intrusion dans le système ou tout simplement pour savoir si de nouveaux administrateurs du domaine ont été nommés à votre insu, il peut être particulièrement intéressant de surveiller les ajouts de comptes au groupe « Admins du domaine ». Nous souhaitons donc être prévenus par e-mail dès qu'un ajout de ce genre se produit, et ce dans la mesure du possible, en temps réel !

Solution

Écrire un script basé sur les événements WMI qui fonctionnera dans une boucle sans fin. Celui-ci surveillera l'arrivée d'un événement particulier dans le journal de sécurité. Cet événement est l'événement d'ajout à l'intérieur d'un groupe global de sécurité et porte l'ID 632. En effet, lorsqu'une modification d'un groupe de sécurité a lieu et si la stratégie d'audit a été activée alors des événements sont automatiquement consignés dans le journal de sécurité du/des contrôleur(s) de domaine. Ce script devra donc fonctionner de préférence sur un contrôleur de domaine.

Nous devons veiller à ce qu'un mail soit envoyé uniquement en cas de modification du groupe « Admins du domaine » et seulement celui-là sous peine de crouler sous les messages.

Voici le script :

```
# Watch-AdminGroup.ps1
$strComputer = '.'
$query = New-Object System.Management.WqlEventQuery `
    "SELECT * FROM __InstanceCreationEvent
    WITHIN 30
    WHERE TargetInstance ISA 'Win32_NTLogEvent'
    AND TargetInstance.EventCode = '632'"

$scope =
    New-Object System.Management.ManagementScope "\\$strComputer\root\cimv2"
$watcher =
    New-Object System.Management.ManagementEventWatcher $scope,$query
$watcher.Start()

while ($true)
{
    $event=$watcher.WaitForNextEvent()

    if ($($event.TargetInstance.Message) -match 'Admins du domaine')
    {
        # envoi d'un mail
        $expediteur = 'mouchard@ps-scripting.com'
        $destinataire = 'arnaud@ps-scripting.com'
        $serveur = 'mailhost.ps-scripting.com'
        $objet = 'Alerte: Ajout de membre dans le groupe Admins du domaine !'

        $corps = "$($event.TargetInstance.User)`n"
        $corps += "$($event.TargetInstance.TimeWritten)`n"
        $corps += "$($event.TargetInstance.Message)"

        $message =
            new-object System.Net.Mail.MailMessage $expediteur, $destinataire,
            $objet, $corps
        $client = new-object System.Net.Mail.SmtpClient $serveur
        $client.Credentials =
            [System.Net.CredentialCache]::DefaultNetworkCredentials
        $client.Send($message)
    }
}
```

Voici quel serait le contenu d'un message envoyé lors d'une détection d'un événement 632 dans le journal de sécurité contenant le mot clé « Admins du domaine » :


```
PS-SCRIPTING\Administrateur
20071112011950.000000+060
Membre du groupe global de sécurité activée ajouté :

Nom du membre : CN=Joe Bar,OU=Utilisateurs,DC=ps-scripting,DC=com

Id. du membre : PS-SCRIPTING\JoeBar

Nom de compte cible : Admins du domaine

Domaine cible : PS-SCRIPTING

Id. de compte cible : PS-SCRIPTING\Admins du domaine

Utilisateur appelant : administrateur

Domaine appelant : PS-SCRIPTING

Id. de session de l'appelant : (0x0,0x28655)

Privilèges : -
```

Améliorer le résultat

Nous pouvons remarquer le caractère particulier de la date retournée dans la seconde ligne de résultat. Celle-ci est dans un format date WMI qu'il conviendrait de reformater, par exemple comme ceci :

```
PS > $dateWMI = '20071112011950.000000+060'
PS > $OFS = ' '
PS > $date = New-Object system.datetime (
                [string]$dateWMI[0..3],[string]$dateWMI[4..5],
                [string]$dateWMI[6..7],[string]$dateWMI[8..9],
                [string]$dateWMI[10..11],[string]$dateWMI[12..13])
PS > $date

lundi 12 novembre 2007 01:19:50
```

Nous avons créé un objet de type **system.datetime** où nous avons passé au constructeur de cette classe des sous-chaînes correspondant à chaque morceau du contenu de **\$dateWMI**. Cependant, pour que cela fonctionne correctement il y a en réalité deux astuces :

- lorsque nous extrayons une sous-chaîne de la variable **\$dateWMI** avec **\$dateWMI[0..3]** ; cela revient à demander successivement la valeur à l'indice 0, puis celle à l'indice 1 jusqu'à l'indice 3. Cela nous retourne les valeurs suivantes :

```
PS > $dateWMI[0..3]
2
0
0
7
```

- Nous devons ensuite convertir ce résultat en type **string** afin de passer cette valeur au constructeur de l'objet **system.datetime**. Nous le faisons en forçant le résultat en une chaîne, comme ceci : **[string]\$dateWMI[0..3]**

Seulement cette commande ne nous retourne pas le résultat attendu. Essayons-là pour voir :

```
PS > [string]$dateWMI[0..3]
2 0 0 7
```

Nous touchons au but, mais pourquoi diable avons-nous un espace entre chaque caractère ? Et bien tout simplement parce qu'il existe dans PowerShell, la variable spéciale **\$OFS**. Celle-ci entre en jeu dès lors qu'il s'agit de convertir un tableau en une chaîne. Et par défaut cette variable contient le caractère blanc ou espace. Nous allons donc remplacer sa valeur par "" ce qui aura pour effet d'enlever les blancs. Regardons cela de plus près :

```
PS > $OFS = ''
PS > [string]$dateWMI[0..3]
```

Notez que vous pouvez vous amuser à mettre n'importe valeur dans **\$OFS**, par exemple :

```
PS > $OFS = ' '  
PS > [string]$dateWMI[0..3]  
2 0 0 7
```

Améliorer le résultat (bis)

Nous avons découvert il y a peu de temps, que l'on pouvait grandement se simplifier la vie pour ce qui concerne le formatage des dates WMI. En effet, il existe une méthode statique du Framework .Net nommée « ToDateTime » issue de la classe « System.Management.ManagementDateTimeConverter ». Ainsi nous pouvons à présent simplifier notre script en écrivant ceci :

```
PS > $dateWMI = '20071112011950.000000+060'  
PS > [System.Management.ManagementDateTimeConverter]::ToDateTime($dateWMI)  
  
lundi 12 novembre 2007 01:19:50
```

Le résultat retourné étant de type DateTime, nous pourrions le formater à souhait (cf. Chapitre Maîtrise du Shell, section Les dates - Les formats).

Créer des comptes utilisateurs par lot

Problématique

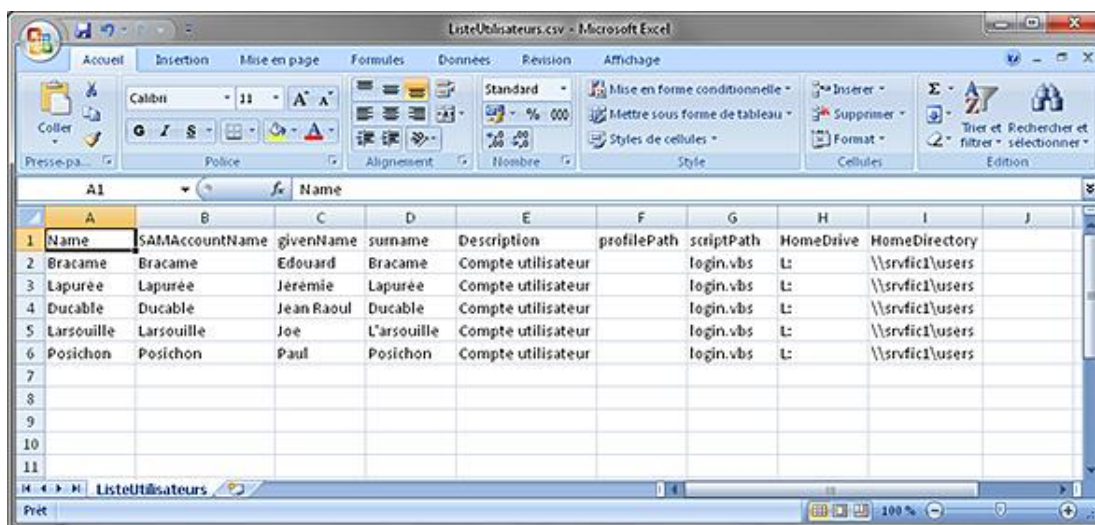
C'est bientôt la rentrée scolaire, et rentrée scolaire rime avec galère ! Comme chaque année nous allons avoir plus de cinq cent comptes utilisateurs à créer pour les étudiants. Mais cette année ne sera plus une année comme les autres, car cette fois nous automatiserons cette tâche ingrate ! Même si la mise au point du script peut être longue, il y a de très fortes chances que nous gagnions du temps par rapport à une opération manuelle. Et quand bien même ce ne serait pas le cas, au moins nous en tirerions une certaine satisfaction personnelle et intellectuelle. D'autre part, nous serons sûrs que tous les comptes seront créés exactement de la même façon, ce qui évitera un grand nombre potentiel d'erreurs manuelles. De plus, nous pourrons réutiliser ce script l'année prochaine...

L'idéal serait qu'à partir d'un fichier texte nous puissions importer les utilisateurs ainsi que tous leurs paramètres associés ; c'est ce que nous allons tenter de faire.

Solution

Pour répondre à cette problématique nous pourrions imaginer la création d'un fichier Excel où chaque ligne contiendrait la description d'un utilisateur, et en particulier les champs suivants :

- Name : nom de l'objet (il s'agit du nom visible dans la console de gestion de l'Active Directory) ;
- SAMAccountName : nom de login ;
- Surname : nom de l'utilisateur ;
- GivenName : prénom de l'utilisateur ;
- Description : description du compte ;
- ProfilePath : chemin à spécifier en cas de profils itinérants ;
- ScriptPath : script de login ;
- HomeDrive : lettre de connexion au home directory ;
- HomeDirectory : chemin réseau (au format UNC) vers un partage sur un serveur.



	A	B	C	D	E	F	G	H	I	J
1	Name	SAMAccountName	givenName	surname	Description	profilePath	scriptPath	HomeDrive	HomeDirectory	
2	Bracame	Bracame	Edouard	Bracame	Compte utilisateur		login.vbs	L:	\\srvfic1\users	
3	Lapurée	Lapurée	Jérémie	Lapurée	Compte utilisateur		login.vbs	L:	\\srvfic1\users	
4	Ducable	Ducable	Jean Raoul	Ducable	Compte utilisateur		login.vbs	L:	\\srvfic1\users	
5	Larsouille	Larsouille	Joe	L'arsouille	Compte utilisateur		login.vbs	L:	\\srvfic1\users	
6	Posichon	Posichon	Paul	Posichon	Compte utilisateur		login.vbs	L:	\\srvfic1\users	
7										
8										
9										
10										
11										

Création d'utilisateurs par lot - Fichier csv

Nous pourrions ensuite enregistrer ce fichier au format CSV (*Comma Separated Values*) pour en obtenir un fichier texte dont les champs seraient séparés par des points-virgules, Excel ne sachant pas utiliser la virgule comme séparateur. Un tel fichier pourrait ressembler à celui-ci :

```
Name;SAMAccountName;givenName;surname;Description;profilePath;scriptPath;
HomeDrive;HomeDirectory
Bracame;Bracame;Edouard;Bracame;Compte utilisateur;;login.vbs;L:;\\srvfic1\users
Lapurée;Lapurée;Jérémie;Lapurée;Compte utilisateur;;login.vbs;L:;\\srvfic1\users
Ducable;Ducable;Jean Raoul;Ducable;Compte utilisateur;;login.vbs;L:;\\srvfic1\users
Larsouille;Larsouille;Joe;L'arsouille;Compte utilisateur;;login.vbs;L:;\\srvfic1\users
Posichon;Posichon;Paul;Posichon;Compte utilisateur;;login.vbs;L:;\\srvfic1\users
```

Solution PowerShell v1

Ensuite il ne nous resterait plus qu'à passer chaque valeur de chaque ligne à un script qui se chargerait de tout. Pour importer un fichier CSV, nous disposons de la commande **Import-CSV**, seulement le problème est que celle-ci ne supporte que la virgule comme séparateur de champs. Par conséquent, nous allons devoir opérer une petite transformation sur notre fichier CSV. Cela se fait très simplement :

```
PS > (Get-Content listeUtilisateurs.csv) -replace ',' ,',' >
Liste-SepVirgule.csv
```

Ce qui nous donne le résultat suivant :

```
Name,SAMAccountName,givenName,surname,Description,profilePath,scriptPath,
HomeDrive,HomeDirectory
Bracame,Bracame,Edouard,Bracame,Compte utilisateur,,login.vbs,L:;\\srvfic1\users
Lapurée,Lapurée,Jérémie,Lapurée,Compte utilisateur,,login.vbs,L:;\\srvfic1\users
Ducable,Ducable,Jean Raoul,Ducable,Compte utilisateur,,login.vbs,L:;\\srvfic1\users
Larsouille,Larsouille,Joe,L'arsouille,Compte utilisateur,,login.vbs,L:;\\srvfic1\users
Posichon,Posichon,Paul,Posichon,Compte utilisateur,,login.vbs,L:;\\srvfic1\users
```

Il ne reste plus qu'à faire l'import du fichier, par exemple avec le script suivant :

```
# Create-User.ps1
param (
    [string]$fichier=$(throw 'Vous devez fournir un fichier dont le
                                séparateur est la virgule !'),
    [string]$OU='CN=Users'
)

# Récupération du domaine courant sous la forme @ps-scripting.com
$domaineSMTP = "@$env:USERDNSDOMAIN"

# Récupération du domaine courant sous la forme ps-scripting.com
$domaineLDAP=$env:USERDNSDOMAIN

# Transformation de la chaîne DNS en chaîne de connexion LDAP
# mondomaine.ps-scripting.com -> mondomaine,DC=ps-scripting,DC=com
# ou ps-scripting.com -> ps-scripting,DC=com
$domaineLDAP = $domaineLDAP.replace('.', ',DC=')

# ajout de DC= en tête de chaîne
# ps-scripting,DC=com -> DC=ps-scripting,DC=com
$domaineLDAP="DC=$domaineLDAP"

$objOU=[ADSI]"LDAP://$OU,$domaineLDAP"

$users = import-csv $fichier
foreach ($user in $users)
{
    # Nom de l'objet Active Directory
    $objUser=$objOU.Create('user', "CN=$(($user.Name))")

    # Définition du SAMAccountName
    $objUser.put('sAMAccountName', $user.SamAccountName)

    # Définition de l'USPN pour l'ouverture de session sécurisée
    # Exemple : Posichon@ps-scripting.com
    $objUser.put('userprincipalName', $user.SamAccountName + $domaineSMTP)

    # Prénom
```

```

$objUser.put('GivenName', $user.givenName)

# Nom
$objUser.put('sn', $user.surname)

# Description
$objUser.put('description', $user.description)

# Chemin du profil Windows
if ($user.profilePath -ne '')
{
    $objUser.put('profilePath', $user.profilePath)
}

# Logon Script
$objUser.put('scriptPath', $user.scriptPath)

# Home Drive
$objUser.put('homeDrive', $user.homeDrive)

# Home Path
$objUser.put('homeDirectory', $user.homeDirectory + '\\' +
$user.SamAccountName)

# Création de la home directory
new-item -path $($user.homeDirectory + '\\' + $user.SamAccountName)
-type directory

$objUser.SetInfo()
Write-Host "Utilisateur $($user.SamAccountName) créé."
}

```

Utilisation :

```
./Create-User.ps1 -fichier Liste-SepVirgule.csv -OU 'OU=Utilisateurs'
```

Si nous ne spécifions pas le paramètre **-OU**, le script créera, par défaut, les utilisateurs dans le conteneur « Users ».

Bien que ce script puisse être grandement amélioré, il vous donne un point de départ intéressant pour vous lancer dans la création massive d'utilisateurs.

Solution PowerShell v2/Module Active Directory Server 2008 R2

Avec PowerShell v2 couplé aux commandelettes apportées par le module Active Directory, vous allez voir avec quelle facilité nous répondrons à cette problématique par rapport à la solution précédente...

Le fait qu'Excel ne sache pas enregistrer les fichiers CSV avec un séparateur autre que le point-virgule ne nous dérange pas plus que cela car la commandelette Import-CSV s'est doté du paramètre Delimiter qui permet de spécifier un délimiteur autre que la virgule.

De plus, avec le module Active Directory nous disposons de la commande New-ADUser. Celle-ci lorsqu'on lui passe en entrée de pipeline un objet avec les bonnes propriétés crée directement l'utilisateur. Cela raccourcira considérablement notre script. Voir ci-après :

```
PS > (Import-Csv .\ListeUtilisateurs.csv -Delimiter ';') | New-ADUser
```

Si l'on souhaite créer les utilisateurs dans une unité d'organisation particulière, il suffit de préciser le paramètre Path, comme dans l'exemple suivant :

```
PS > (Import-Csv .\ListeUtilisateurs.csv -Delimiter ';') |
New-ADUser -Path 'OU=Finance,DC=Powershell-scripting,DC=com'
```

Vérifier la version logicielle d'une application à distance

Problématique

Un matin, votre supérieur vient vous voir et vous demande de dresser un inventaire sur les différentes versions d'un applicatif déployé sur un important parc de machines. C'est normalement à ce moment là que vous réalisez que vous ne disposez ni d'un outil de rapports applicatifs, ni d'un inventaire de déploiement logiciel à jour.

Solution

Pour répondre à cette problématique nous pourrions imaginer la création d'un script qui va, pour chaque compte d'ordinateur, vérifier dans la base de registre la version logicielle du lecteur (Windows Media Player par exemple). Pour cette étude de cas, nous allons réutiliser une partie des scripts précédents pour obtenir la liste des postes clients enregistrés dans l'Active Directory.

-
- Pour que l'accès à la base de registre à distance fonctionne sur une machine Windows 7, vérifiez que le service « Registre à distance (*RemoteRegistry*) » soit bien démarré.
-

La solution :

```
# Get-MPVersion.ps1
# --- Déclaration des fonctions ---
# --- Fonction pour savoir si une machine est en ligne ---
function Ping-Host
{
    param ($HostName)
    $tmp = ping.exe $HostName -n 1
    if ($LASTEXITCODE -ne 0)
    {
        $false
    }
    else
    {
        $true
    }
}

# --- Fonction pour connaître la version de l'application ---
function Get-Key
{
    param ([String]$HostName)
    #Accès à la base de registre
    $Cle = "SOFTWARE\Microsoft\MediaPlayer\PlayerUpgrade"
    $Type = [Microsoft.Win32.RegistryHive]::LocalMachine
    $Cle_Reg = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey($Type,$HostName)
    $Cle_Reg = $Cle_Reg.OpenSubKey($Cle)
    if ($Cle_Reg -eq $null)
    {
        $Version="Non Installé"
    }
    else
    {
        $Liste = $Cle_Reg.GetValueNames()
        foreach($Nom_Valeur in $Liste)
        {
            if($Nom_Valeur -eq 'PlayerVersion')
            {
                $Version = $Cle_Reg.GetValue('PlayerVersion')
            }
        }
    }
    $Version
}

# --- Début du script ---
```

```
# - Requete Active Directory -
# Création d'un objet objDomaine qui se connecte à un controleur
# de domaine.
# Le fait de ne rien spécifier entre les cotes spécifie que le script va
# interroger le premier controleur de domaine qu'il va trouver.
$objDomaine = [ADSI]''

#Création d'un objet de type DirectorySearcher pour rechercher
#dans l'annuaire.
$objRecherche = New-Object System.DirectoryServices.DirectorySearcher
($objDomaine)

#Création de la requête et application de celle-ci.
#On filtre sur les machines Windows.
$requete = '(&(objectCategory=computer)(name=*)(operatingSystem=Windows*))'
$objRecherche.Filter = $requete

#Methode de recherche des comptes dans l'AD
$listeMachines = $objRecherche.Findall()

[PSObject[]]$table = $null
# - Boucle sur toute la liste des serveurs retournés -
foreach ($machine in $listeMachines)
{
    $machine = $machine.Properties["name"]
    if (ping-host $machine) #test du ping
    {
        $version = Get-Key $machine
        if($version -ne $null)
        {
            $resultTemp = New-Object PSObject
            $resultTemp |
                Add-Member -MemberType NoteProperty -name Machine -value $machine
            $resultTemp |
                Add-Member -MemberType NoteProperty -name Version -value $version
            [PSObject[]]$table += [PSObject]$resultTemp
        }
    }
    else
    {
        $resultTemp = New-Object PSObject
        $resultTemp |
            Add-Member -memberType NoteProperty -name Machine -value $machine
        $resultTemp |
            Add-Member -memberType NoteProperty -name Version -value 'Hors ligne'
        [PSObject[]]$table += [PSObject]$resultTemp
    }
}
$table
```

Utilisation :

./Get-MPVersion

Résultat :

Machine	Version
-----	-----
{W2K8R2VM}	Non Installé
{WIN7_US_X64}	12,0,7600,16415
{WINXP}	Hors ligne
{WIN2K8X64}	Non Installé
{W2K3R2SRV}	10,0,0,3997
{WINXPPSV2}	9,0,0,4503
{EXCH2K10SRV}	Non Installé
{WINXPPSV1}	9,0,0,4503

Quelques explications :

Dans un premier temps, notre script intègre une fonction capable de retourner la valeur de la clé PlayerVersion dans la base de registre d'une machine distante. Pour ce faire, on utilise la méthode OpenRemoteBaseKey du framework (l'accès à une base de registre distant ne s'effectue pas via le provider local de PowerShell).

```
function Get-Key
{
    param ([String]$HostName)
    #Accès à la base de registre
    $Cle = "SOFTWARE\Microsoft\MediaPlayer\PlayerUpgrade"
    $Type = [Microsoft.Win32.RegistryHive]::LocalMachine
    $Cle_Reg = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey($type,$HostName)
    $Cle_Reg = $Cle_Reg.OpenSubKey($Cle)
    if ($Cle_Reg -eq $null)
    {
        $Version="Non Installé"
    }
    else
    {
        $Liste = $Cle_Reg.GetValueNames()
        foreach($Nom_Valeur in $Liste)
        {
            if($Nom_Valeur -eq 'PlayerVersion')
            {
                $Version = $Cle_Reg.GetValue('PlayerVersion')
            }
        }
    }
    $Version
}
```

 Avec PowerShell, l'utilisation de la méthode OpenRemoteBaseKey sur une machine distante est soumise à la condition suivante : appartenir un domaine Windows. En effet, si cette condition n'est pas remplie, alors le second argument passé à cette méthode ne devra pas être le nom d'ordinateur de la machine, mais son adresse IP.

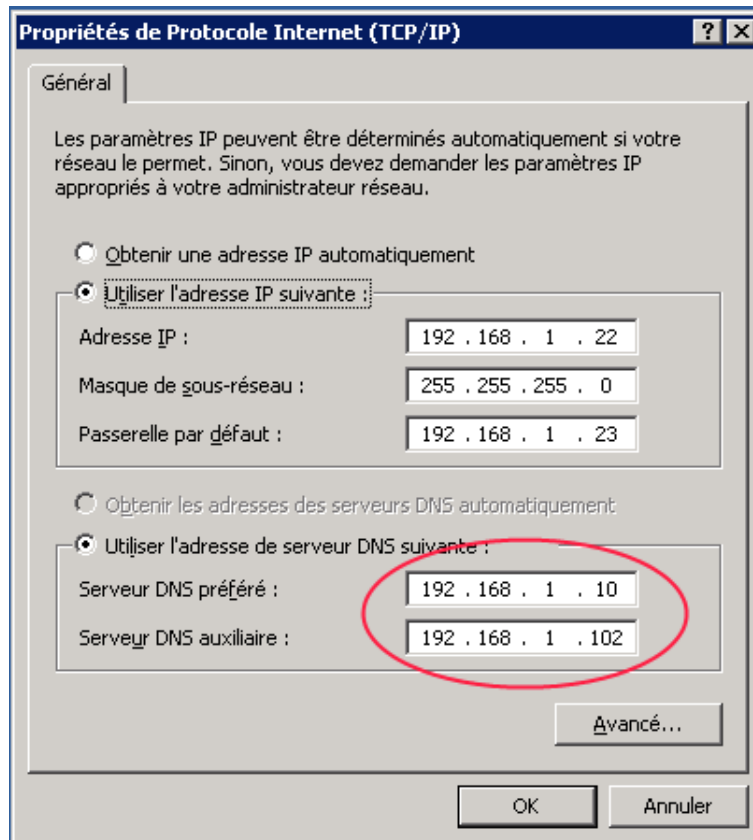
Ensuite il suffit de parcourir l'annuaire afin d'appliquer la fonction Get-Key sur l'ensemble des postes clients.

Mise à jour de la configuration réseau d'un ensemble de machines

Problématique

Nous venons d'effectuer une migration de nos serveurs DNS et nous avons dû installer de nouvelles machines à la place des anciennes. Par conséquent pour terminer cette migration, il va falloir mettre à jour la configuration réseau de tous nos serveurs pour prendre en compte ce changement. Nos serveurs ayant une configuration réseau statique, un script sera le bienvenu pour automatiser cette modification de configuration. Cela nous évitera de modifier manuellement ce paramétrage sur chacun de nos serveurs. Dans cette étude de cas, nous supposons que le pare-feu réseau et ceux des machines distantes laissent passer le ping ainsi que les requêtes WMI.

Voici dans l'interface graphique les champs qu'il nous faut mettre à jour :



Paramètres réseaux à changer

Solution

Utiliser WMI pour interroger et modifier à distance le paramétrage de la configuration réseau.

Dans un premier temps, nous allons nous attacher à faire un script qui récupère la configuration DNS d'une machine et qui retourne un objet personnalisé. Nous le nommerons **Get-DNSConfiguration.ps1**. Puis dans un second temps, nous ferons celui qui permet de modifier la configuration DNS, nous l'appellerons **Set-DNSConfiguration.ps1**. Tous deux prendront en entrée un paramètre pour indiquer le nom de la machine sur laquelle agir.

Get-DNSConfiguration.ps1

```
# Get-DNSConfiguration - v.1
#requires -version 2

param ($ComputerName)

if (Test-Connection $ComputerName -Quiet -Count 1)
{
    # récupération des interfaces réseaux actives
    $cnxActives = Get-WmiObject Win32_NetworkAdapter -Computer $ComputerName |
```

```

        Where {$_.NetConnectionStatus -eq 2}

    # récupération de la configuration des interfaces réseaux actives
    $confActive =
Get-WmiObject Win32_NetworkAdapterConfiguration -Comp $ComputerName |
    Where { ($_.index -eq $(ScnxActives.index)) }
    $result = $confActive.DNSServerSearchOrder
    $Ping = 'OK'
}
else
{
    $result = $null
    $Ping = 'NOK'
}

$objResult = New-Object PSObject
$objResult | add-Member -memberType NoteProperty -name HostName -value
$ComputerName
$objResult | add-Member -memberType NoteProperty -name Ping -value $ping
$objResult | add-Member -memberType NoteProperty -name DNS -value $result
$objResult

```

Faisons un essai :

```

PS > .\Get-DNSConfiguration.ps1 W2K3R2SRV

```

HostName	Ping	DNS
-----	----	---
W2K3R2SRV	OK	{192.168.1.23, 192.168.1.24}

Parfait, le script fonctionne !

Quelques explications

Le mot réservé Requires commence par indiquer que ce script ne fonctionne qu'avec la version 2 minimum de PowerShell. En effet nous faisons appel à la commande Test-Connection et cette dernière n'existe pas dans PowerShell v1.

Param définit la variable ComputerName, celle-ci contiendra le nom de l'ordinateur récupéré sur la ligne de commandes. Nous utilisons ensuite la commande ping pour déterminer si la machine qui recevra la requête WMI est bien allumée. Pour raccourcir le temps de réponse nous avons positionné le paramètre Count à 1, ce qui limite à un seul envoi de ping (par défaut 3 ping sont envoyés). Le paramètre Quiet permet d'obtenir un retour de type booléen au lieu d'un objet de type Win32_PingStatus, comme par défaut.

Arrive ensuite l'essence même du script : la requête WMI, ou plutôt les requêtes devrait-on dire. La première énumère les interfaces réseaux ayant un statut « connecté ». La seconde utilise une autre classe WMI pour lister les configurations réseaux des interfaces. Nous lui appliquons un filtre afin qu'elle ne nous retourne que la configuration des interfaces réseaux « actives » (connectées) dont l'identifiant a été récupéré avec la première requête.

Nous demandons la propriété DNSServerSearchOrder, celle-ci nous retourne le paramétrage DNS de la configuration réseau active.

Pour terminer, nous créons un objet personnalisé de type PSObject et lui ajoutons quelques propriétés. Cela permet à notre script de retourner un objet plutôt que simplement du texte. Vous comprendrez plus loin l'intérêt de faire cela.

Test en vraie grandeur

Pour tester notre script sur un nombre plus important de machines, l'idéal serait d'aller chercher le nom des machines directement dans l'Active Directory Domain Services ou plus simplement, pour commencer, dans un fichier texte.

Dans ce cas, nous pourrions écrire ceci :

```

PS > Get-Content ficMachines.txt | Foreach {./Get-DNSConfiguration.ps1 $_}

```

Si cependant nous avons importé dans notre session les commandelettes du module Active Directory (cf. Chapitre Exécution à distance - Communications à distance Windows PowerShell) alors nous pourrions écrire ceci :

```

PS > Get-ADComputer -Filter * | Foreach {./Get-DnsConfiguration $_.Name}

```

Dans les deux cas, voici ce que nous obtenons :

HostName	Ping	DNS
-----	----	---
W2K8R2VM	OK	{127.0.0.1}
WIN7_US_X64	OK	{192.168.1.23}
WINXP	NOK	
WIN2K8X64	OK	{192.168.1.23}
W2K3R2SRV	OK	{192.168.1.23, 192.168.1.24}
WINXPPSV2	OK	{192.168.1.23}
EXCH2K10SRV	OK	{192.168.1.23, 192.168.1.24}
WINXPPSV1	OK	{192.168.1.23}

Plutôt intéressant n'est-ce pas ? Nous pouvons constater au passage que la machine nommée « WINXP » n'a pas répondu au ping, elle a donc le statut « NOK ». Par conséquent, nous n'avons pas effectué de requêtes WMI sur celle-ci afin d'éviter un timeout, ce qui nous fait gagner du temps dans l'exécution de notre script.

 Les valeurs pour la propriété DNS apparaissent entre accolades, cela signifie que le résultat obtenu est de type tableau (array).

Optimisations PowerShell v2 avec les fonctions avancées

Les fonctions avancées permettent aux scripts de se comporter comme des commandelettes natives. L'utilisation de ces fonctions permettrait à notre script d'accepter nativement le pipeline comme flux d'entrée, ce qui nous éviterait d'avoir à faire un *Foreach* sur la ligne de commande et la simplifierait considérablement. De plus, nous aurions toujours la possibilité de passer un nom d'ordinateur au paramètre Computer.

Observons à présent les modifications apportées à notre script dont nous avons au passage changé le numéro de version :

```
# Get-DNSConfiguration - v.2
#requires -version 2

[CmdletBinding()]
param (
    [Parameter(Mandatory=$true, ValueFromPipeline=$true)]
    [String[]]$Computers
)

Process
{
    foreach ($computerName in $Computers)
    {
        if (Test-Connection $ComputerName -Quiet -Count 1)
        {
            # récupération des interfaces réseaux actives
            $cnxActives = Get-WmiObject Win32_NetworkAdapter -Computer $ComputerName |
                Where {$_.NetConnectionStatus -eq 2}

            # récupération de la configuration des interfaces réseaux actives
            $confActive =
                Get-WmiObject Win32_NetworkAdapterConfiguration -Comp $ComputerName |
                Where { ($_.index -eq $($cnxActives.index)) }
            $result = $confActive.DNSServerSearchOrder
            $Ping = 'OK'
        }
        else
        {
            $result = $null
            $Ping = 'NOK'
        }

        $objResult = New-Object PSObject
        $objResult | add-Member -memberType NoteProperty -name HostName `
            -value $ComputerName
        $objResult | add-Member -memberType NoteProperty -name Ping -value $Ping
        $objResult | add-Member -memberType NoteProperty -name DNS -value $result
        $objResult
    }
}
```

Les modifications réalisées dans ce script sont les suivantes :

- Ajout de [CmdletBinding()] pour indiquer l'usage des fonctions avancées,
- Ajout au niveau du bloc Param de [Parameter(Mandatory=\$true, ValueFromPipeline=\$true)], afin d'obliger l'entrée d'un paramètre et d'autoriser la réception des paramètres via le pipe,
- Ajout d'un bloc Process, nécessaire dans les fonctions avancées qui utilisent le pipeline. Nous aurions pu aussi ajouter les blocs Begin et End.
- Ajout d'une boucle Foreach pour traiter le cas de paramètres multiples.

À présent si nous avons une liste d'ordinateurs dans un fichier, nous pouvons écrire ceci :

```
PS > Get-Content ficMachines.txt | ./Get-DNSConfiguration.ps1
```

Ou encore ceci :

```
PS > ./Get-DNSConfiguration.ps1 -Computers (Get-Content ficMachines.txt)
```

Et si nous prenons les ordinateurs à partir de l'Active Directory :

```
PS > Get-ADComputer -Filter * | Select Name -Expand Name |  
.\Get-DNSConfiguration.ps1
```

Il nous faut nécessairement passer par un filtre de type Select-Object afin de ne récupérer qu'une seule propriété d'objet : le nom. Sinon toutes les propriétés des objets sont passées au pipe ce qui a pour conséquence de générer une erreur.

Revenons à notre étude de cas. Il nous faut maintenant construire le script qui effectuera la modification de la configuration réseau. Observons celui-ci :

```
# Set-DNSConfiguration  
  
param (  
    $ComputerName = $(throw "Vous devez spécifier un nom d'ordinateur !"),  
    $DNSServerList = @('192.168.1.30', '192.168.1.40')  
)  
  
# récupération des interfaces réseaux actives  
$cnxActives = Get-WmiObject Win32_NetworkAdapter -Computer $ComputerName |  
    Where {$_.NetConnectionStatus -eq 2}  
  
# récupération de la configuration des interfaces réseaux actives  
$confActive = Get-WmiObject Win32_NetworkAdapterConfiguration -Comp $ComputerName |  
    Where { ($_.index -eq $($cnxActives.index)) }  
  
$result = $confActive.SetDNSServerSearchOrder($DNSServerList)  
  
if ($result.returnValue -eq 0)  
{  
    Write-Host "$ComputerName : Mise à jour DNS réussie" -foreground Yellow  
}  
else  
{  
    Write-Host "$ComputerName : Echec de la mise à jour DNS !" -foreground Yellow  
}
```

Cette fois-ci notre script est quasi-identique à la première version du script Get-DNSConfiguration.ps1 excepté qu'au lieu de récupérer une propriété il applique une méthode. Il s'agit de la méthode *SetDNSServerSearchOrder*. Celle-ci prend en paramètre un objet de type tableau de chaînes de caractères contenant les adresses IP de nos nouveaux serveurs DNS.

Nous pouvons remarquer aussi que nous avons un paramètre de plus : *DNSServerList*. Ce dernier est initialisé avec des valeurs ; ces dernières constituent alors les valeurs par défaut.

Observons notre script en action :

```
PS > ./Set-DNSConfiguration -ComputerName WinXPPSV2
```

WinXPPSV2 : Mise à jour DNS réussie

Nous venons d'appliquer la nouvelle configuration DNS avec les valeurs par défaut à la machine WinXPPSV2. Vérifions si cela a bien fonctionné :

```
PS > ./Get-DNSConfiguration -ComputerName WinXPPSV2
```

HostName	Ping	DNS
-----	----	---
WinXPPSV2	OK	{192.168.1.30, 192.168.1.40}

Parfait, passons maintenant à la vitesse supérieure ! Supposons que nous voulions être sélectifs dans la façon d'appliquer les changements. Nous ne souhaitons appliquer les nouveaux paramètres qu'aux machines ayant dans leur configuration DNS l'adresse IP 192.168.1.24.

Avant de modifier quoi que ce soit regardons d'abord la configuration générale de notre parc de machines :

```
PS > Get-ADComputer -Filter * | Select Name -Expand Name | .\Get-DNSConfiguration.ps1
```

HostName	Ping	DNS
-----	----	---
W2K8R2VM	OK	{127.0.0.1}
WIN7_US_X64	OK	{192.168.1.23}
WINXP	NOK	
WIN2K8X64	OK	{192.168.1.23}
W2K3R2SRV	OK	{192.168.1.23, 192.168.1.24}
WINXPPSV2	OK	{192.168.1.30, 192.168.1.40}
EXCH2K10SRV	OK	{192.168.1.23, 192.168.1.24}
WINXPPSV1	OK	{192.168.1.23}

Essayons avec le filtre suivant :

```
PS > Get-ADComputer -Filter * | Select Name -Expand Name | .\Get-DNSConfiguration.ps1 |  
Where { ($_.DNS -contains '192.168.1.24') }
```

HostName	Ping	DNS
-----	----	---
W2K3R2SRV	OK	{192.168.1.23, 192.168.1.24}
EXCH2K10SRV	OK	{192.168.1.23, 192.168.1.24}

Nous avons obtenu les machines référençant dans leur configuration réseau, un serveur DNS ayant l'adresse 192.168.1.24. À présent nous allons appliquer uniquement à ce groupe de machines la nouvelle configuration DNS, comme ceci :

```
PS > $a = Get-ADComputer -Filter * | Select Name -Expand Name |  
.\Get-DNSConfiguration.ps1 | Where { ($_.DNS -contains '192.168.1.24') }
```

```
PS > $a | foreach { .\Set-DNSConfiguration.ps1 -ComputerName $_.HostName -  
-DNS @( '192.168.1.50', '192.168.1.60' ) }
```

Nous venons d'affecter une configuration DNS particulière à notre sélection de machines. Ce qui nous donne au final le résultat suivant :

```
PS > Get-ADComputer -Filter * | Select Name -Expand Name | .\Get-DNSConfiguration.ps1
```

HostName	Ping	DNS
-----	----	---
W2K8R2VM	OK	{127.0.0.1}
WIN7_US_X64	OK	{192.168.1.23}
WINXP	NOK	
WIN2K8X64	OK	{192.168.1.23}
W2K3R2SRV	OK	{192.168.1.50, 192.168.1.60}
WINXPPSV2	OK	{192.168.1.30, 192.168.1.40}
EXCH2K10SRV	OK	{192.168.1.50, 192.168.1.60}
WINXPPSV1	OK	{192.168.1.23}

Humm, tout cela ne semble pas très homogène... Allez, faisons un dernier effort pour mettre toutes nos machines à

l'identique en laissant les valeurs par défaut de la fonction Set-DNSConfiguration :

```
PS > Get-ADComputer -Filter * | Select Name -Expand Name |  
    foreach { .\Set-DNSConfiguration.ps1 -ComputerName $_ }
```

```
W2K8R2VM : Opération réussie  
WIN7_US_X64 : Opération réussie  
WIN2K8X64 : Opération réussie  
W2K3R2SRV : Opération réussie  
EXCH2K10SRV : Opération réussie  
WINXPPSV1 : Opération réussie
```

Et voilà...

```
PS > Get-ADComputer -Filter * | Select Name -Expand Name | .\Get-DNSConfiguration.ps1
```

HostName	Ping	DNS
-----	----	---
W2K8R2VM	OK	{192.168.1.30, 192.168.1.40}
WIN7_US_X64	OK	{192.168.1.30, 192.168.1.40}
WINXP	NOK	
WIN2K8X64	OK	{192.168.1.30, 192.168.1.40}
W2K3R2SRV	OK	{192.168.1.30, 192.168.1.40}
WINXPPSV2	OK	{192.168.1.30, 192.168.1.40}
EXCH2K10SRV	OK	{192.168.1.30, 192.168.1.40}
WINXPPSV1	OK	{192.168.1.30, 192.168.1.40}

Ressources Web externes

Vous trouverez dans cette partie quelques sources d'informations techniques autour de PowerShell. À la fois utiles et pratiques, ces dernières vous permettront, si vous le souhaitez, d'approfondir vos connaissances en la matière.

1. Sites Internet Francophones

Powershell-scripting.com : la communauté PowerShell francophone

Ce site entièrement consacré à PowerShell se veut être la référence en matière d'informations ; que celles-ci soient générales ou techniques. Vous y trouverez de nombreux tutoriaux, de nombreux scripts prêts à l'emploi (dans la bibliothèque), ainsi qu'un forum très vivant. De temps à autres, il y a également des concours de scripting organisés sur ce site avec des lots à gagner.

Il est animé par vos nobles serveurs Robin Lemesle et Arnaud Petitjean.



www.powershell-scripting.com, la communauté PowerShell Francophone

Le Blog de Janel (janel.spaces.live.com)

Les ressources en langue française ce font rare, cependant, nous vous conseillons aussi le **Blog de Janel** qui, malgré sa baisse d'activité, est une mine d'informations techniques. Jacques Barathon, alias Janel, travaille chez Microsoft. Son passe temps favori est d'écumer les forums à la recherche de questions pointues sur PowerShell et d'y répondre ; il en fait ensuite un billet sur son Blog en entrant dans les moindres détails. Son Blog est un must de part la qualité de ses explications, à consulter d'urgence...



Le blog de Janel

Le Blog d'Antoine Habert (<http://devinfra.blogspot.com/>)

Antoine Habert est un architecte système et MVP PowerShell qui anime un blog autour de PowerShell, mais également sur un produit de sa propre création : PoshBoard, un outil orienté création d'interface utilisateur via des Widgets intégrés dans un portail web.

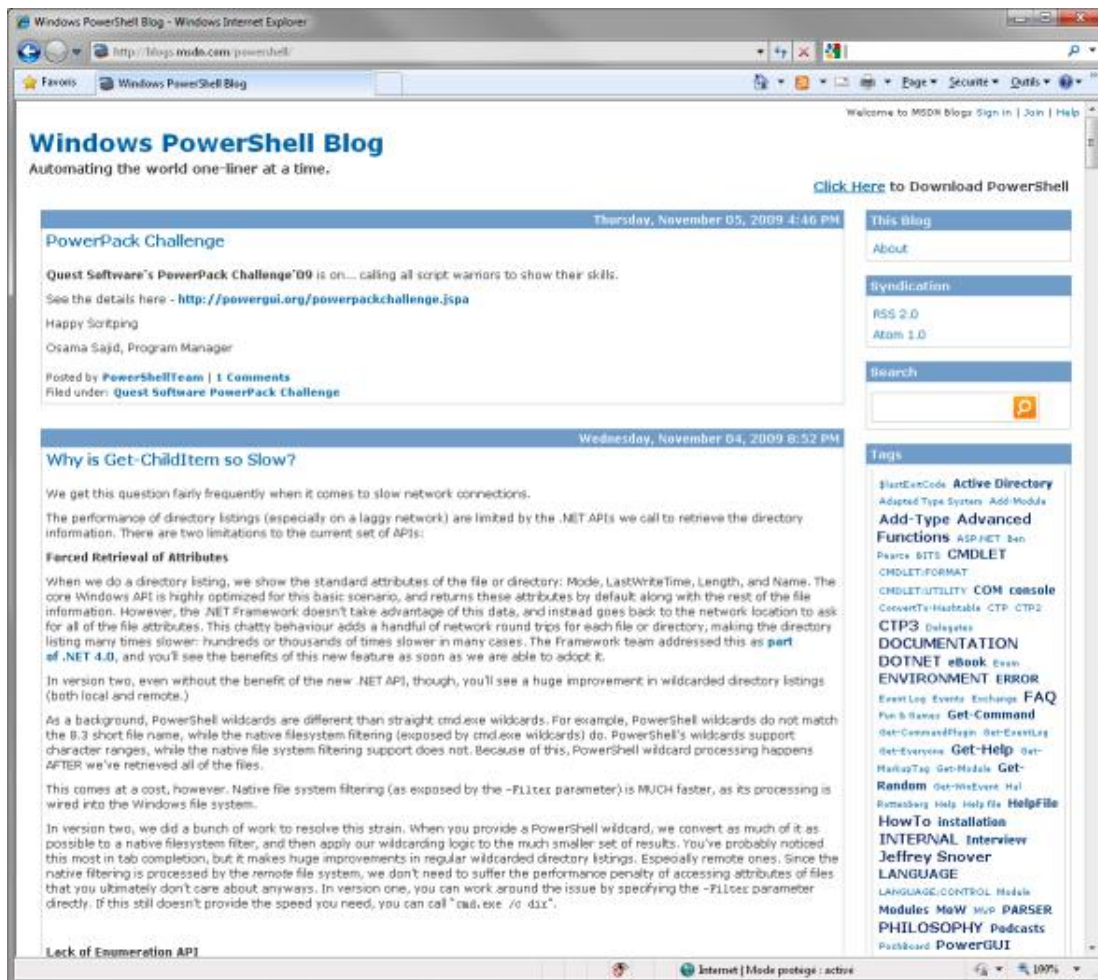


Le blog d'Antoine Habert

2. Sites Internet Anglophones

Le Blog de l'équipe PowerShell (blogs.msdn.com/PowerShell/)

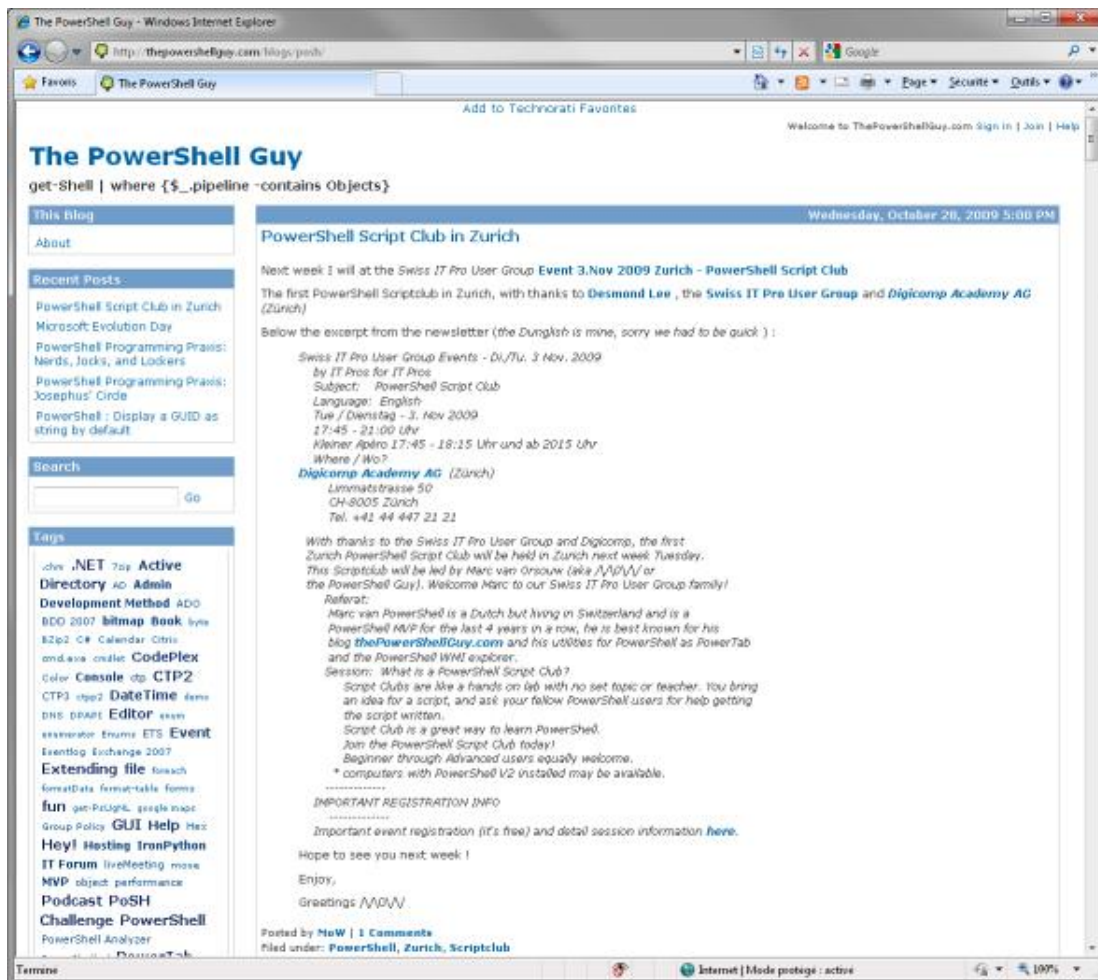
L'équipe PowerShell maintient un Blog où vous y découvrirez toutes les nouveautés lorsqu'il y en a mais surtout de nombreuses astuces ou éclaircissements sur des fonctions non documentées de PowerShell. Vous pouvez également interagir avec les membres de l'équipe en laissant des avis ou commentaires sur chacun de leurs billets.



Le blog de l'équipe PowerShell

Le Blog de MoW (thePowerShellGuy.com/blogs/posh/)

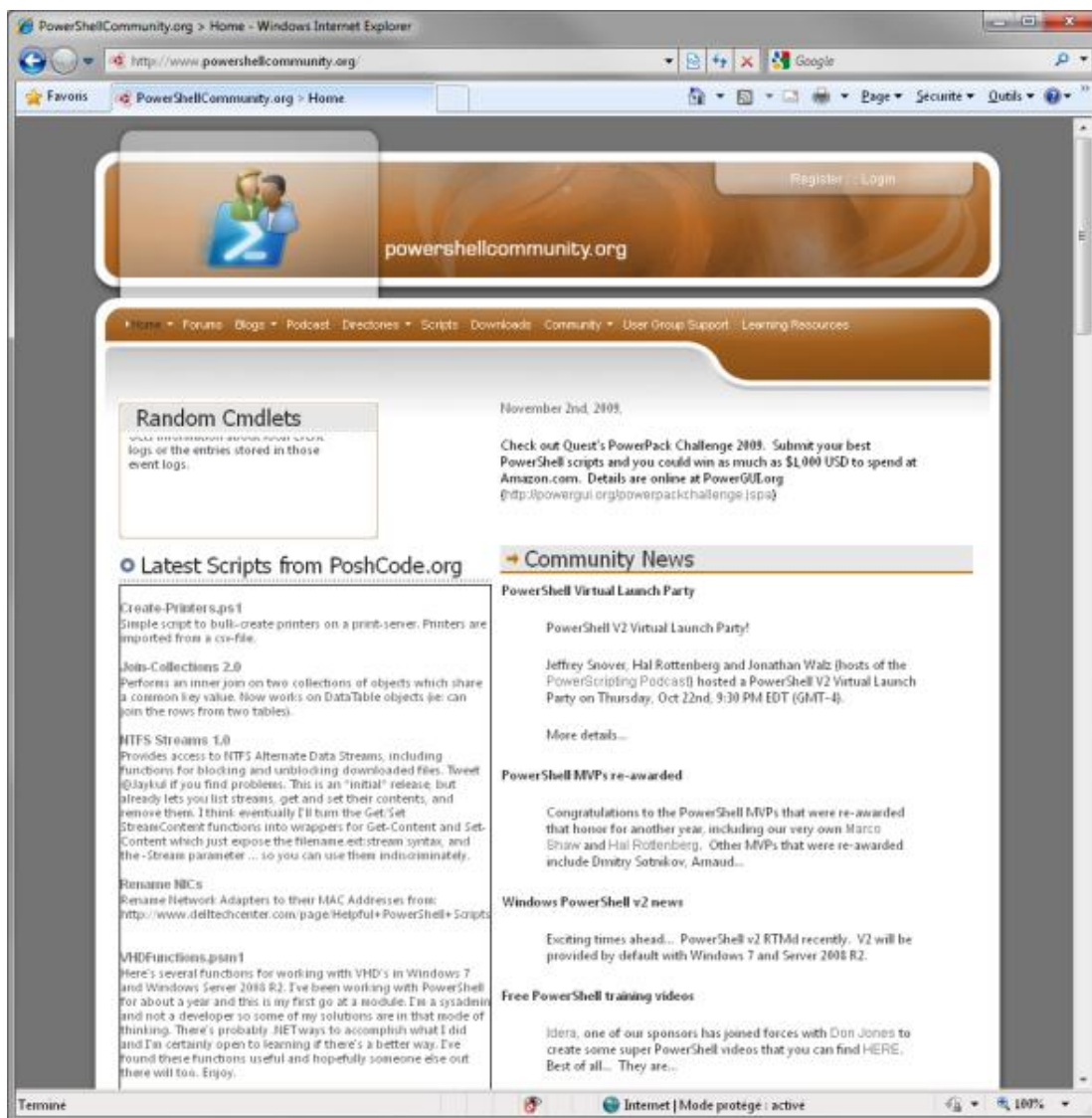
MOW, alias « the PowerShell guy » est un MVP PowerShell très expérimenté. Son vrai nom est Marc Van Orsouw, il est hollandais mais son Blog est en anglais. Il a accompagné sur son Blog (dès 2005) la sortie de PowerShell lorsqu'on en était alors qu'à la version préliminaire qui s'appelait Monad. MOW est reconnu internationalement comme étant l'un des gourous PowerShell extérieur à Microsoft.



Le blog de MOW

www.powershellcommunity.org

Ce site est celui de la communauté PowerShell internationale, il est géré et a été fondé par Don Jones. Don Jones est un MVP américain de renom dans le milieu du scripting notamment par le biais de la société SAPIEN qu'il a lui-même créée. Vous trouverez sur www.powershellcommunity.org de nombreux articles techniques, de nombreuses news, ainsi qu'une base naissante de scripts PowerShell.



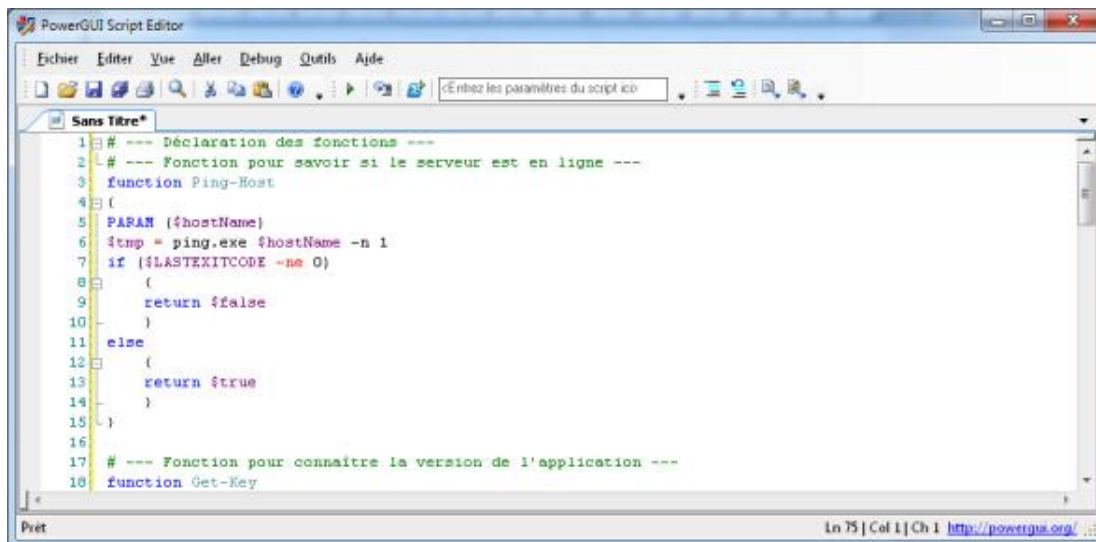
Site de la communauté PowerShell internationale

Outils tiers

Vous trouverez dans cette partie quelques outils tiers qui s'avèrent être bien pratiques et qui apportent un cadre de travail confortable lorsque l'on utilise PowerShell de façon intensive.

1. PowerGUI

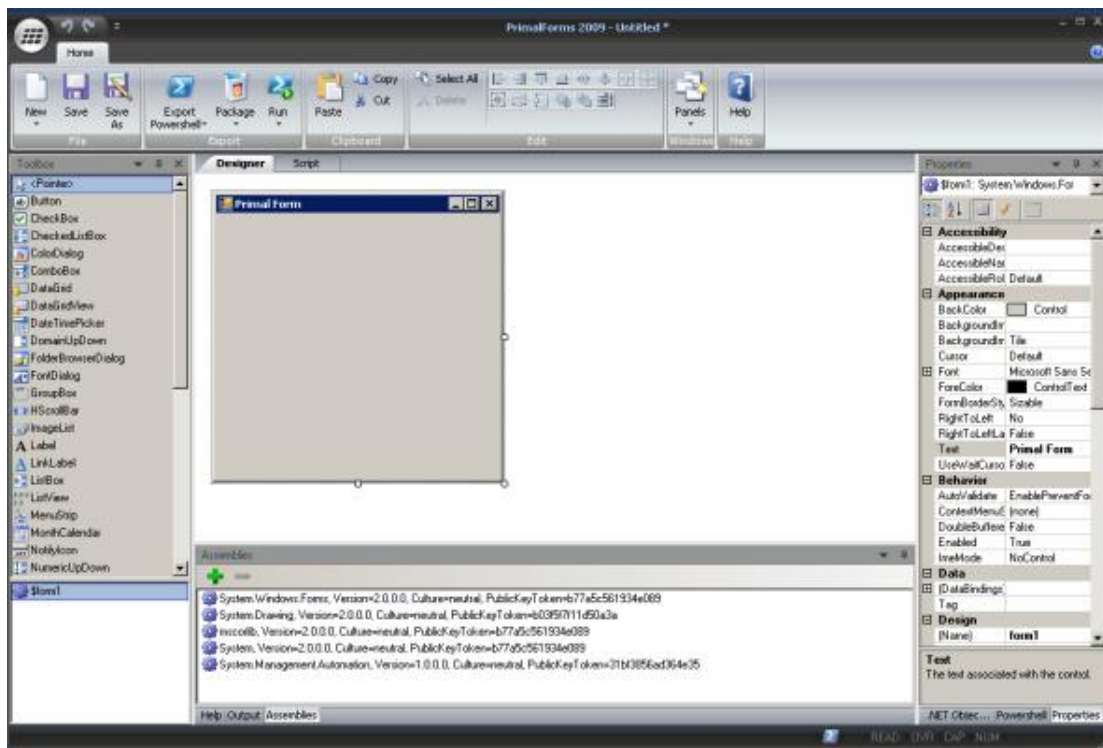
PowerGUI est une console d'édition PowerShell. Bien plus qu'un éditeur de script, PowerGUI est également un outil capable de gérer des plug-in divers : Exchange 2007, Operations Manager 2007 et bien d'autres produits. PowerGUI existe depuis près de deux ans, il se trouve être également bien plus complet que PowerShell ISE. Il intègre notamment un éditeur à la Visual Studio avec la fonctionnalité « IntelliSense ». Mais également, une fonctionnalité toute simple que nous apprécions lorsqu'il s'agit de mettre en forme un script : l'indentation des blocs d'instructions. Certains d'entre vous qui pratiquent la virtualisation le connaissons peut-être sous l'appellation « The VESI ». car PowerGUI est intégré dans le projet communautaire « Virtualization EcoShell ».



Interface de PowerGUI

2. PrimalForm

PrimalForms est un outil logiciel édité par **Sapien Technologies**. Cet outil vient se placer comme le premier du genre à permettre la création d'interfaces graphiques pour les utilisateurs de PowerShell. L'apparition d'un réel éditeur de formulaire graphique vient apporter ce qu'il manquait vraiment pour une plus grande efficacité dans la conception de scripts à interaction visuelle.



Interface de PrimalForm

3. PowerGadget

Les PowerGadgets (www.softwarefx.com/sfxSqlProducts/powergadgets/)

Softwarefx.com est une société américaine qui développe et commercialise des gadgets pour le volet Windows que vous pouvez programmer ou personnaliser grâce à PowerShell. Ces gadgets sont en réalité de véritables instruments de supervision sur votre poste de travail. En effet, un PowerGadget est un gadget au design réussi qui peut être de différents types, jauge, plan, courbe ou même graphique.



PowerGadgets

Conclusion

Vous avez pu voir, tout au long de la lecture de cet ouvrage, qu'il n'existe pas une solution unique pour chaque problème. En effet, il en existe de nombreuses qui diffèrent selon les connaissances de chacun. En outre, vous aurez remarqué que les technologies COM, WMI, ADSI et .NET souvent se chevauchent, ce qui fait qu'il est parfois difficile de répondre à une question aussi simple que celle-ci : « avec quelle technologie vais-je réussir à automatiser telle ou telle tâche ? ».

Nous vous conseillons de regarder d'abord dans le jeu de commandes PowerShell, puis si vous ne trouvez pas votre bonheur de vous tourner vers .NET (les accès avancés à WMI et à ADSI se faisant par son intermédiaire), et en dernier recours de vous tourner vers COM. Il est préférable de privilégier les technologies les plus récentes aux plus anciennes.

Vous aurez également remarqué que l'accès au Framework .NET étend considérablement le champ d'action de PowerShell. Par conséquent, selon le degré de complexité des scripts, la frontière entre le scripting et le développement devient de plus en plus mince. Nous connaissons d'ailleurs certains développeurs C# qui se tournent régulièrement vers PowerShell pour tester certaines classes du Framework ou pour réaliser des maquettes. Ils font ainsi parce que PowerShell leur fait gagner du temps du fait qu'il est possible grâce à son interpréteur de commandes de tester rapidement des morceaux de code sans avoir à les compiler. Il faut avouer que le langage de Windows PowerShell est assez proche du C#, en exagérant un peu on pourrait presque l'assimiler à une version simplifiée de celui-ci.

Ainsi un administrateur système aguerri aux techniques du scripting PowerShell et à .NET pourra, lui aussi, sans beaucoup d'efforts tenter une incursion dans le monde du développement d'application en C#. Peut-être pour ajouter des fonctionnalités supplémentaires à PowerShell... ?

Annexe 1 : Liste des commandes PowerShell v1

La version 1 de PowerShell est riche d'un jeu de 129 commandes.

Add-Content	Add-History	Add-Member
Add-PSSnapin	Clear-Content	Clear-Item
Clear-ItemProperty	Clear-Variable	Compare-Object
ConvertFrom-SecureString	Convert-Path	ConvertTo-Html
ConvertTo-SecureString	Copy-Item	Copy-ItemProperty
Export-Alias	Export-Clixml	Export-Console
Export-Csv	ForEach-Object	Format-Custom
Format-List	Format-Table	Format-Wide
Get-Acl	Get-Alias	Get-AuthenticodeSignature
Get-ChildItem	Get-Command	Get-Content
Get-Credential	Get-Culture	Get-Date
Get-EventLog	Get-ExecutionPolicy	Get-Help
Get-History	Get-Host	Get-Item
Get-ItemProperty	Get-Location	Get-Member
Get-PfxCertificate	Get-Process	Get-PSDrive
Get-PSPProvider	Get-PSSnapin	Get-Service
Get-TraceSource	Get-UICulture	Get-Unique
Get-Variable	Get-WmiObject	Group-Object
Import-Alias	Import-Clixml	Import-Csv
Invoke-Expression	Invoke-History	Invoke-Item
Join-Path	Measure-Command	Measure-Object
Move-Item	Move-ItemProperty	New-Alias
New-Item	New-ItemProperty	New-Object
New-PSDrive	New-Service	New-TimeSpan
New-Variable	Out-Default	Out-File
Out-Host	Out-Null	Out-Printer
Out-String	Pop-Location	Push-Location
Read-Host	Remove-Item	Remove-ItemProperty
Remove-PSDrive	Remove-PSSnapin	Remove-Variable
Rename-Item	Rename-ItemProperty	Resolve-Path
Restart-Service	Resume-Service	Select-Object
Select-String	Set-Acl	Set-Alias
Set-AuthenticodeSignature	Set-Content	Set-Date
Set-ExecutionPolicy	Set-Item	Set-ItemProperty
Set-Location	Set-PSDebug	Set-Service
Set-TraceSource	Set-Variable	Sort-Object
Split-Path	Start-Service	Start-Sleep
Start-Transcript	Stop-Process	Stop-Service
Stop-Transcript	Suspend-Service	Tee-Object
Test-Path	Trace-Command	Update-FormatData
Update-TypeData	Where-Object	Write-Debug
Write-Error	Write-Host	Write-Output
Write-Progress	Write-Verbose	Write-Warning

Annexe 2 : Liste des commandes PowerShell v2

La version 2 apporte 107 nouvelles commandes à la version 1, soit un total de 236.

Add-Computer	Add-Type
Checkpoint-Computer	Clear-EventLog
Clear-History	Complete-Transaction
Connect-WSMan	ConvertFrom-Csv
ConvertFrom-StringData	ConvertTo-Csv
ConvertTo-Xml	Debug-Process
Disable-ComputerRestore	Disable-PSBreakpoint
Disable-PSSessionConfiguration	Disable-WSManCredSSP
Disconnect-WSMan	Enable-ComputerRestore
Enable-PSBreakpoint	Enable-PSRemoting
Enable-PSSessionConfiguration	Enable-WSManCredSSP
Enter-PSSession	Exit-PSSession
Export-Counter	Export-FormatData
Export-ModuleMember	Export-PSSession
Get-ComputerRestorePoint	Get-Counter
Get-Event	Get-EventSubscriber
Get-FormatData	Get-HotFix
Get-Job	Get-Module
Get-PSBreakpoint	Get-PSCallStack
Get-PSSession	Get-PSSessionConfiguration
Get-Random	Get-Transaction
Get-WinEvent	Get-WSManCredSSP
Get-WSManInstance	Import-Counter
Import-LocalizedData	Import-Module
Import-PSSession	Invoke-Command
Invoke-WmiMethod	Invoke-WSManAction
Limit-EventLog	New-Event
New-EventLog	New-Module
New-ModuleManifest	New-PSSession
New-PSSessionOption	New-WebServiceProxy
New-WSManInstance	New-WSManSessionOption
Out-GridView	Receive-Job
Register-EngineEvent	Register-ObjectEvent
Register-PSSessionConfiguration	Register-WmiEvent
Remove-Computer	Remove-Event
Remove-EventLog	Remove-Job
Remove-Module	Remove-PSBreakpoint
Remove-PSSession	Remove-WmiObject
Remove-WSManInstance	Reset-ComputerMachinePassword
Restart-Computer	Restore-Computer
Select-Xml	Send-MailMessage
Set-PSBreakpoint	Set-PSSessionConfiguration
Set-StrictMode	Set-WmiInstance
Set-WSManInstance	Set-WSManQuickConfig
Show-EventLog	Start-Job
Start-Process	Start-Transaction
Stop-Computer	Stop-Job
Test-ComputerSecureChannel	Test-Connection
Test-ModuleManifest	Test-WSMan
Undo-Transaction	Unregister-Event
Unregister-PSSessionConfiguration	Update-List
Use-Transaction	Wait-Event
Wait-Job	Wait-Process
Write-EventLog	

Annexe 3 : Liste des commandes du module Active Directory

Le module Active Directory de Windows Server 2008 R2 apporte 76 commandes supplémentaires.

```
Add-ADComputerServiceAccount
Add-ADDomainControllerPasswordReplicationPolicy
Add-ADFineGrainedPasswordPolicySubject
Add-ADGroupMember
Add-ADPrincipalGroupMembership
Clear-ADAccountExpiration
Disable-ADAccount
Disable-ADOptionalFeature
Enable-ADAccount
Enable-ADOptionalFeature
Get-ADAccountAuthorizationGroup
Get-ADAccountResultantPasswordReplicationPolicy
Get-ADComputer
Get-ADComputerServiceAccount
Get-ADDefaultDomainPasswordPolicy
Get-ADDomain
Get-ADDomainController
Get-ADDomainControllerPasswordReplicationPolicy
Get-ADDomainControllerPasswordReplicationPolicyUsage
Get-ADFineGrainedPasswordPolicy
Get-ADFineGrainedPasswordPolicySubject
Get-ADForest
Get-ADGroup
Get-ADGroupMember
Get-ADObject
Get-ADOptionalFeature
Get-ADOrganizationalUnit
Get-ADPrincipalGroupMembership
Get-ADRootDSE
Get-ADServiceAccount
Get-ADUser
Get-ADUserResultantPasswordPolicy
Install-ADServiceAccount
Move-ADDirectoryServer
Move-ADDirectoryServerOperationMasterRole
Move-ADObject
New-ADComputer
New-ADFineGrainedPasswordPolicy
New-ADGroup
New-ADObject
New-ADOrganizationalUnit
New-ADServiceAccount
New-ADUser
Remove-ADComputer
Remove-ADComputerServiceAccount
Remove-ADDomainControllerPasswordReplicationPolicy
Remove-ADFineGrainedPasswordPolicy
Remove-ADFineGrainedPasswordPolicySubject
Remove-ADGroup
Remove-ADGroupMember
Remove-ADObject
Remove-ADOrganizationalUnit
Remove-ADPrincipalGroupMembership
Remove-ADServiceAccount
Remove-ADUser
Rename-ADObject
Reset-ADServiceAccountPassword
Restore-ADObject
Search-ADAccount
Set-ADAccountControl
Set-ADAccountExpiration
Set-ADAccountPassword
Set-ADComputer
```

Set-ADDefaultDomainPasswordPolicy
Set-ADDomain
Set-ADDomainMode
Set-ADFineGrainedPasswordPolicy
Set-ADForest
Set-ADForestMode
Set-ADGroup
Set-ADObject
Set-ADOrganizationalUnit
Set-ADServiceAccount
Set-ADUser
Uninstall-ADServiceAccount
Unlock-ADAccount

Annexe 4 : Liste des alias

Name	Definition
----	-----
%	ForEach-Object
?	Where-Object
ac	Add-Content
asnp	Add-PSSnapIn
cat	Get-Content
cd	Set-Location
chdir	Set-Location
clc	Clear-Content
clear	Clear-Host
clhy	Clear-History
cli	Clear-Item
clp	Clear-ItemProperty
cls	Clear-Host
clv	Clear-Variable
compare	Compare-Object
copy	Copy-Item
cp	Copy-Item
cpi	Copy-Item
cpp	Copy-ItemProperty
cvpa	Convert-Path
dbp	Disable-PSBreakpoint
del	Remove-Item
diff	Compare-Object
dir	Get-ChildItem
ebp	Enable-PSBreakpoint
echo	Write-Output
epal	Export-Alias
epcsv	Export-Csv
epsn	Export-PSSession
erase	Remove-Item
etsn	Enter-PSSession
exsn	Exit-PSSession
fc	Format-Custom
fl	Format-List
foreach	ForEach-Object
ft	Format-Table
fw	Format-Wide
gal	Get-Alias
gbp	Get-PSBreakpoint
gc	Get-Content
gci	Get-ChildItem
gcm	Get-Command
gcs	Get-PSCallStack
gdr	Get-PSDrive
ghy	Get-History
gi	Get-Item
gjb	Get-Job
gl	Get-Location
gm	Get-Member
gmo	Get-Module
gp	Get-ItemProperty
gps	Get-Process
group	Group-Object
gsn	Get-PSSession
gsnp	Get-PSSnapIn
gsv	Get-Service
gu	Get-Unique
gv	Get-Variable
gwmi	Get-WmiObject
h	Get-History
history	Get-History
icm	Invoke-Command
iex	Invoke-Expression

ihy	Invoke-History
ii	Invoke-Item
ipal	Import-Alias
ipcsv	Import-Csv
ipmo	Import-Module
ipsn	Import-PSSession
ise	powershell_ise.exe
iwmi	Invoke-WMIMethod
kill	Stop-Process
lp	Out-Printer
ls	Get-ChildItem
man	help
md	mkdir
measure	Measure-Object
mi	Move-Item
mount	New-PSDrive
move	Move-Item
mp	Move-ItemProperty
mv	Move-Item
nal	New-Alias
ndr	New-PSDrive
ni	New-Item
nmo	New-Module
nsn	New-PSSession
nv	New-Variable
ogv	Out-GridView
oh	Out-Host
popd	Pop-Location
ps	Get-Process
pushd	Push-Location
pwd	Get-Location
r	Invoke-History
rbp	Remove-PSBreakpoint
rcjb	Receive-Job
rd	Remove-Item
rdr	Remove-PSDrive
ren	Rename-Item
ri	Remove-Item
rjb	Remove-Job
rm	Remove-Item
rmdir	Remove-Item
rmo	Remove-Module
rni	Rename-Item
rnp	Rename-ItemProperty
rp	Remove-ItemProperty
rsn	Remove-PSSession
rsnp	Remove-PSSnapin
rv	Remove-Variable
rvpa	Resolve-Path
rwmi	Remove-WMIObject
sajb	Start-Job
sal	Set-Alias
saps	Start-Process
sasv	Start-Service
sbp	Set-PSBreakpoint
sc	Set-Content
select	Select-Object
set	Set-Variable
si	Set-Item
sl	Set-Location
sleep	Start-Sleep
sort	Sort-Object
sp	Set-ItemProperty
spjb	Stop-Job
spps	Stop-Process
spsv	Stop-Service
start	Start-Process
sv	Set-Variable
swmi	Set-WMIInstance

tee	Tee-Object
type	Get-Content
where	Where-Object
wjb	Wait-Job
write	Write-Output

Annexe 5 : Liste des fonctions

CommandType	Name	Definition
Function	A:	Set-Location A:
Function	B:	Set-Location B:
Function	C:	Set-Location C:
Function	cd..	Set-Location ..
Function	cd\	Set-Location \
Function	Clear-Host	\$space = New-Object System.Management.Auto...
Function	D:	Set-Location D:
Function	Disable-PSRemoting	...
Function	E:	Set-Location E:
Function	F:	Set-Location F:
Function	G:	Set-Location G:
Function	Get-Verb	...
Function	H:	Set-Location H:
Function	help	...
Function	I:	Set-Location I:
Function	ImportSystemModules	...
Function	J:	Set-Location J:
Function	K:	Set-Location K:
Function	L:	Set-Location L:
Function	M:	Set-Location M:
Function	mkdir	...
Function	more	param([string[]]\$paths)...
Function	N:	Set-Location N:
Function	O:	Set-Location O:
Function	P:	Set-Location P:
Function	prompt	\$(if (test-path variable:/PSDebugContext){ '[DBG]:...
Function	Q:	Set-Location Q:
Function	R:	Set-Location R:
Function	S:	Set-Location S:
Function	T:	Set-Location T:
Function	TabExpansion	...
Function	U:	Set-Location U:
Function	V:	Set-Location V:
Function	W:	Set-Location W:
Function	X:	Set-Location X:
Function	Y:	Set-Location Y:
Function	Z:	Set-Location Z:

Annexe 6 : Liste des sources de trace (Get-Tracesource)

Name	Description
----	-----
SingleShell	SingleShell
MshSnapinLoadUnload	Loading and unloading mshsnapins
MshConsoleInfo	MshConsoleInfo object that is constructed fr...
PSSnapInInfo	PSSnapInInfo
PSSnapInReader	PSSnapInReader
RunspaceConfigurationEntryCollection	RunspaceConfigurationEntryCollection
RunspaceConfigurationEntry	RunspaceConfigurationEntry
CmdletConfigurationEntry	CmdletConfigurationEntry
AssemblyConfigurationEntry	AssemblyConfigurationEntry
RunspaceConfiguration	RunspaceConfiguration
RunspaceInit	Initialization code for Runspace
CmdletProviderAttribute	The attribute that declares that a class is...
ProviderConfigurationEntry	ProviderConfigurationEntry
ResourceManagerCache	Maintains a cache of the loaded resource man...
ConsoleShell	Entry point in to ConsoleShell
ConsoleHost	ConsoleHost subclass of S.M.A.PSHost
ConsoleHostRunspaceInit	Initialization code for ConsoleHost's Runspace
ConsoleHostUserInterface	Console host's subclass of S.M.A.Host.Console
ConsoleHostRawUserInterface	Console host's subclass of S.M.A.Host.RawCon...
ConsoleControl	Console control methods
LocalRunspace	LocalRunspace
UniversalResourceName	The namespace navigation tracer
ExecutionContext	The execution context of a particular instan...
AuthorizationManager	tracer for AuthorizationManager
PSAuthorizationManager	tracer for PSAuthorizationManager
InternalHost	S.M.A.InternalHostUserInterface
InternalHostRawUserInterface	S.M.A.InternalHostRawUserInterface
SessionState	SessionState Class
SessionStateScope	A scope of session state that holds virtual ...
Runspace	Runspace base class
ETS	Extended Type System
TypeConversion	Traces the type conversion algorithm
Parser	Parser
Tokenizer	Tokenizer
DetailedCommandDiscovery	The detailed tracing of CommandDiscovery. Su...
CommandDiscovery	Traces the discovery of cmdlets, scripts, fu...
TypeTable	TypeTable
DeserializingTypeConverter	DeserializingTypeConverter class
TypeInfoDataBaseManager	TypeInfoDataBaseManager
TypeInfoDataBaseLoader	TypeInfoDataBaseLoader
XmlLoaderBase	XmlLoaderBase
FormatFileLoading	Loading format files
State	The APIs that are exposed to the Cmdlet base...
CoreCommandProvider	The namespace navigation tracer
PSCredential	PSCredential
PSDriveInfo	The namespace navigation tracer
CmdletProviderClasses	The namespace provider base classes tracer
PSTransaction	PSTransaction
CmdletProviderContext	The context under which a core command is be...
MshLog	MshLog
EventLogLogProvider	EventLogLogProvider
AliasProvider	The CmdletProvider for shell aliases
EnvironmentProvider	The core command provider for environment va...
FileSystemProvider	The namespace navigation provider for the fi...
VariableProvider	The core command provider for shell variables
RegistryProvider	The namespace navigation provider for the Wi...
CertificateProvider	The core command provider for certificates
X509StoreLocation	store location information
DriveCommandAPI	The APIs that are exposed to the Cmdlet base...
LocationGlobber	The location globber converts PowerShell pat...
PathResolution	Traces the path resolution algorithm.
ProviderCommandAPI	The APIs that are exposed to the Cmdlet base...
WildcardPattern	WildcardPattern

PathInfo	An object that represents a path in Monad.
CommandFactory	CommandFactory
History	History class
InitialSessionState	InitialSessionState
CommandMetadata	The metadata associated with a cmdlet.
RuntimeDefinedParameters	The classes representing the runtime-defined...
CompiledCommandParameter	The metadata associated with a parameter tha...
ParameterCollectionTypeInfo	A class that wraps up the type information a...
CompiledCommandAttribute	The metadata associated with an attribute th...
ParameterSetSpecificMetadata	The metadata associated with a parameterset ...
ParameterMetadata	The metadata associated with a bindable obje...
MemberResolution	Traces the resolution from member name to th...
PSSnapInLoadUnload	Loading and unloading mshsnapins
HostUtilities	tracer for HostUtilities
Executor	ConsoleHost pipeline execution helper
PipelineStateInfo	PipelineStateInfo
Pipeline	Pipeline
PipelineBase	PipelineBase
ParameterCollection	ParameterCollection
ObjectStream	Read/Write memory-based object stream
ObjectWriter	Writer for ObjectStream
PipelineThread	PipelineThread
LocalPipeline	LocalPipeline
InternalCommand	InternalCommand
ScriptCommandProcessor	ScriptCommandProcessor
PipelineProcessor	PipelineProcessor
ParameterBinderBase	A abstract helper class for the CommandProce...
ParameterBinding	Traces the process of binding the arguments ...
ParameterBinderController	Controls the interaction between the command...
ParameterBindingException	Exception thrown when a parameter binding er...
CommandSearch	CommandSearch
CommandProcessor	CommandProcessor
NavigationCommands	The namespace navigation tracer
Cmdlet	Cmdlet
CommandParameterInternal	Internal definition of a parameter
ReflectionParameterBinder	The parameter binder for real CLR objects th...
CredentialAttribute	CredentialAttribute
CmdletProviderIntrinsics	The APIs that are exposed to the Cmdlet base...
ProviderIntrinsics	The APIs that are exposed to the Cmdlet base...
CommandProcessorBase	CommandProcessorBase
SessionStateProvider	Providers that produce a view of session sta...
PathCommandAPI	The APIs that are exposed to the Cmdlet base...
OutDefaultCommand	OutDefaultCommand
TerminatingErrorContext	TerminatingErrorContext
ConsoleLineOutput	ConsoleLineOutput
WriteLineHelper	WriteLineHelper
format_out_OutputManagerInner	OutputManagerInner
format_out_FrontEndCommandBase	FrontEndCommandBase
format_out_ImplementationCommandBase	ImplementationCommandBase
CommandNotFoundException	Exception thrown when a command could not be...
ErrorRecord	ErrorRecord
format_out_CommandWrapper	CommandWrapper
OutLineOutputCommand	Out-Lineoutput Implementation
format_out_OutCommandInner	OutCommandInner
FormatObjectDeserializer	class to deserialize property bags into form...
Deserializer	Deserializer class
PSVariableCommandAPI	The APIs that are exposed to the Cmdlet base...
ErrorCategoryInfo	ErrorCategoryInfo
FormatInfoDataClassFactory	FormatInfoDataClassFactory
FormatViewBinding	Format view binding
DisplayDataQuery	DisplayDataQuery
TypeMatch	F&O TypeMatch
CommandCompletion	Command completion functionality
Parameter	Simple name/value pair
ExecutionHelper	CommandCompletion execution helper
FileSystemContentStream	The provider content reader and writer for t...
SortObject	Class that has sort-object command implement...
OrderObjectBase	Base class for sort like command implementation
OrderByProperty	Util class for sort like command implementation

ParameterProcessor	ParameterProcessor
ObjectCommandComparer	ObjectCommandComparer
GetCommandCmdlet	Trace output for get-command
CommonCommandParameters	This class is used to expose the ubiquitous ...
HostCmdlets	trace switch for *-host and related cmdlets
MeasureObject	Class that has measure-object command implem...
SessionStateException	SessionStateException
ApplicationInfo	The command information for applications tha...
NativeCP	NativeCP
NativeCommandParameterBinder	The parameter binder for native commands
StreamingTextWriter	StreamingTextWriter
ScriptParameterBinder	The parameter binder for shell functions
ScriptAsCmdlet	Trace output for script cmdlets
GetHelpCommand	GetHelpCommand
HelpSystem	HelpSystem
HelpProvider	HelpProvider
AliasHelpProvider	AliasHelpProvider
ProviderHelpProvider	ProviderHelpProvider
HelpFileHelpProvider	HelpFileHelpProvider
HelpProviderWithFullCache	HelpProviderWithFullCache
FaqHelpProvider	FaqHelpProvider
GlossaryHelpProvider	GlossaryHelpProvider
GeneralHelpProvider	GeneralHelpProvider
DefaultHelpProvider	DefaultHelpProvider
HelpRequest	HelpRequest
CommandHelpProvider	CommandHelpProvider
MamlCommandHelpInfo	MamlCommandHelpInfo
HelpInfo	HelpInfo
MamlNode	MamlNode
ProviderCategory	ProviderCategory Class
AliasHelpInfo	AliasHelpInfo
CmdletInfo	The command information for PowerShell cmdle...
ErrorDetails	ErrorDetails
AliasInfo	The command information for aliases. Aliases...
SelectObject	Class that has select-object command impleme...
FunctionInfo	The command information for MSH functions.
ScriptBlock	Traces the execution of a ScriptBlock