

# Programmation avec PowerShell

Windows Powershell™ est un nouvel environnement de ligne de commande Windows spécialement conçu pour les administrateurs système. Il comprend une invite interactive et un environnement de script qui peuvent être utilisés indépendamment l'un de l'autre ou ensemble. Dans ce cours, nous allons nous intéresser principalement à la manière de *"programmer ou d'étendre"* Windows Powershell.

Pour ceux qui souhaitent s'initier aux commandes Windows Powershell, je ne peux que les encourager à télécharger la documentation ici :

<http://www.microsoft.com/downloads/details.aspx?FamilyId=B4720B00-9A66-430F-BD56-EC48BFCA154F&displaylang=en>.

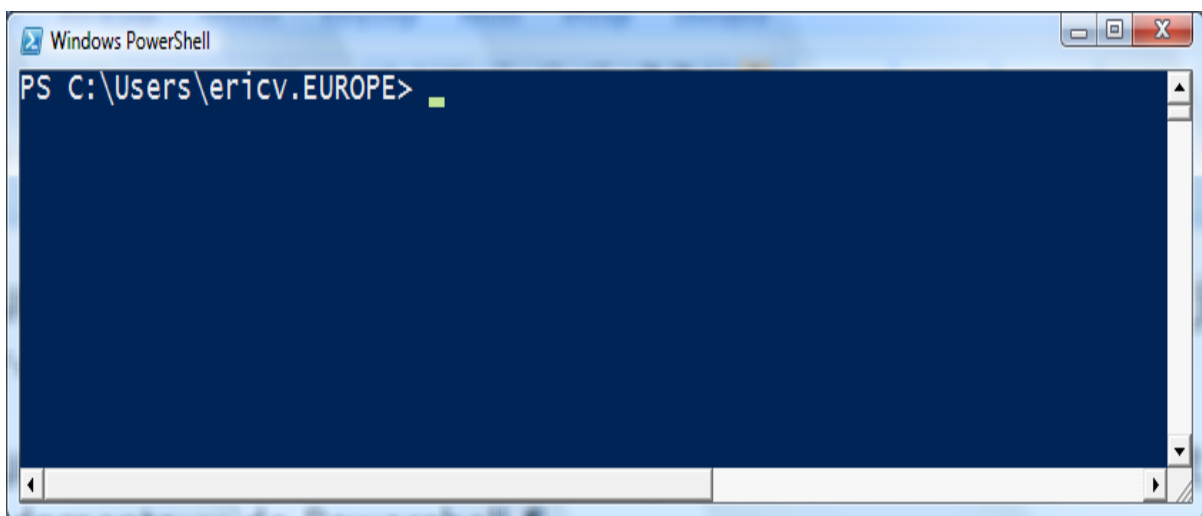
Pour bien comprendre la philosophie de Windows Powershell, il est important de connaître deux principes fondamentaux.

1. Les commandes ne sont pas basées sur du texte mais sur des objets (.NET en l'occurrence)
2. Les commandes sont extensibles.

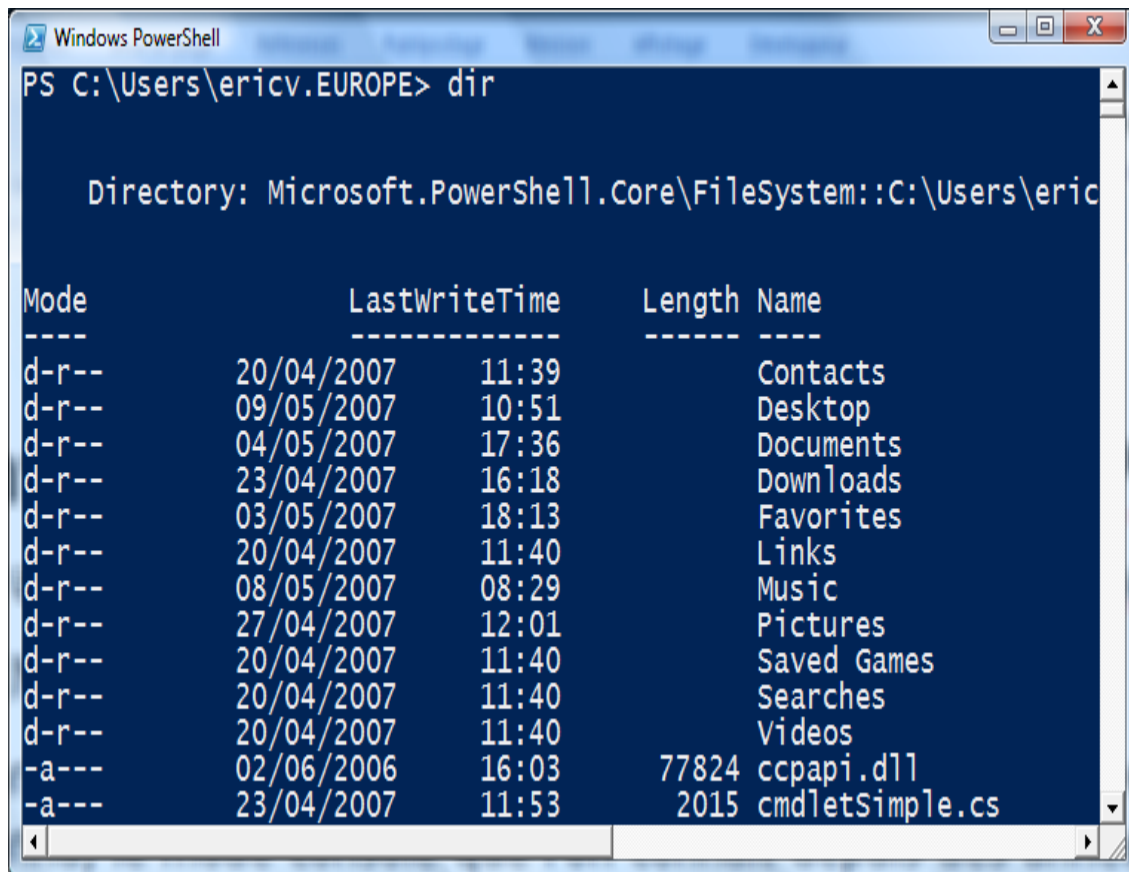
Pour assimiler ces deux principes, nous débuterons par une introduction au langage de script de Windows Powershell, puis nous explorerons des notions plus avancées, telles que la consommation d'objet .NET. Enfin nous terminerons par le développement d'une simple extension à Windows Powershell, connue sous le doux nom de Cmdlet et nous explorerons un exemple plus complexe de cmdlet qui construit et compile à la volée un Service Web.

## Introduction au langage de script de Powershell

Lorsque vous démarrez Windows Powershell, vous vous retrouvez dans un environnement type mode console que l'on connaît depuis des années avec les commandes batch du DOS.



On peut y taper nos commandes favorites comme la commande DIR.



```
Windows PowerShell
PS C:\Users\ericv.EUROPE> dir

Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\ericv.EUROPE

Mode                LastWriteTime         Length Name
----                -
d-r--            20/04/2007         11:39     Contacts
d-r--            09/05/2007         10:51     Desktop
d-r--            04/05/2007         17:36     Documents
d-r--            23/04/2007         16:18     Downloads
d-r--            03/05/2007         18:13     Favorites
d-r--            20/04/2007         11:40     Links
d-r--            08/05/2007          08:29     Music
d-r--            27/04/2007         12:01     Pictures
d-r--            20/04/2007         11:40     Saved Games
d-r--            20/04/2007         11:40     Searches
d-r--            20/04/2007         11:40     Videos
-a---            02/06/2006         16:03    77824 ccpapi.dll
-a---            23/04/2007         11:53     2015 cmdletSimple.cs
```

La commande CD (change directory), RMDir, MD, Del, help, etc. Mais en faisant bien attention vous vous apercevrez de quelques différences notables.

En effet, si vous aviez l'habitude de la commande DIR /W pour afficher en colonne, plutôt qu'en liste, vous découvrirez que cette option ne fonctionne plus. Alors pourquoi ? Tout simplement parce que la commande DIR n'existe plus. En effet, la réelle commande pour afficher les fichiers est *get-childitem* et DIR n'est qu'un alias mappé sur cette dernière. Il en est de même pour toutes les autres commandes. Si vous souhaitez voir tous les alias utilisez la commande "*get-alias*" prévue à cet effet.

Toutes commandes Windows Powershell sera donc construit sous cette forme. Faire l'inventaire de toutes les commandes et leurs utilisations n'est pas le but de cet article, mais si vous débutez, la commande *get-help* est un bon point de départ.

Pour nos démonstrations de scripts, nous allons utiliser une commande fournie avec Windows Powershell, la commande, *get-process*, qui permet de lister les processus en cours.

Alors allons-y.

Dans Windows Powershell, tapez la ligne suivante :

```
PS> get-process
```

Vous devriez voir en colonne, la liste des processus en cours.

Il est possible d'afficher cette liste d'une manière différente en utilisant le caractère "pipe" ainsi que le format liste

```
PS> get-process | format-list
```

Vous pouvez également arrêter n'importe quel processus en combinant plusieurs commandes avec le caractère "pipe"

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

Arrêtons-nous sur cette dernière commande et décortiquons là.

Comme je le disais en introduction, Powershell manipule des objets plutôt que du texte. La commande "Get-process notepad", retourne dans une variable intrinsèque à powershell un objet de type .NET, puis le caractère "pipe" dans notre commande, indique à Powershell de continuer et de passer le résultat de la 1<sup>er</sup> commande à la seconde commande. Tous les processus *notepad* en cours seront alors arrêtés.

Pour vérifier ce que j'avance vous pouvez affecter le résultat de la commande à votre propre variable, ainsi :

```
PS> $MaVariable=get-process notepad
```

Puis tapez :

```
PS> $MaVariable
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
9644	58	0,05	4068	notepad		57	3 1376
57	3	1376	9648	58	0,06	3248	notepad

On remarquera que j'ai plusieurs instances de *notepad* en mémoire

Si je tape la commande suivante :

```
PS> $MaVariable.GetType()
```

```
IsPublic IsSerial
```

```
Name
```

```
BaseType
```

```
-----
```

```
-----
```

```
-----
```

```
True Object[]
```

```
True  
System.Array
```

On s'aperçoit alors que Powershell a construit en mémoire un tableau d'objets (System.Array).

La caractéristique d'un tableau, c'est que l'on peut y faire référence par son index. Pour accéder au premier élément du tableau :

```

PS> $MaVariable[0] Handles    NPM(K)      PM(K)      WS(K)
VM(M)    CPU(s)      Id
ProcessName
-----
-----
          9648      58      0,06    3248 notepad          57      3    1376

```

Je peux alors itérer sur le contenu du tableau avec le langage de script de Windows Powershell qui se rapproche du c#

```

PS> for($i=0; $i -lt $MaVariable.GetLength(0); $i++) {$MaVariable[$i]}

```

On y trouve l'instruction *for* traditionnelle, l'initialisation d'une variable \$i de type System.Int32 et l'opérateur de comparaison "-lt" (inférieur)

\$MaVariable étant un tableau de type System.Array, je peux alors avec la méthode getLength retrouver sa longueur, et le tour est joué.

Mais je pourrais également itérer de manière plus simple comme ceci :

```

PS> foreach ($proc in $mavariabale) {$proc}

```

Comme tout est objet, je peux également avec Windows Powershell référencer chaque propriété de chaque objet grâce à l'opérateur "." (point) comme ceci :

```

PS> foreach ($proc in $mavariabale) {$proc.ID}

```

Comme dans tout langage, il existe d'autres instructions dans sa syntaxe que je ne détaillerai pas ici, mais vous retrouverez bien évidemment :

1. des opérateurs arithmétiques (+, -, \*, /, %)
2. Des opérations sur les tableaux
3. Des opérateurs d'affectations (=, +=, -=, \*=, /=, %=)
4. Des valeurs booléens
5. Des commentaires (#)
6. Des opérateurs de comparaisons (-eq, -ne, -gt, -ge, -lt, -le)
7. Des instructions conditionnelles (if, while)
8. Des séquences d'échappement
9. Des fonctions, etc

Comme tout bon programmeur, vous souhaitez évidemment éviter de retaper à chaque fois la même chose. Pour ce faire, il est possible avec Windows Powershell, de créer ces propres fonctions.

Tapez la commande suivante :

```

PS> notepad MesFonctions.ps1

```

Pour créer un fichier qui se nomme MesFonctions.ps1

Ajoutez-y le code suivant :

```
Function
AfficherProcessus{      param($NomProcessus)      $MaVariable=get-process
$NomProcessus      out-host -
inputObject $MaVariable}
```

Pour vérifier son contenu

```
PS> type MesFonctions.ps1
```

Ensuite il faut charger le fichier

```
PS> . .\MesFonctions.ps1
```

Puis exécuter la fonction

```
PS> AfficherProcessus Notepad
```

Notre fonction accepte des paramètres en entrée, mais on peut également grâce à l'instruction "return" sortir de sa portée.

Enfin on peut développer des filtres qui permettent une action simple exemple.

Ajoutez dans notre fichier MesFonctions.ps1 le code suivant :

```
filter Ralentir ($Tempo=100) {      if
($DernierAffichage)      {      $Duree= ([DateTime]::Now -
$DernierAffichage)
.TotalMilliseconds      if ($Duree -le $Tempo)      {      Start-Sleep -
Milliseconds ($Tempo - $Duree)      }      }
      $DernierAffichage= [DateTime]::Now      $_}
```

Puis rechargez les fonctions

```
PS> . .\MesFonctions.ps1
```

Puis tapez la commande :

```
PS> get-process | ralentir
```

L'affichage des processus se fait plus lentement.

Si vous regardez bien le code du filtre, vous devez y voir plusieurs choses intéressantes :

1. La variable spéciale \$\_ qui correspond à l'objet actif dans notre pipeline (tuyau) en l'occurrence la liste des processus
2. La possibilité d'appeler des méthodes statiques comme [DateTime]::Now d'une classe .NET

C'est ce dernier point que nous allons illustrer dans le paragraphe suivant : *Consommer un composant .NET avec powershell*

# Consommer un composant .NET avec Powershell

Comme vous avez pu le constater, Windows Powershell est un environnement de script puissant et ouvert. Basé sur la plate-forme .NET, il permet d'en tirer la quintessence, comme par exemple la possibilité d'exécuter des méthodes statiques de n'importe quelle librairie .NET.

Dans cette démo, ce que je vous propose, c'est d'aller encore plus loin et d'afficher nos processus dans une Windows Form plutôt que sur la console par défaut.

La première chose à faire est donc de charger la librairie .NET qui contient les Windows Forms dans Windows PowerShell

```
PS> [reflection.assembly]::LoadWithPartialName("System.Windows.Forms")
GAC      Version      Location ---      -
True     v2.0.50727
         C:\Windows\assembly\GAC_MSIL\System.Windows.Forms\2.0.0.0__b77a5c56193
4e089\System.Windows.For...
```

Une fois l'assemblage chargé, on peut alors créer un Windows Form en mémoire à l'aide de l'instruction new-object

```
PS> $MaForm=New-Object System.Windows.Forms.Form
```

La variable \$MaForm est donc désormais une Windows Forms que l'on peut afficher

```
PS> $MaForm.ShowDialog()
```

On peut également lui donner un titre particulier

```
PS> $MaForm.Text="Liste des processus courants"
```

Ensuite nous allons ajouter à notre Windows Form un contrôle qui sera capable d'afficher la liste des processus. Par exemple une DataGridView

```
PS> $MaGrid=New-Object System.Windows.Forms.DataGridViewPS>
$MaGrid.Dock=[System.Windows.Forms.DockStyle]::FillPS>
$MaForm.Controls.Add($MaGrid)
```

La grille est ajoutée à la liste des contrôles de la Windows Form et a la particularité de prendre toute sa surface.

Pour vérifier si cela a fonctionné

```
PS> $MaForm.ShowDialog()
```

Ensuite, nous allons remplir un tableau avec la liste des processus comme suit :

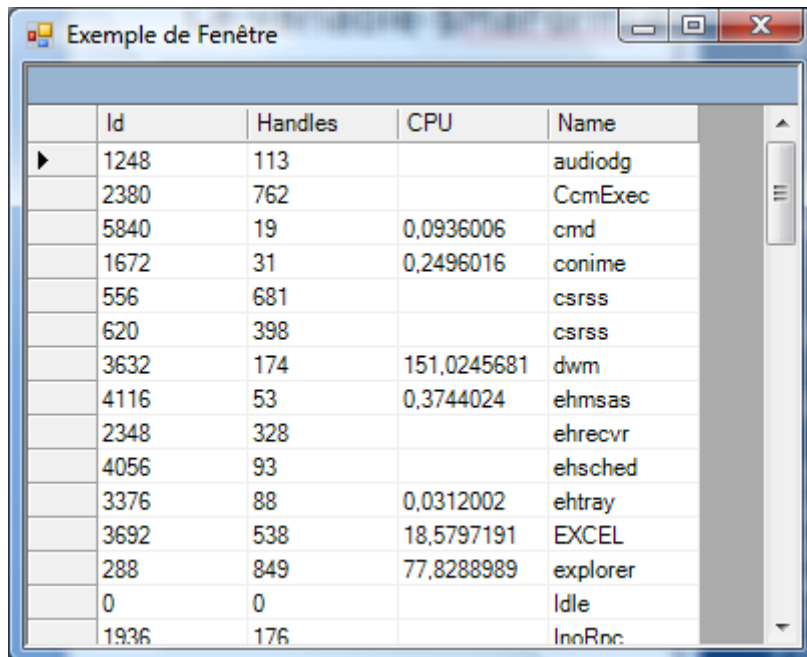
```
PS> $MesProcessus=New-object System.Collections.ArrayListPS>
$MesProcessus.AddRange($(get-process))
```

Le tableau \$MesProcessus doit contenir la liste de nos processus, il suffit alors d'affecter ce tableau à la datagrid

```
PS> $MaGrid.DataSource=$MesProcessus
```

Ensuite on affiche le résultat et le tour est joué

```
PS> $MaForm.ShowDialog()
```



The screenshot shows a Windows PowerShell window titled "Exemple de Fenêtre". Inside the window, a table displays a list of system processes. The table has five columns: Id, Handles, CPU, and Name. The data is as follows:

	Id	Handles	CPU	Name
▶	1248	113		audiodg
	2380	762		CcmExec
	5840	19	0,0936006	cmd
	1672	31	0,2496016	conime
	556	681		csrss
	620	398		csrss
	3632	174	151,0245681	dwm
	4116	53	0,3744024	ehmsas
	2348	328		ehrecvr
	4056	93		ehsched
	3376	88	0,0312002	ehtray
	3692	538	18,5797191	EXCEL
	288	849	77,8288989	explorer
	0	0		Idle
	1936	176		lsass

Comme vous pouvez le constater, Windows Powershell est un environnement puissant, qui peut sans trop d'effort (pour peu que l'on connaisse la plate-forme .NET quand même) être étendu à l'aide de simple script.

Néanmoins, il existe des cas où le scripting n'est pas suffisant, et qu'il faille étendre d'une manière différente les commandes Powershell. Pour ce faire, Powershell propose la possibilité de développer ces propres commandes (cmdlet), c'est ce que nous allons aborder dans le prochain chapitre.

## Développer une extension à powershell (CmdLet)

Les environnements traditionnels proposent généralement des commandes sous forme de plusieurs programmes exécutables, qui manipulent les entrées sorties sous forme de texte. Chaque programme doit fournir ses propres paramètres, sa gestion d'affichage etc ... Par contraste, les commandes Windows Powershell, sont des instances de classes .NET qui dérivent d'une simple classe nommée cmdlet. La commande doit fournir et valider des paramètres en entrée, fournir des détails sur le type d'objet et son formatage et c'est tout. Windows Powershell fait le reste, il parse les paramètres, fait la liaison avec leurs valeurs, formate et affiche la sortie. Comme vous le verrez dans les lignes qui suivent, développer une cmdlet est très simple.

Pour développer une cmdlet peut importe le langage, c'est à votre convenance mais il faut que se soit un langage qui cible la plate-forme .NET, dans notre exercice d'ailleurs, nous utiliserons les 3 langages majeures de Microsoft, Visual Basic 2005, C#, et C++/CLI.

1. Dans Visual Studio 2005 créez un nouveau projet de type ClassLibrary.
2. Ajoutez les références aux assemblages suivants :
  - C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShell\v1.0\System.Management.Automation.dll
  - System.Configuration.Install
3. Incorporez les espaces de noms suivants

VB	C#	C++/CLI
Imports System.ComponentModel	using System.ComponentModel;	using namespace System::ComponentModel;
Imports System.Management.Automation	using System.Management.Automation;	using namespace System::Management::Automation;
Imports System.Collections.Generic	using System.Collections.Generic;	using namespace System::Collections::Generic;
Imports System.Diagnostics	using System.Diagnostics;	using namespace System::Diagnostics;
Imports System.Text	using System.Text;	using namespace System::Text;
Imports System.Configuration.Install	using System.Configuration.Install;	using namespace System::Configuration::Install;

4.Dérivez votre classe de la classe Cmdlet

VB	C#	C++/CLI
<Cmdlet(VerbsCommon.Get, "TestVB")> _	[Cmdlet(VerbsCommon.Get, "TestCS")]	[Cmdlet(VerbsCommon::Get, "TestCPP")]
<b>Public Class TestCommande</b>	<b>public class TestCommande : Cmdlet</b>	<b>public ref class TestCommande : public Cm</b>
Inherits Cmdlet		

Le nom d'une cmdlet consiste en un verbe qui désigne l'action que nous voulons faire et un nom qui indique l'élément. Par exemple "Get" pour le verbe "TestVB" pour le nom.

A noter que nous utilisons l'attribut CmdletAttribute pour identifier la classe comme une cmdlet. Il permet au runtime de Windows Powershell de l'appeler et de l'identifier.

5.Surchargez la méthode ProcessRecord() et le tour est joué

VB	C#	C++/CLI
<b>Protected Overrides Sub ProcessRecord()</b>	<b>protected override void ProcessRecord()</b>	<b>protected : virtual void ProcessRecord() over</b>
<b>WriteObject("Test d'une commande cmdlet</b>		<b>{</b>



développée en VB : " + _message)	{	WriteObject("Test d'une commande cmdlet développée en CPP/CLI : " + _message);
End Sub	WriteObject("Test d'une commande cmdlet développée en CSharp : " + _message);	}
	}	

A ce stade votre cmdlet est terminée, vous pourriez compiler et l'exécuter cela fonctionnerait. Néanmoins pour être tout à fait complète, une cmdlet n'est réellement utile que si on peut lui passer des paramètres.

## 6.Ajout d'un paramètre à notre cmdlet

VB	C#	C++/CLI
Private _message As String	private String _message;	private :String^ _message;
<Parameter(Position:=0)> _	[Parameter (Position = 0)]	public :
Public Property Message() As String	public String Message	[Parameter (Position = 0)]
Get	{	property String^ Message
Return _message	get { return _message; }	{
End Get	set { _message = value; }	String^ get() { return _message; }
Set(ByVal value As String)	}	void set(String^ value) { _message = value; }
_message = value		}
End Set		
End Property		

On déclare une donnée membre privée *\_message* ainsi qu'une propriété nommée *Message*. On marque en suite la propriété avec l'attribut *ParameterAttribut* en lui donnant la position 0 par exemple. (Pour de plus amples informations sur les attributs d'un paramètre, reportez-vous au SDK disponible ici <http://msdn2.microsoft.com/en-us/library/aa830112.aspx>)

Enfin notre cmdlet à besoin d'être enregistrée comme une snapin. Pour ce faire nous allons lui ajouter une classe qui dérive de la classe **PSSnapIn**.

VB	C#	C++/CLI
<RunInstaller(True)> _	[RunInstaller (true)]	[RunInstaller (true)]
Public Class TestCommandeVBPSSnapIn	public class TestCommandeCSPSSnapIn : PSSnapIn	public ref class TestCommandeCPPPSSnapIn : PSSnapIn
Inherits PSSnapIn	{	{
Public Sub New()		public :
MyBase.New()	public TestCommandeCSPSSnapIn()	

End Sub	: base()	property String^ Name
	{	{
	}	virtual String^ get() override
Public Overrides ReadOnly Property Name() As String	public override String Name	{
Get	{	return "TestCommandeCPPSSnapIn"
Return "TestCommandeVBSSnapIn"	get	}
End Get	{	}
End Property	return "TestCommandeCSPSSnapIn";	property String^ Vendor
	}	{
Public Overrides ReadOnly Property Vendor() As String	}	virtual String^ get () override
Get	public override String Vendor	{
Return "Vendor"	{	return "Vendor";
End Get	get	}
End Property	{	}
	return "Vendor";	property String^ VendorResource
	}	{
Public Overrides ReadOnly Property VendorResource() As String	}	virtual String ^ get() override
Get	public override String VendorResource	{
Return "TestCommandeVBSSnapIn"	{	return "TestCommandeCPPSSnapIn";
End Get	get	}
End Property	{	property String^ Description
	return "TestCommandeCSPSSnapIn";	{
	}	virtual String^ get() override
Public Overrides ReadOnly Property Description() As String	}	{
	public override String Description	return "Ceci est une commande Powershell développée en C++/CLI";
Get	{	}
Return "Ceci est une commande Powershell développée en VB"	get	}
End Get	{	property String^ DescriptionResource
	return "Ceci est une commande Powershell développée en CSharp";	{
End Property		virtual String^ get() override

```

    }
    {
        return "DescriptionRessource";
    }
}

Public Overrides ReadOnly Property
DescriptionResource() As String
    Get
        Return "DescriptionRessource"
    End Get
End Property

End Class

public override string
DescriptionResource
{
    get
    {
        return
        "DescriptionRessource";
    }
}
}

```

Il est important de marquer sa classe avec l'attribut *RunInstallerAttribut*, pour que l'utilitaire installutil retrouve la classe qui sera enregistrée en tant que snapin. Toutes les propriétés surchargées ne retournent que des chaînes de caractères décrivant la cmdlet, vous y mettez ce que vous voulez.

Il est temps maintenant de tester notre cmdlet.

### 1. Nous allons l'enregistrer à l'aide de l'utilitaire installutil.exe

```
PS>C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\installutil.Exe
```

```
CHEMIN\TestCS.dllPS>C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\installutil.Exe
```

```
CHEMIN\TestVB.dllPS>C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\installutil.Exe
CHEMIN\TestCPP.dllMicrosoft (R) .NET Framework
Installation utility Version 2.0.50727.312Copyright (c) Microsoft Corporation. All rights
reserved.Running a transacted installation. Beginning the Install phase of the installation.See the
contents of the log file for the C:\Perso\Articles\01 - Powershell Partie I\CPP\TestCPP\debug\TestCPP.dll
assembly's progress.The file is located at C:\Perso\Articles\01 - Powershell
PartieI\CPP\TestCPP\debug\TestCPP.dll.InstallLog.Installing assembly 'C:\Perso\Articles\01 - Powershell
PartieI\CPP\TestCPP\debug\TestCPP.dll'.Affected parameters are: logtoconsole = assemblypath
=C:\Perso\Articles\01 - Powershell Partie I\CPP\TestCPP\debug\TestCPP.dll logfile =
C:\Perso\Articles\01- Powershell Partie I\CPP\TestCPP\debug\TestCPP.InstallLog The Install phase completed
successfully, and the Commit phase is beginning.See the contents of the log file for the
C:\Perso\Articles\01 - Powershell Partie I\CPP\TestCPP\debug\TestCPP.dll assembly's progress.The file is
located at C:\Perso\Articles\01 - Powershell PartieI\CPP\TestCPP\debug\TestCPP.dll.InstallLog.Committing
assembly 'C:\Perso\Articles\01 - Powershell PartieI\CPP\TestCPP\debug\TestCPP.dll'.Affected
parameters are: logtoconsole = assemblypath =C:\Perso\Articles\01 - Powershell Partie
I\CPP\TestCPP\debug\TestCPP.dll logfile = C:\Perso\Articles\01- Powershell Partie
I\CPP\TestCPP\debug\TestCPP.InstallLog The Commit phase completed successfully. The transacted install has
completed.
```

## 2. Ensuite nous allons ajouter le snapin

```
PS> add-pssnapin TestCommandeVBPS> add-pssnapin  
TestCommandeCSPSSnapInPS> add-pssnapin TestCommandeCPPPSnapIn
```

Pour vérifier les snapins enregistrés.

```
PS> get-pssnapinName : TestCommandeCPPPSnapInPSVersion : 1.0Description : Ceci  
est une commande Powershell développée en C++/CLI
```

## 3. Enfin testons notre cmdlet

```
PS> get-testvb $(get-date)Test d'une commande cmdlet développée enVB  
:05/10/2007 14:51:40PS> get-testcs $(get-date)Test  
d'une commande cmdlet développée enCSharp : 05/10/2007 14:52:44PS> get-  
testcpp $(get-date)Test d'une commande cmdlet  
développée enCPP/CLI : 05/10/2007 14:53:16
```

Comme vous pouvez le voir ici, je passe la date courante à ma cmdlet. Vous aurez remarqué la notation `$(get-date)`, qui indique à Windows PowerShell d'exécuter la commande `get-date` avant notre cmdlet et de mettre le résultat dans une variable. Si j'avais tapé à la place `PS> get-testvb get-date`, Windows PowerShell aurait traité `get-date` comme une chaîne de caractères et non pas comme une commande.

A noter également que je n'ai pas précisé le nom du paramètre (grâce à sa position), mais une commande `PS> get-testvb -message $(get-date)` aurait été équivalente.

Néanmoins, comme nous l'avons vu en début d'article, les commandes Windows Powershell peuvent être utilisées au travers de pipeline (de tuyau), c'est-à-dire que nous pouvons passer sur une seule ligne de commande plusieurs commandes séparées par le caractère pipe (`|`) comme cette commande :

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

Dans l'état actuel des choses notre cmdlet ne supporte pas le pipeline, car rien n'indique de lier un paramètre au pipeline de sortie. Pour ce faire, nous allons modifier légèrement notre cmdlet pour prendre en compte le pipeline en ajoutant à l'attribut `Parameter`, l'option `ValueFromPipeline`.

VB	C#	C++/CLI
<code>&lt;Parameter(Position:=0, _  ValueFromPipeline:=True)&gt; _</code>	<code>[Parameter(Position = 0, ValueFromPipeline=true)]</code>	<code>[Parameter(Position = 0, ValueFromPipeline=true)]</code>

Avec cette attribut positionné, il est désormais possible d'exécuter notre cmdlet de la manière suivante.

```
PS> get-date | get-testvbPS> get-date | get-testcsPS> get-date |get-testcpp
```

La commande *get-date* sera exécutée, puis le résultat sera fourni à notre cmdlet.

## Modification du profile pour installer les cmdlets

Avec l'utilitaire installutil.exe vous avez vu comment installer une cmdlet dans Windows Powershell. Néanmoins, à chaque fois que vous quitterez l'environnement, vous devrez relancer les commandes d'installation. Ce qui peut être fastidieux.

Pour éviter ça, Windows Powershell fourni la possibilité de modifier le profile de l'utilisateur. Procédez comme suit.

1. Vérifiez que le fichier profil n'existe pas déjà :

```
PS> test-path $profile
```

Si la commande vous retourne Faux, vous devez en créer en.

2. Création du fichier de profile par défaut

- a. Dans le répertoire *documents* de l'utilisateur, créez le répertoire WindowsPowershell

- b. Créez le fichier Microsoft.PowerShell.Profile.ps1

```
PS> cd $Home\documentsPS> md WindowsProwerShellPS> NOTEPAD  
Microsoft.PowerShell.Profile.ps1
```

3. Ajoutez dans ce fichier la fonction suivante :

```
function pro { notepad $profile }
```

Elle aura pour but de charger le profile automatiquement lorsque vous taperez la commande *pro*

4. Ajoutez-y les commandes suivantes:

```
set-alias installutil  
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\installutil.Exe  
installutil CHEMIN\TestVB.dll  
add-pssnapin TestCommandeVBPSSnapIn  
installutil CHEMIN\TestCS.dll"  
add-pssnapin TestCommandeCSPSSnapIn  
installutil CHEMIN \TestCPP.dll  
add-pssnapin TestCommandeCPPPSSnapIn
```

Ou CHEMIN représente le chemin d'accès à la cmdlet

Désormais, à chaque fois que vous chargerez l'environnement Windows Powershell, l'installation de la cmdlet se lancera.

## Utilisation d'une cmdlet pour construire et consommer un service Web à la volée

Comme nous venons de le voir, développer une cmdlet reste somme toute assez facile, si vous avez de bonne notion de développement avec la plate-forme .NET.

Dans l'exemple que je vous propose maintenant, c'est d'aller un peu plus loin et de pousser l'exercice en utilisant la puissance de Windows Powershell et de la plate-forme .NET, pour la création et la consommation d'un service Web à la volée.

Comme vous devez le savoir maintenant, un Service Web, est indépendant de la plate-forme et du langage utilisé. Cela permet à différent système de pouvoir dialoguer entre eux de manière transparente. Un Service Web, comme son nom l'indique, fonctionne essentiellement aujourd'hui au travers du protocole http (bien que l'on puisse utiliser d'autre protocole tel que TCPIP, SMTP etc..) et il doit fournir un service comme par exemple, un taux de change, la météo, le prix des bananes, j'en passe et des meilleurs.

Que diriez-vous alors de pouvoir à partir de Windows Powershell, exécuter n'importe quel bout de code à la volée qui se trouverait sur des systèmes distant et récupérer des données ? C'est ce que jeme propose de faire avec vous dans les lignes qui suivent.

Pour pouvoir accomplir cette tâche, il m'a fallu développer plusieurs composants que je ne détaillerais pas ici car cela serait hors sujet (vous trouverez de toute manière tout le code avec cet article).

Le projet est décomposé en 3 composants principaux.

- Le composant *WSDLHelper*, qui construit à la volée le proxy du service Web. Pour que notre démonstration fonctionne vous devez d'ailleurs vous assurez que vous possédez bien l'utilitaire "C:\ProgramFiles\Microsoft Visual Studio 8\SDK\v2.0\Bin\wsdl.exe
- Le composant *CompilerHelper*, qui compile à la volée le proxy créé par WSDLHelper
- La cmdlet *PSServiceWeb*, qui permet de lier tout ce beau monde et qui possède la commande get-ServiceWeb  
Cette cmdlet possède plusieurs paramètres que je ne détaille pas ici, mais elle en possède un obligatoire *URL* qui est marqué avec l'option Mandatory=true et qui représente l'url du Service Web à utiliser.

Passons à la démonstration.

### 1.Installez la cmdlet

```
PS> installutil PSServiceWeb.dll
PS> add-pssnapin
CommandServiceWebPSSnapIn
PS> set-alias wsd1
CommandServiceWebPSSnapIn\get-ServiceWeb
```

### 2.Créez un répertoire c:\temp

### 3.Allez sur le site <http://www.xmethods.net> pour sélectionner un Service Web

### 4.Sur ce site allez tout en bas de la page et sélectionnez le service suivant :

A noter qu'il est développé sur une plate-forme Glue.

### 5.Pour construire un proxy, il nous faut un fichier WSDL. Cliquez sur [CurrencyExchange Rate](http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl) et copiez la ligne "http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl"

### 6.Allez dans Windows PowerShell et tapez la ligne suivante

```
PS> $ASSEMBLY=wsdl -url
http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl
```

### 7.Désormais dans la variable \$ASSEMBLY nous avons en mémoire notre proxy au service Web *CurrencyExchangeService* Nous devons alors récupérer ses classes pour pouvoir les instancier.

```
PS> $ASSEMBLY.GetTypes() IsPublic IsSerial Name
BaseType
-----
True      False    CurrencyExchangeService
           System.Web.Services.Protocols.SoapHttpClientProtocol True    True
getRateCompletedEventHandler System.MulticastDelegate
           True    False    getRateCompletedEventArgs
System.ComponentModel.AsyncCompletedEventArgs
```

### 8.La classe qui nous intéresse ici est *CurrencyExchangeService*. Instancions-la.

```
PS> $SW=new-object CurrencyExchangeService
```

### 9.Puis listons les méthodes quelle possède

```
PS> $SW | get-member -membertype method TypeName: CurrencyExchangeService Name
MemberType Definition-----
-----
Method      System.Void Abort()
Method      System.Void add_Disposed()
add_Disposed(EventHandlervalue)add_getRateCompleted Method      System.Void
add_getRateCompleted(getRateCompletedEventHandlerva...BegingetRate
Method      System.IAsyncResult BegingetRate(Stringcountry1, String
country...CancelAsync Method      System.Void CancelAsync(Object
userState)CreateObjRef Method
System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType)Discover
Method      System.Void Discover()Dispose
Method      System.Void Dispose()EndgetRate
Method      System.Single
EndgetRate(IAsyncResultasynResult)Equals Method
```

```
System.Boolean Equals(Object obj) GetHashCode
                                Method      System.Int32 GetHashCode() GetLifetimeService
Method      System.Object GetLifetimeService() getRate
                                Method      System.Single getRate(String country1,String
country2)
```

La méthode qui nous intéresse ici est `getRate()` qui prend deux chaînes de caractères en entrée symbolisant les monnaies à utiliser.

## 10.Exécutons notre service Web

```
PS> $SW.GetRate("EURO","USA")
```

Vous devez obtenir le taux de change entre l'euro et le dollar.

Il est possible que le temps de réponse ne soit pas instantané, n'oubliez pas que vous faite une requête au travers d'internet.

### Remarque :

Par défaut, la cmdlet crée le proxy en csharp dans le répertoire `c:\temp` sous le nom `PowerShell.Proxy.ServiceWeb.dll`, mais vous pouvez changer le langage ainsi que le chemin de la manière suivante:  
PS> \$ASSEMBLY=wsdl -url  
`http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl -path`  
`c:\MonProxy -language vb`Cette commande créera sur la racine le fichier `MonProxy.dll` en Visual Basic

## Conclusion

Comme nous le venons de le voir, Windows Powershell est un environnement qui va simplifier la vie des administrateurs, mais en tant que développeurs, vous allez pouvoir y trouver également votre compte. En effet, si vous êtes plus enclin comme votre serviteur à utiliser des commandes batch pour certains travaux de tous les jours, vous allez pouvoir étendre l'environnement Windows Powershell à votre convenance.

Sachez néanmoins, qu'avec Windows Powershell, nous pouvons encore aller plus loin. En effet dans la seconde partie de notre article, nous aborderons des notions plus approfondie encore comme la gestion de son propre fournisseur Windows Powershell, la possibilité de développer son propre hôte de commande Powershell. L'exercice consistera à permettre à un administrateur de pouvoir exécuter des commandes Windows Powershell d'administration à partir d'Excel et d'utiliser la puissance d'Excel pour formater le résultat.