Utiliser WMI avec PowerShell

Par Laurent Dardenne, le 28 août 2009.

Les débutants trouveront dans ce tutoriel les bases de l'utilisation de WMI sous PowerShell, les autres quelques approfondissements de WMI et techniques avancées.

Un grand merci à shawn12 pour sa relecture et ses corrections.

Les fichiers sources :

ftp://ftp-developpez.com/laurent-dardenne/articles/Windows/PowerShell/Utiliser-WMI-avec-PowerShell/fichiers/Utiliser-WMI-avec-PowerShell.zip

Testé avec PowerShell V1 sous Windows XP et PowerShell V2 sous Windows Seven RC1.

Site de l'auteur : http://laurent-dardenne.developpez.com/

Chapitres

1 Qu’est-ce que WMI ?

WMI, qui est l’acronyme de **W**indows **M**anagement **I**nstrumentation, permet d'accéder et de partager des informations liées à l’infrastructure (matériel et logiciel) d’un réseau d'entreprise.

WMI permet de s’affranchir de la connaissance d’API système pour obtenir des informations de bas niveau telles que la taille de la mémoire ou le type et le nombre de cartes réseaux installées, il autorise également une supervision simplifiée d’événement systèmes.

Si vous ne connaissez pas WMI je vous invite à lire auparavant les tutoriaux suivants traitant des bases de WMI :

http://laurent-dardenne.developpez.com/articles/wmi-p1/

http://dotnet.developpez.com/articles/wmi1/

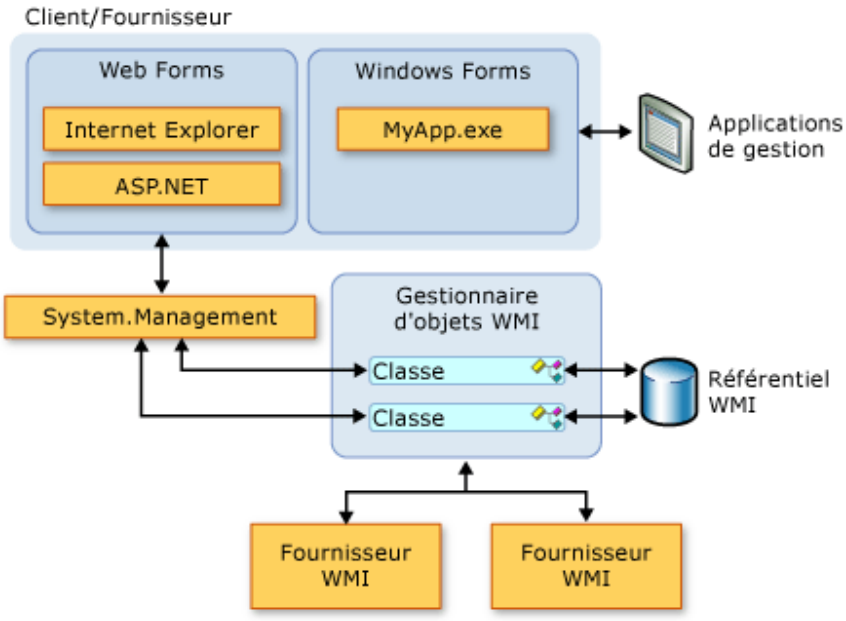
Les API WMI sont basées sur la technologie COM, sous dotnet on y accède via les classes de l’espace de nom **System.Management** :

http://msdn.microsoft.com/fr-fr/library/system.management(VS.80).aspx .

Les limites techniques de PowerShell version 1 ne permettent pas de toutes les utiliser par exemple les appels asynchrones qui nécessitent une gestion d’événements.

Comme dit dans le SDK dotnet (http://msdn.microsoft.com/fr-fr/library/ms257361(VS.80).aspx)

« *L'illustration ci-dessous identifie les trois niveaux WMI et indique comment les couches de l'espace de noms System.Management sont définies dans WMI* :



(c) Microsoft, MSDN

PowerShell, comme d'autres langages de scripting sous Windows, est également un client de WMI et ajoute comme nous allons le voir une couche d'accès supplémentaire.

2 Le cmdlet Get-WMIObject

Sous PowerShell V1 il n'existe qu'un cmdlet lié à WMI, **Get-WMIObject** qui d'après la documentation « *Obtient des instances de classes WMI ou des informations à propos des classes disponibles.* ».

Par défaut il interroge l'espace de nom **root\cimv2** de l'ordinateur local,le nom de l'espace de nom est codé en dur dans le code du cmdlet il n'est pas possible de le paramétrer en modifiant la clé de registre dédiée

Avant tout vous ne devez pas confondre les classes WMI à proprement parlé avec celles de dotnet les encapsulant. De plus tous les objets sont à leur tour adaptés dans PowerShell afin de faciliter leur manipulation. Cette adaptation constitue la couche supplémentaire indiquée précédemment.

Prenons comme exemple la récupération d'informations sur la carte mère d'un ordinateur. Pour cela on passe en argument au cmdlet **Get-WmiObject** le nom de la classe WMI:

```
$ObjetWMI = Get-WmiObject "Win32_MotherboardDevice"
```

Avec cette syntaxe on récupère une ou plusieurs instances de classe WMI. Il n'existe qu'une seule instance de carte mère par ordinateur. Affichons le type de la classe de l'objet récupéré :

```
$ObjetWMI.GetType()
```

```
IsPublic IsSerial Name                BaseType
-------- -------- ----                --------
True     True     ManagementObject    System.Management.ManagementBaseObject
```

Comme vous pouvez le voir, le nom de la classe de l'objet renvoyée par Get-WMIObject est *ManagementObject* encapsulant une instance d'une classe WMI.

Pour ce même objet le nom de type renvoyé par Get-Member :

```
$ObjetWMI|Get-Member
```

```
   TypeName: System.Management.ManagementObject#root\cimv2\Win32_MotherboardDevice

Name        MemberType   Definition
----        ----------   ----------
Reset       Method       System.Management.ManagementBaseObject Reset()
```

est la concaténation du nom de type de l'objet PowerShell et des propriétés *__Namespace* et *__Class* de l'objet WMI :

```
$ObjetWMI.__NAMESPACE
```

```
root\cimv2
```

```
$ObjetWMI.__CLASS
```

```
Win32_MotherboardDevice
```

Pourtant si on consulte la documentation de la classe *ManagementObject*, ces deux propriétés n'y sont pas indiquées :

http://msdn.microsoft.com/fr-fr/library/system.management.managementobject_members(VS.80).aspx

Elles ne sont pas plus indiquées dans la documentation de la classe WMI *Win32_MotherboardDevice* http://msdn.microsoft.com/en-us/library/aa394204(VS.85).aspx.

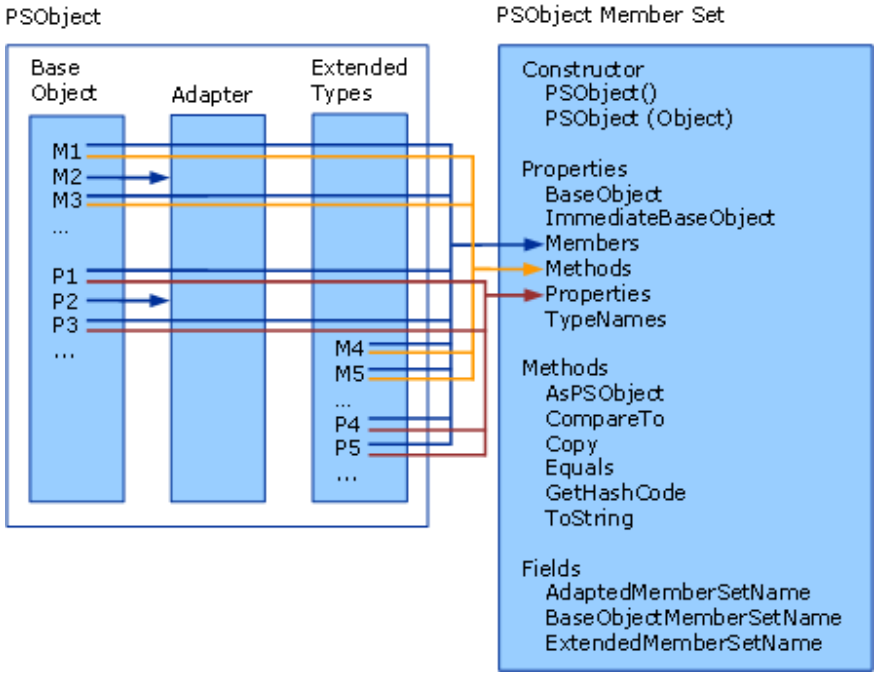
Les propriétés débutant par un double underscore « __ » sont des propriétés systèmes prédéfinies, c'est-à-dire qu'aucune classe ancêtre ne les déclare, mais on les retrouve sur toutes les instances WMI (http://msdn.microsoft.com/en-us/library/aa394584(VS.85).aspx)

Notre objet est donc une suite d'adaptation par le référentiel WMI puis par dotnet et enfin par PowerShell. A l'origine la classe dotnet *ManagementObject* accède aux membres de l'objet WMI par différentes collections :

```
$ObjetWMI.psbase
```

```
Scope          : System.Management.ManagementScope
Path           : \\ MACHINENAME \root\cimv2:Win32_MotherboardDevice.DeviceID="Motherboard"
Options        : System.Management.ObjectGetOptions
ClassPath      : \\MACHINENAME\root\cimv2:Win32_MotherboardDevice
Properties     : {Availability, Caption, ConfigManagerErrorCode,ConfigManagerUserConfig...}
SystemProperties : {__GENUS, __CLASS, __SUPERCLASS, __DYNASTY...}
Qualifiers     : {dynamic, Locale, provider, UUID}
…
```

L'adaptation de PowerShell replace au premier plan certains membres de l'objet WMI afin de faciliter leur accès. Le format d'affichage de certaines classes est réalisé à l'aide du fichier *types.ps1xml*.



Enfin, sachez que l'objet "primaire" WMI est un objet COM basé sur l'interface nommée **IWbemClassObject** (http://msdn.microsoft.com/en-us/library/aa391433(VS.85).aspx).

La documentation de cette interface nous fournira des informations complémentaires, telles que les codes d'erreur renvoyés par ses méthodes.

Vous noterez qu'avec PowerShell il n'y pas de phase de connexion explicite au référentiel WMI cible, PowerShell s'en charge. De plus comme il gère les collections à notre place, il n'est plus nécessaire de préciser qu'on récupère une instance ou une liste d'instances.

2.1.1 À propos de psbase

Certaines propriétés ou collections ne sont accessibles qu'en passant par l'objet dotnet encapsulant l'objet WMI ($ObjetWMI.psbase) et pas par l'objet adapté de PowerShell (($ObjetWMI). Sachez aussi que Get-WmiObject renvoie des copies des instances WMI.

Vous pouvez consulter le tutoriel suivant qui détaille le principe d'adaptation des objets dotnet sous PowerShell :

http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/CreationDeMembresSynthetiquesSousPowerShell/

2.2 Accéder aux détails d'une classe WMI

La définition de la classe WMI est accessible par l'appel suivant :

```
$ObjetWMI=[wmiclass]"\\localhost\root\cimv2:Win32_MotherboardDevice"
$objetWMI.psbase.GetText([System.Management.TextFormat]::MOF)
```

```
[dynamic: ToInstance, provider("CIMWin32"): ToInstance, Locale(1033): ToInstance, UUID("{8502C4BA-5FBB-11D2-AAC1-006008C78BC7}"): ToInstance]
class Win32_MotherboardDevice : CIM_LogicalDevice
{
 [read: ToSubClass, key, Override("DeviceId"): ToSubClass, MappingStrings{"WMI"}: ToSubClass] string DeviceID = NULL;
…
```

Un fichier MOF, **M**anaged **O**bject **F**ormat, est le fichier source contenant la description de la classe WMI. Voir sur le sujet : http://msdn.microsoft.com/en-us/library/aa823192(VS.85).aspx

Rassurez-vous, pour une utilisation courante de WMI sous PowerShell il n'est pas nécessaire de manipuler ce type de fichier.

Notez que dans le chemin d'accès du raccourci [wmiclass], que nous aborderons plus tard, le caractère deux points sépare le nom de l'espace de nom du nom de la classe.

Le résultat peut aussi être renvoyé en XML :

```
$objetWMI.psbase.GetText([System.Management.TextFormat]::CimDtd20)
```

2.2.1 Où trouver des informations sur les classes WMI disponibles ?

WMI Reference : http://msdn.microsoft.com/en-us/library/aa394572(VS.85).aspx

Attention sur certaines versions de serveur, toutes les classes et provider WMI peuvent ne pas être installés par défaut et nécessiter une installation supplémentaire.

2.3 Utilisation des informations d'identification

Il est possible d'authentifier l'utilisateur à l'aide du paramètre *-Credential* :

```
 #Demande de nom et de mot de passe
```

```
$MyCredential=Get-Credential
Get-WmiObject -class "Win32_MotherboardDevice" -credential $MyCredential

 #Demande de mot de passe uniquement
Get-WmiObject -class "Win32_MotherboardDevice" `
 -credential DOMAINE\NomUtilisateur
```

Attention sous PowerShell V1, l'usage de crédential diffèrent entre le poste local et le système distant pose problème.

Pour la classe *ManagementEventWatcher* que nous aborderons par la suite, on doit utiliser la classe *ManagementScope* :

```
$WmiNameSpace="\\.\root\cimv2"
$scope =New-Object System.Management.ManagementScope $WmiNameSpace
$scope.options.Username = $MyCredential.GetNetworkCredential().Username
$scope.options.Password = $MyCredential.GetNetworkCredential().Password
```

3 Les raccourcis WMI

Dans un des exemples précédents la notation **[wmiclass]** est équivalente à l'écriture du code suivant :

```
$Objet=New-Object System.Management.ManagementClass Win32_MotherboardDevice
```

Ce que nous confirme l'affichage suivant :

```
[wmiclass]

IsPublic IsSerial Name                 BaseType
-------- -------- ----                 --------
True     True     ManagementClass      System.Management.ManagementObject
```

Voici la liste des raccourcis WMI disponibles :

[WMI]	Accède à une instance de classe WMI particulière en précisant un chemin unique.
	`$ObjetWMI= [Wmi]'Win32_MotherboardDevice.DeviceID="Motherboard"'`
	Similaire à un appel à **Get-WMIObject** ou au code suivant :
	`$ObjetWMI= New-Object System.Management.ManagementBaseObject `
	` Win32_MotherboardDevice`

[WMICLASS]	Accède à une classe WMI notamment à ces méthodes statiques.
	`$ClassWMI=[wmiclass]"\\localhost\root\cimv2:Win32_MotherboardDevice"`
	Similaire au code suivant :
	`$ClassWMI= New-Object System.Management.ManagementClass `
	` Win32_MotherboardDevice`

[WMISEARCHER]	Exécute une interrogation WMI à l'aide du langage WQL et renvoie le résultat. `([wmiSearcher]'SELECT * FROM Win32_MotherboardDevice where`

	DeviceID="Motherboard"').Get() Similaire au code suivant : $WMISearcher = New-Object System.Management.ManagementObjectSearcher $WMISearcher.Query ='SELECT * FROM Win32_MotherboardDevice where DeviceID="Motherboard"' $WMISearcher.Get()

Notre premier exemple :

```
$ObjetWMI = get-wmiobject "Win32_MotherboardDevice"
```

peut donc être réécrit à l'aide d'un raccourci WMI :

```
$ObjetWMI = [Wmi]'Win32_MotherboardDevice.DeviceID="Motherboard"'
```

Dans le tableau des raccourcis, j'indique qu'ils sont similaires à certaines portions de codes, car comme le montrent les exemples suivants, le résultat diffère dans certains cas :

```
 #il manque la lettre finale D à "Motherboar"
$ObjetWMI = get-wmiobject "Win32_MotherboardDevice" -filter`
"DeviceID='Motherboar'"
$ObjetWMI
```

```
#RAS car $ObjetWMI est à $null
$ObjetWMI = [Wmi]'Win32_MotherboardDevice.DeviceID="Motherboar"'
```

Impossible de convertir la valeur « Win32_MotherboardDevice.DeviceID="Motherboar" » en type « System.Management.ManagementObject ». Erreur : « Non trouvé »

L'objectif est identique, mais **Get-WmiObject** intercepte et traite en interne l'erreur due au fait qu'aucune clé ne correspond à la valeur fournie.

La propriété *DeviceID* est bien une clé, mais si vous recherchez une instance à partir d'un nom de propriété qui ne possède pas le qualificateur ***key***, la cause de l'erreur est différente :

```
$ObjetWMI = [Wmi]'Win32_MotherboardDevice.Name="Carte-mère"'
```

Impossible de convertir la valeur « Win32_MotherboardDevice.Name="Carte-mère" » en type « System.Management.ManagementObject ». Erreur : « Chemin de l'objet non valide »

Note :

Pour déterminer si un objet WMI est une définition de classe ou une instance de classe, on utilisera la propriété *__GENUS* .

La valeur 1 (WBEM_GENUS_CLASS) indique une classe et la valeur 2 (WBEM_GENUS_INSTANCE) indique une instance ou un événement.

3.1 L'unicité des instances WMI

Pour récupérer une seule instance d'une classe donnée, on doit préciser sa clé. WMI étant un référentiel, *c.-à-d.* une base de données, chaque instance possède un identifiant qui peut être une combinaison de clés (précisées lors de la conception de la classe WMI).

Sur le sujet vous pouvez consulter l'entrée suivante sur le blog de l'équipe PowerShell :

http://blogs.msdn.com/powershell/archive/2008/04/15/wmi-object-identifiers-and-keys.aspx

Au sujet des qualificateurs il est dit au chapitre 4.10.2 du tutoriel d'introduction à WMI :

Le qualificateur Key indique que la propriété est une clef de classe et est employé pour identifier des instances uniques d'une ressource gérée dans une collection de ressources identiques.

On peut également retrouver la combinaison de clés d'une instance ainsi :

```
$ObjetWMI.__RELPATH
```

```
Win32_MotherboardDevice.DeviceID="Motherboard"
```

On comprend mieux le message d'erreur *«Chemin de l'objet non valide»*. Un chemin complet d'objet d'instance ajoute le nom et la valeur d'une propriété de la classe à un chemin complet d'objet de classe : **\\Server\Namespace:Class.KeyName="KeyValue"**

```
gwmi Win32_Thread|% {$_.__RELPATH}|Select -first 1
```

```
Win32_Thread.Handle="0",ProcessHandle="0"
```

La recherche d'une instance possédant de multiples clés s'effectue ainsi :

```
gwmi Win32_Thread -filter "Handle='0' and ProcessHandle='0'"
```

3.1.1 Améliorer les performances lors d'une recherche

Lors d'une recherche dans le référentiel WMI, on récupère par défaut l'intégralité des instances d'une classe donnée, il est recommandé d'utiliser le paramètre *–filter* du cmdlet **Get-WmiObject** au lieu de filtrer le résultat sous Powershell :

```
measure-command {gwmi Win32_Thread -filter "Handle='0' and `
ProcessHandle='0'"}
measure-command {gwmi Win32_Thread |Where {$_.Handle -eq 0 –and `
$_.ProcessHandle -eq 0}}
```

Dans le prolongement on peut souhaiter ne récupérer qu'une suite de propriétés d'un objet, on utilisera alors le paramètre *–query* :

```
gwmi -query "SELECT Handle,ProcessHandle FROM Win32_Thread"
 #Ou à l'aide du paramètre -property
gwmi "win32_thread" -property Handle,ProcessHandle
```

Ici comme avec le cmdlet **Select-Object**, les propriétés qui ne sont pas listées ne sont plus accessibles.

```
 #Récupère Handle,ProcessHandle pour un seul thread
gwmi -query "SELECT Handle,ProcessHandle FROM Win32_Thread Where`
Handle='0' and ProcessHandle='0'"
```

Notez que la valeur passée au paramètre *–query* utilise la syntaxe WQL, **W**MI **Q**uery **L**anguage.

Ces améliorations nous sont confirmées par le tableau comparatif suivant, issu de MSDN US :

Méthode/Requête WQL (VBScript)	Octets renvoyés
objSWbemServices.InstancesOf("Win32_Service")	157,398
objSWbemServices.ExecQuery("SELECT * FROM Win32_Service")	156,222
objSWbemServices.ExecQuery("SELECT Name FROM Win32_Service")	86,294
objSWbemServices.ExecQuery("SELECT StartMode FROM Win32_Service")	88,116
objSWbemServices.ExecQuery("SELECT StartMode FROM Win32_Service WHERE State='Running'")	52,546
objSWbemServices.ExecQuery("SELECT StartMode, State FROM Win32_Service WHERE State='Running'")	56,314
objSWbemServices.ExecQuery("SELECT * FROM Win32_Service WHERE Name='WinMgmt'")	27,852
objSWbemServices.Get("Win32_Service.Name='WinMgmt'")	14,860

3.2 Instance transitoire

Certaines classes sont particulières, prenons l'exemple suivant :

```
get-wmiobject "Win32_PingStatus"
```

```
Get-WmiObject : Non pris en charge
```

La classe existe bien, mais comme l'indique sa documentation elle ne supporte pas l'énumération, c'est-à-dire qu'il n'existe pas d'instance persistante pour cette classe.

Seule une interrogation crée et renvoie une instance :

```
$P=Get-WmiObject -query "SELECT * FROM Win32_PingStatus WHERE
Address='127.0.0.1'"
$P
```

L'instance WMI encapsulée dans la variable $P n'existe que du coté du client.

Rares sont les classes ne supportant pas l'énumération : Win32_ClientApplicationSetting.

D'autres n'existent que sur certaines versions de Windows, ici sur Windows Server 2003 uniquement :

```
 #Poste Windows XP
get-wmiobject "Win32_Volume"
```

```
Get-WmiObject : Classe non valide
```

3.3 Méthode statique

Certaines méthodes sont statiques, c'est-à-dire qu'elles ne sont accessibles que par un objet représentant une classe WMI et pas par un objet représentant une instance d'une classe WMI.

Par exemple pour la classe **Win32_Printer** la méthode *AddPrinterConnection* n'est pas accessible à partir d'une instance :

```
$p=gwmi Win32_Printer
$p[0]|gm -membertype *method |select name|Sort name
```

```
Name
----
CancelAllJobs
Pause
PrintTestPage
…
```

Mais bien à partir de la classe :

```
$C=[wmiClass]"Win32_Printer"
$C.psbase.Methods|select name|sort name
```

```
Name
----
AddPrinterConnection
CancelAllJobs
…
```

L'appel se faisant ainsi :

```
$C.AddPrinterConnection("Test")
```

```
__GENUS        : 2
…
ReturnValue    : 1801
```

Ici la valeur de retour de l'appel renvoie 1801 pour *Invalid Printer Name.*

4 Get-WMIobject et le pipeline

L'usage de Get-WmiObject avec le pipeline nécessite une approche particulière, car ce cmdlet ne lit pas les informations sur le pipeline, ce qui fait que la construction suivante n'est pas possible :

```
Import-Csv computers.csv | Get-WmiObject
```

On doit utiliser celle-ci :

```
Import-Csv computers.csv |`
  foreach {
    Get-Wmiobject -computerName $_.ComputerName -Class $_.Class
  }
#Ou encore
Import-Csv Computers.csv |`
 Get-WmiObject -ComputerName {$_.ComputerName} -Class {$_.Class}
```

Issu de : http://blogs.msdn.com/powershell/archive/2006/06/23/643674.aspx

4.1 Inventaires divers sur le référentiel WMI

4.1.1 Tester le statut du service WMI

```
function Test-WinmgmtIsRunning
{
  if ((Get-Service|where {$_.Name -eq "winmgmt"}).Status -eq "Stopped")
```

```
   {Throw "Le service d'infrastructure de gestion Windows (winmgmt) est`
    arrêté.`n`rImpossible de continuer."}
}
```

Dans certains cas le fait de placer en pause ce service, puis d'utiliser PowerShell provoquera quelques erreurs.

4.1.2 Récupérer la liste des espaces de nom

```
gwmi -namespace "root" '__Namespace'|%{ $_.Name}
```

4.1.3 Récupérer la liste des classes d'un espace de nom

```
gwmi -namespace "root\cimv2\Applications\MicrosoftIE" –list
```

4.1.4 Récupérer la liste des providers d'un espace de nom

```
gwmi -namespace "root\cimV2" -class "__Win32Provider"|% {$_.Name}
```

La liste des providers proposés par Microsoft :

http://msdn.microsoft.com/en-us/library/aa394570(VS.85).aspx

4.1.5 Retrouver le nom du provider d'une classe

Le nom du provider se trouve dans la liste des qualificateurs d'une classe :

```
([wmiclass]"root\cimv2\Applications\MicrosoftIE:MicrosoftIE_Summary").`
psbase.Qualifiers|Where {$_.name -eq "provider"}
```

Note :

Vous pouvez retrouver le nom de la DLL l'hébergeant en recherchant dans la registry la valeur de la propriété CLSID, vous devrez au préalable créer un drive sur la ruche HKEY_CLASSES_ROOT :

```
New-PSDrive -Name HKCR -PSProvider Registry -Root HKEY_CLASSES_ROOT
```

4.1.6 Retrouver les noms des classes dérivées

```
([wmiclass]"__InstanceCreationEvent").__Derivation
```

4.1.7 Retrouver la description d'une classe WMI

Le script nommé *Add-WmiHelpFunctions.ps1*, disponible dans les sources, permet de retrouver la description d'une classe ou d'un membre.

Vous pouvez consultez les classes système WMI existantes ici :

http://msdn.microsoft.com/en-us/library/aa394583(VS.85).aspx

4.1.8 Fonction de conversion de nom de chemin de fichier

```
function ConvertTo-WindowsPath([string]$WMIPathName)
```

```
{  #convertit un nom de fichier au format WMI en
   #un nom de fichier au format Windows
    $WMIPathName.Replace('\\','\')
}#ConvertTo-WindowsPath

function ConvertTo-WMIPath([string]$WindowsPathName)
{  #convertit un nom de fichier au format Windows en
   #un nom de fichier au format WMI, les caractères '\' y sont dupliqués.
    $WindowsPathName.Replace("\", "\\")
}#ConvertTo-WMIPath

function ConvertTo-WQLPath([string]$WindowsPathName)
{  #convertit un nom de fichier au format Windows en
   #un nom de fichier au format WQL.
   #
   #   $Path=ConvertTo-WQLPath "C:\Temp"
   #   $Wql=("Targetinstance ISA 'CIM_DirectoryContainsFile' and `
 TargetInstance.GroupComponent='Win32_Directory.Name=`"{0}`"'" -F $Path)

    $WindowsPathName.Replace("\", "\\\\")
}#ConvertTo-WQLPath
```

4.2 Les traitements en masse

Get-WmiObject propose le paramètre *–computername* qui attend un tableau de chaîne de caractères, on peut donc lui indiquer une liste de nom de machine ou d'adresse IP. Ce qui permettra d'exécuter une même requête sur plusieurs machines et simplifiera l'écriture :

```
"localHost","127.0.0.1","127.0.0.0"| Foreach { Get-WmiObject –class `
"Win32_MotherboardDevice" -computername $_ -ErrorAction "Continue"}
```

```
Get-WmiObject -class "Win32_MotherboardDevice" –computername `
"localHost","127.0.0.1","127.0.0.0"
```

Ou encore à l'aide d'un fichier texte où chaque ligne contient un nom de machine :

```
"localHost,127.0.0.1,127.0.0.0".Split(",")>c:\temp\Postes.txt
Get-WmiObject -class "Win32_MotherboardDevice" –computername (Get-Content
C:\Temp\Postes.txt)
```

La dernière adresse IP génère une erreur et pose un problème, une seule erreur arrêtera le traitement, de plus le nom de machine la provoquant n'est pas indiqué.

4.2.1 La commande Get-WmiObject ne génère pas d'erreur en cas de Timeout.

Dans certains cas la phase de connexion à WMI peut être longue bloquant ainsi le traitement d'autres machines d'une même liste. **Get-WmiObject** ne proposant pas de paramètre pour une prise en charge d'un timeout on doit utiliser les classes dotnet :

```
 #Chargement préalable
[void][reflection.assembly]::loadwithpartialname("System.Management")

Function Get-Wmi ([string]$RequestWQL, [string]$Scope = "root\cimv2")
{ # Auteur Mephisto, sur PowerShell-Scripting.com
   #Requete
  $query = New-Object System.Management.WqlObjectQuery $RequestWQL
   #Scope defaut:'root\cimv2'  à distance:'\\machine\root\cimv2'
  $objScope = New-Object System.Management.ManagementScope $Scope
   #Objet de recherche
  $searcher = New-Object System.Management.ManagementObjectSearcher `
              $objScope,$query
  $searcher.Options.Timeout = New-Object `
                                timespan([Int32]0, [Int32]1, [Int32]0)
  return $searcher.Get()
}
```

Ce script est à améliorer, notamment sur la gestion des exceptions que chaque classe dotnet utilisées peut renvoyer.

4.3 La gestion des erreurs

L'interrogation avec un nom de machine ou une adresse IP erronée génère l'erreur suivante :

```
Get-WmiObject -class "Win32_MotherboardDevice" -computername 127.0.0.0
Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT : 0x800706BA)
```

WMI utilisant DCOM pour accéder aux machines distantes, cette erreur (*Remote Procedure Call*) pour une adresse valide peut être due à un problème dans la configuration DCOM ou un problème réseau.

On peut déjà éviter la propagation et l'affichage de l'exception en utilisant la variable prédéfinie **$ErrorActionPreference** de portée globale :

```
$ErrorActionPreference
$ErrorActionPreference = "SilentlyContinue"
$error.clear()
Get-WmiObject -class "Win32_MotherboardDevice" -computername 127.0.0.0
$error
```

Ou utiliser le paramètre *–ErrorAction*, de portée locale, commun aux cmdlets :

```
Get-WmiObject -class "Win32_MotherboardDevice" -computername 127.0.0.0 `
-ErrorAction "SilentlyContinue"
```

Si on souhaite dissocier l'erreur de cet appel de cmdlet de la liste globale des erreurs, on utilisera un autre paramètre commun à tous les cmdlets, *-ErrorVariable* :

```
$error.clear()
Get-WmiObject -class "Win32_MotherboardDevice" -computername 127.0.0.0 `
-EA "SilentlyContinue" –ErrorVariable ErreurWMI
$error # cette liste est renseignée
$ErreurWMI # Cette variable aussi
```

La syntaxe suivante ajoute plusieurs erreurs dans la variable indiquée :

```
Get-WmiObject -class "Win32_MotherboardDevice" -computername 127.0.0.0 `
-EA "SilentlyContinue" –ErrorVariable +ErreurWMI
$ErreurWMI
```

Si cette variable n'existe pas, elle est créée par le cmdlet. Pour une gestion plus fine des erreurs lors d'opérations en masse, on utilisera le pipeline ou une boucle :

```
$ErrorActionPreference = "Continue"
$error.clear()
Get-Content c:\temp\Postes.txt |`
  ForEach-Object {
     Get-WmiObject -class "Win32_MotherboardDevice" –computername $_
  }
```

Ainsi, le traitement de toutes les machines devient possible, reste à gérer les erreurs.

Comme l'indique ce post : http://www.vistax64.com/powershell/180262-how-work-error-interception.html , l'affectation de la valeur « Stop » au paramètre *–ErrorAction* permet de récupérer les erreurs non critiques, l'erreur est affichée, mais ne provoque pas un arrêt du script. Ainsi, on peut traiter l'erreur courante :

```
$ErrorActionPreference = "Continue"
$error.clear()
Get-Content c:\temp\Postes.txt |`
  ForEach-Object {
    trap [Management.Automation.ActionPreferenceStopException]
     { Write-Warning "[$Current] Exception"
       Write-Host $ErreurWMI[0].Message –fore darkgray
       continue
     }
  $Current=$_
   Get-WmiObject -class "Win32_MotherboardDevice" –computername $Current `
      -EA "Stop" –EV ErreurWMI
  }
```

On pourrait ajouter un test de présence de la machine sur le réseau, dans ce cas vous pouvez utiliser le script suivant proposé sur le blog de Janel qui utilise des classes dotnet et pas WMI :

http://janel.spaces.live.com/blog/cns!9B5AA3F6FA0088C2!180.entry

4.4 Utiliser SWbemLastError

```
$ErreurWMI = new-object -comobject "WbemScripting.SWbemLastError"
```

New-Object : La création d'une instance du composant COM avec le CLSID {C2FEEEAC-CFCD-11D1-8B05-00600806D9B6} à partir de IClassFactory a échoué en raison de l'erreur suivante : 80004005.

Dans un contexte où il n'y a pas d'erreur WMI la création de cet objet COM provoquera toujours cette erreur comme l'indique le SDK Win32 :

"You can create an SWbemLastError error object to inspect the extended error information that is associated with a previous method call. If error information is not available, an attempt to create an error object will fail. If the call succeeds and an error object returns, the status of the object is reset."

De plus, l'usage couplé de PowerShell et d'une seule API de scripting WMI ne semble pas possible suite à des problèmes de conception, bien qu'effectuer une tâche simple uniquement à l'aide API de scripting WMI soit possible sous PowerShell.

En cas d'erreur WMI, PowerShell émet une exception de type **ManagementException :**

http://msdn.microsoft.com/en-us/library/system.management.managementexception_members(VS.80).aspx

Voir aussi le lien suivant à propos de la description des numéros d'erreur que l'on peut retrouver dans les fichiers de log de WMI présent dans le répertoire C:\WINDOWS\system32\wbem\Logs:

Détail des fichiers de log : http://msdn.microsoft.com/en-us/library/aa827355(VS.85).aspx

WMI Error Constants : http://msdn.microsoft.com/en-us/library/aa394559.aspx

Le lien suivant énumère tous les codes d'erreur WMI actuellement définis sous dotnet :

http://msdn.microsoft.com/fr-fr/library/system.management.managementstatus(VS.80).aspx

5 Créer une instance

Jusqu'à maintenant nous n'avons fait que lire le référentiel WMI afin de récupérer des objets (des instances de classes ou des classes). Comme nous l'avons vu la création d'objets WMI est possible à partir d'un client, par exemple un process :

```
 #Récupére la classe
$ClassProcess= [wmiclass]"Win32_Process"
 #Crée et renvoi une instance de la classe Win32_Process
$Process= $ClassProcess.CreateInstance()
 #Affiche le nouvel objet process
$Process
```

Il nous reste à renseigner les champs de notre objet :

```
$Process.CommandLine = "c:\windows\System32\Cmd.exe"
```

Puis à créer l'instance dans le référentiel, pour le moment elle est créée sur le poste uniquement en local, il reste à mettre à jour le référentiel WMI :

```
$Process.Put()
```

Malheureusement, on obtient l'erreur suivante

```
Exception lors de l'appel de « Put » avec « 0 » argument(s) : « Exception lors de l'appel de « Put » avec « 0 » argument(s) : « Le fournisseur ne prend pas en charge l'opération tentée »
```

Le détail de l'erreur nous indique que cette opération n'est pas supportée par le provider WMI :

```
Resolve-Error #Script disponible sur le blog de MS-PowerShell
```

```
…
ErrorInformation  :  System.Management.ManagementBaseObject
ErrorCode         :  ProviderNotCapable
Message           :  Le fournisseur ne prend pas en charge l'opération tentée
```

Vérifions si la classe WMI propose d'autres méthodes :

```
$ClassProcess.psbase.methods|fl name
```

```
Name : Create
Name : Terminate
Name : GetOwner
Name : GetOwnerSid
Name : SetPriority
Name : AttachDebugger
```

Essayons la méthode *Create* mais cette fois-ci en l'appelant à partir du nom de classe et pas sur l'instance $Process :

```
$processID=$ClassProcess.Create("c:\windows\System32\Cmd.exe")
 #Sur une machine distante
$processID=([WMICLASS]"\\$Distant\ROOT\CIMV2:win32_process").Create(
"notepad.exe")
```

Cela fonctionne.

Essayons l'appel de la méthode *Put* sur une autre classe WMI :

```
$VarEnvironment= ([wmiclass]"Win32_Environment").CreateInstance()
$VarEnvironment.UserName = [System.Environment]::UserName
$VarEnvironment.Name = "VariableWMI"
$VarEnvironment.VariableValue = "Variable d'environnement créée avec WMI "
$VarEnvironment.Put()
```

Pour informations cette nouvelle variable n'est pas insérée dans les données du provider des variables d'environnement de PowerShell (*env:)*.

Notez que l'on doit renseigner au minimum les clés de l'instance :

```
$V=gwmi "Win32_Environment"
$V[0].__relpath
```

```
Win32_Environment.Name="ComSpec",UserName="<SYSTEM>"
```

Pour cette classe, l'appel à *Put* réussit, on peut en déduire que toutes les instances créées en local ne peuvent être insérées dans le référentiel WMI. La raison en est un choix de conception indiqué dans la liste des qualificateurs de la classe.

Savoir si une classe peut être créée, supprimée, modifiée :

```
([wmiclass]"Win32_Environment").psbase.Qualifiers|? {
 $_.name -match "^Supports"}| Select name
```

```
Name
----
SupportsCreate
SupportsDelete
SupportsUpdate
```

Pour connaître le mode de création d'une instance d'une classe :

```
"Win32_Process","Win32_Environment"|`
 Foreach {
  ([wmiclass]"$_").psbase.Qualifiers| `
  Where {$_.name -match "By$"}|`
  Select Name,value
 }
```

```
Name                 Value
----                 -----
CreateBy             Create          # Win32_Process
DeleteBy             DeleteInstance
CreateBy             PutInstance     # Win32_Environment
DeleteBy             DeleteInstance
```

Vous trouverez à l'adresse suivante la liste des qualificateurs possibles :

http://msdn.microsoft.com/en-us/library/aa394571(VS.85).aspx

Ici la liste des qualificateurs standards (createBy, supportsCreate,…) :

http://msdn.microsoft.com/en-us/library/aa393650(VS.85).aspx

La méthode Put() provoque quelques bugs sous PowerShell V1:

http://blogs.msdn.com/powershell/archive/2008/08/12/some-wmi-instances-can-have-their-first-method-call-fail-and-get-member-not-work-in-powershell-v1.aspx

Sur certaines plateformes Windows vous devez appeler plusieurs fois cette méthode avant qu'elle réussisse, voir MS-Connect pour le détail.

5.1 Mettre à jour un objet dans le référentiel WMI

On utilisera également la méthode Put(), si toutefois :

- la classe le permet,
- les propriétés sont en lecture/écriture et
- que le compte utilisé possède les droits nécessaires pour mettre à jour le référentiel.

```
 #  *** Dans une nouvelle session PS  ***
$Filtre="Name=`"VariableWMI`" and
UserName=`"$env:ComputerName\\$Env:UserName`""
$Objet=gwmi Win32_Environment -filter $Filtre
$Objet.VariableValue="Modification"
$Objet.Put()
gwmi Win32_Environment -filter $Filtre
```

Dans quelque cas on doit spécifier le mode d'accès de la manière suivante :

```
$service= $S=gwmi Win32_Service -filter 'name="winmgmt"'
$service.State="Paused"
$PutOptions = New-Object System.Management.PutOptions
$PutOptions.Type = [System.Management.PutType]::UpdateOnly
$service.Put($PutOptions)
```

Sous PowerShell v1, malgré l'erreur le service est bien mis en pause :

```
Get-Service "winmgmt"
Set-Service "winmgmt" -status running
```

5.2 Supprimer une instance dans le référentiel WMI

On utilisera la méthode *Delete* pour supprimer une instance dans le référentiel :

```
$Filtre="Name=`"VariableWMI`" and
UserName=`"$env:ComputerName\\$Env:UserName`""
$Objet=gwmi Win32_Environment -filter $Filtre
$Objet.Delete()
```

La suppression de cette variable d'environnement est effective et est bien propagée dans les autres sessions PowerShell actives.

6 Modifier les privilèges de sécurité

Pour certaines méthodes on doit activer les droits nécessaires à leur exécution :

```
$OS=GWMI Win32_OperatingSystem
$OS.win32Shutdown(1,$null)
```

```
Exception lors de l'appel de « Win32Shutdown » : « Privilège non maintenu.  »
```

```
Resolve-Error
```

```
…
ErrorInformation : System.Management.ManagementBaseObject
ErrorCode        : PrivilegeNotHeld
Message          : Privilège non maintenu.
…
```

Pour que l'appel réussisse on doit activer les privilèges ainsi :

```
$OS.PSbase.Scope.Options.EnablePrivileges = $true
```

Voyons ce que nous indique la documentation dotnet à propos de cette erreur :

PrivilegeNotHeld	L'opération a échoué parce que le client n'avait pas le privilège de sécurité nécessaire.

Il faut tout de même que le compte en question puisse activer ce privilège, cette propriété ne donne pas plus de droits que ceux autorisés pour ce compte. C'est l'appel de la méthode qui nécessite d'activer ce droit, comme le précise la documentation de cette propriété :

« Obtient ou définit une valeur indiquant si les privilèges utilisateur doivent être activés pour l'opération de connexion. »

On peut retrouver les droits nécessaires à l'exécution d'une méthode par son qualificateur *privileges* :

```
$OS=[wmiclass]"Win32_OperatingSystem"
$OS.psbase.Methods["Win32Shutdown"].qualifiers["privileges"]
```

```
Name            : privileges
Value           : {SeShutdownPrivilege}
```

7 Interrogation semi-synchrone

La classe ManagementObjectSearcher permet d'utiliser ce mode :

```
function CallSemiSynchrone
([switch]$ReturnImmediately,[switch]$Rewindable) {
  $WMISearcher = New-Object System.Management.ManagementObjectSearcher
  $WMISearcher.Query ="ASSOCIATORS OF {Win32_Directory.Name='C:\Temp'}
Where ResultClass = CIM_DataFile"
  $WMISearcher.options.ReturnImmediately = $ReturnImmediately
  $WMISearcher.options.Rewindable=$Rewindable
  $Avant=get-date
   $F=$WMISearcher.Get()
  $Aprés=get-date
  write-host "Durée de l'appel à Get()=$(($Aprés-$Avant).Ticks)"

  Write-Host "Première lecture `$F.count=$($F.count)"
  Write-Host "Seconde lecture `$F.count=$($F.count)"
}

CallSemiSynchrone #$false $false
CallSemiSynChrone -ReturnImmediately #$true $false
CallSemiSynchrone -Rewindable #$false $true
CallSemiSynChrone -ReturnImmediately -Rewindable #$true $true
```

On constate que

- pour le premier appel, la seconde lecture du tableau $F renvoi zéro éléments car la propriété *Rewindable* est à $False.

- pour le second appel, la seconde lecture renvoie également zéro éléments et le temps d'appel de la méthode Get() est très inférieure au premier appel, car la propriété *ReturnImmediately* est à $True.

Si Rewindable à $false la collection ne peut être énumérée qu'une seule fois.

Si on remplace les deux lignes de lecture $F.Count par deux lignes :

```
$F|Get-Member
```

Le second appel à Get-Member provoquera l'erreur suivante :

```
Une erreur s'est produite lors de l'énumération parmi une collection : Un objet COM qui a été séparé de son RCW sous-jacent ne peut pas être utilisé.
```

ou en Anglais :

```
COM object that has been separated from its underlying RCW can not be used
```

L'objet sous-jacent est libéré automatiquement.

Si ReturnImmediately est $true, l'appel retourne immédiatement. La récupération effective des résultats se produit lorsque la collection résultante est parcourue.

8 Le type de donnée Date sous WMI

WMI peut renvoyer des propriétés contenant une date dans une chaîne de caractères au format suivant :

yyyymmddHHMMSS.mmmmmmsUUU (CIM DATETIME)

Par exemple, '20050820151418.734375-000' est équivalent à '20/08/2005 17:14:18' tout en tenant compte du fuseau horaire.

```
$DateWmi=(gwmi Win32_OperatingSystem).LocalDatetime
20090529085300.375000+120
```

Pour obtenir une date dotnet on peut utiliser un objet COM provenant des API de scripting WMI:

```
$DT = new-object -comobject "WbemScripting.SWbemDateTime"
$DT
$dt.SetVarDate($DateWmi)
$dt
$DateDotNET=$dt.GetVarDate()
$DateDotNET
```

Cette classe permet également de traiter des dates de fichier WMI ou de construire un intervalle.

Comme ce besoin est récurrent, les concepteurs de PowerShell nous ont facilité le codage en ajoutant deux membres (scriptmethod) pour chaque instance WMI :

```
$O=Get-WmiObject -class Win32_OperatingSystem
 #Date WMI vers DateTime
$o.ConvertToDateTime($o.LocalDatetime)
 #DateTime vers Date WMI
$o.ConvertFromDateTime([datetime]::now)
```

La définition des membres nous indique que le Framework propose la classe **System.Management.ManagementDateTimeConverter** qui est spécialisée dans ces conversions :

```
$o.ConvertFromDateTime.script
[System.Management.ManagementDateTimeConverter]::ToDmtfDateTime($args[0])
$o.ConvertToDateTime.script
[System.Management.ManagementDateTimeConverter]::ToDateTime($args[0])
```

Note : Pour les curieux vous pouvez lire, dans la BCL, le code de ces méthodes à l'aide de l'outil Reflector, vous serez peut être surprit par la manière de décoder ce format de date.

8.1 Les requêtes WQl

WQL-Supported Date Formats :

http://msdn.microsoft.com/en-us/library/aa394607(VS.85).aspx

8.2 Les compteurs de performance

Sous .NET les actualiseurs (SWBemRefresher), couplés aux classes Win32_Perf*, ne sont pas pris en charge. On peut rafraîchir un élément par l'appel à la méthode Get() mais pas une liste complète d'élément :

```
 #Une instance du process Notepad doit exister
$C=gwmi Win32_PerfFormattedData_PerfProc_Process -filter "name='notepad'"
$C.ElapsedTime
Start-Sleep 3
$C.Psbase.Get()
$C.ElapsedTime
```

J'ai joint dans le fichier contenant les codes sources, un exemplaire de la revue gratuite, *Windows Administration in Realtime*(**http://nexus.realtimepublishers.com/rtwa.php**) contenant un article sur la classe dotnet **System.Diagnostics.PerformanceCounter.**

Voir aussi l'exemple d'utilisation de l'objet SWBemRefresher dans le fichier suivant :

Something About Scripting - Stopping Services in a Specific Order.mht

La version 2 de PowerShell proposera le cmdlet Get-Counter à partir de Windows Seven.

Voir aussi :

Limitations of WMI in .NET Framework

http://msdn.microsoft.com/fr-fr/library/ms186136(VS.80).aspx

9 Associations et références

Les classes WMI que nous avons vu jusqu'ici contenaient des informations sur des éléments du système, il existe un autre type de classe, la classe d'association qui définit les relations entre deux classes, par exemple entre un disque et une partition, **Win32_DiskDriveToDiskPartition**.

Une instance WMI peut donc être reliée à une ou plusieurs instances d'une association.

Un disque (**Win32_DiskDrive)** peut avoir plusieurs partitions (**Win32_DiskPartition)** et peut être relié à plusieurs classes d'associations :

- **Win32_PnPDevice,** associe un device avec la fonction qu'il remplit,
- **Win32_SystemDevices,** associe un ordinateur à un device logique, etc.

Vous pouvez identifier une classe d'association en examinant le qualificateur *Association* de la classe.

```
([wmiclass]"Win32_DiskDriveToDiskPartition").psbase.`
qualifiers["Association"]
```

Avant de continuer consultons sur MSDN la définition de cette classe :

```
class Win32_DiskDriveToDiskPartition : CIM_MediaPresent
{
  Win32_DiskDrive     REF Antecedent;
  Win32_DiskPartition REF Dependent;
};
```

Le mot clé ***REF*** indique que la propriété est une référence, c'est-à-dire un pointeur vers une autre instance.

Une référence définit le rôle que chaque objet joue dans le contexte d'une association.

Ici la propriété *Antecedent* référence une instance représentant le disque où la partition existe.

La propriété *Dependent* référence une instance représentant la partition résidant sur l'unité de disque.

À partir d'une instance de la classe **Win32_DiskDriveToDiskPartition** on peut donc accéder soit au disque soit à la partition. Bien que le plus souvent le parcours des données débute d'un disque. Voyons cela autour d'un exemple :

```
 #Récupère une seule instance WMI représentant un disque
$d=gwmi Win32_DiskDrive|select -first 1
```

Récupérons les instances d'associations, il s'agit de références :

```
$d.psbase.GetRelationships()
$d.psbase.GetRelationships()|% {$_.__class}
```

```
Win32_PnPDevice
Win32_DiskDrivePhysicalMedia
Win32_SystemDevices
Win32_DiskDriveToDiskPartition
```

Notez que le résultat renvoyé peut varier selon la configuration de votre matériel.

Récupérons les instances des objets associés, il s'agit cette fois-ci des objets référencés. Les instances qui sont retrouvées sont désignées sous le nom de points finaux/terminaison

(endpoints). Chaque point final retourné l'est autant de fois qu'il y a d'associations entre lui et l'objet source (un disque peut avoir plusieurs partitions) :

```
$d.psbase.GetRelated()
$d.psbase.GetRelated()|% {$_.__class}
```

```
Win32_PnPEntity
Win32_PhysicalMedia
Win32_ComputerSystem
Win32_DiskPartition
```

Affichons maintenant les instances de la classe **Win32_DiskDriveToDiskPartition** de notre disque. Pour une meilleure lecture du résultat contentons-nous d'un affichage au format MOF :

```
$p=$d.psbase.GetRelationships()|`
   Where {$_.__class -eq "Win32_DiskDriveToDiskPartition"}
$p|%{$_.psbase.GetText([System.Management.TextFormat]::MOF)}
 #Ou
$p=$d.psbase.GetRelationships("Win32_DiskDriveToDiskPartition")
 #Attention ici $p est du type ManagementObjectCollection
$p|%{$_.psbase.GetText([System.Management.TextFormat]::MOF)}
```

```
  #Ce disque possède une seule partition
instance of Win32_DiskDriveToDiskPartition
{
      Antecedent = "\\\\PC\\root\\cimv2:Win32_DiskDrive.DeviceID=\"\\\\.\\PHYSICALDRIVE1\"";
      Dependent = "\\\\PC\\root\\cimv2:Win32_DiskPartition.DeviceID=\"Disk #1, Partition #0\"";
};
```

Chacune de ces propriétés contient le chemin complet de l'instance concernée :

```
 #Lit l'instance dans le référentiel.
 #puis affiche le disque associé à cette partition
[wmi]($p.Antecedent)
```

A son tour une instance d'association pour un objet donné, peut être associée à d'autre classe :

```
$Disques=gwmi Win32_DiskDrive
$partitions=$Disques|%{$_.psbase.GetRelated("Win32_DiskPartition")}
$DisquesLogiques=$Partitions|%{$_.psbase.GetRelated("Win32_LogicalDisk")}
```

On explore donc les relations suivantes :

Win32_DiskDrive->Win32_DiskPartition->Win32_LogicalDisk

Si on souhaite structurer ces informations, on utilisera des hashtables imbriquées comme le montre le script *Get-DiskPartition.ps1* disponible dans les sources.

Pour plus de détail sur les hashtables imbriquées consultez le chapitre *6.3 Imbrication de structure* du tutoriel suivant :

http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/StructuresDeDonneesSousPowerShell/

Notez que l'outil *WMI CIM Studio* permet de visualiser et de parcourir les associations d'une classe.

9.1.1 À propos des collections limitant l'accès à leurs éléments

De nombreuses méthodes des classes dotnet, encapsulant les appels à WMI, renvoient des collections de type **ManagementObjectCollection** :

```
$d=gwmi Win32_DiskDrive|select -first 1
$p=$d.psbase.GetRelationships("Win32_DiskDriveToDiskPartition")
$p.GetType()
```

```
IsPublic IsSerial Name                           BaseType
-------- -------- ----                           --------
True     False    ManagementObjectCollection     System.Object
```

L'indexation sur ce type de classe n'est pas possible, car elle n'implémente pas l'interface **IList** qui met en œuvre l'indexation sur une collection :

```
$p.GetType().GetInterfaces()
```

```
IsPublic IsSerial Name        BaseType
-------- -------- ----        --------
True     False    ICollection
True     False    IEnumerable
True     False    IDisposable
```

```
[system.array].GetInterfaces()
```

```
IsPublic IsSerial Name        BaseType
-------- -------- ----        --------
True     False    ICloneable
True     False    IList
True     False    ICollection
True     False    IEnumerable
```

On utilisera un énumérateur pour parcourir la collection ou on recopiera cette collection dans un tableau :

```
  #crée un tableau de la taille de la collection
$Associations=new-object Object[] $p.count
 #On recopie la collection dans le tableau à partir de l'index 0
$p.CopyTo($Associations,0)
```

9.2 Retrouver les associations et les références d'une instance avec WQL

On peut aussi utiliser WQL pour effectuer les mêmes opérations.

Retrouver les instances d'associations d'un objet WMI :

```
$d=gwmi Win32_DiskDrive|select -first 1
$d.psbase.GetRelated()
 #Requete WQL équivalente
$RequestDisk="Associators of {Win32_DiskDrive.DeviceID='$($d.DeviceID)'}"
([wmiSearcher] ($RequestDisk)).get()
 #ou encore
```

```
gwmi -query "Associators OF {Win32_DiskDrive.DeviceID='$($d.DeviceID)'}"
```

Retrouver les références d'associations d'un objet WMI :

```
$d.psbase.GetRelationships()
  #Requete WQL équivalente
$RequestDisk="References OF {Win32_DiskDrive.DeviceID='$($d.DeviceID)'}"
([wmiSearcher] ($RequestDisk)).get()
#ou encore
gwmi -query "References OF {Win32_DiskDrive.DeviceID='$($d.DeviceID)'}"
```

9.2.1 Affiner la recherche à l'aide de mot-clé WQL

Les instructions WQL proposent de nombreuses options détaillées ici :

WQL (SQL for WMI) : http://msdn.microsoft.com/en-us/library/aa394606(VS.85).aspx

Grammaire de WQL : http://msdn.microsoft.com/en-us/library/cc250720(PROT.10).aspx

Quelques exemples :

Recherche, à partir du premier disque physique, la définition des classes associées à la classe **Win32_DiskDrive** :

```
$Requete="Associators of {Win32_DiskDrive.DeviceID='\\.\PHYSICALDRIVE0'} "
$Filtre="WHERE ClassDefsOnly"
([wmiSearcher]("$Requete$Filtre")).Get()
```

```
Win32_PnPEntity          Win32_ComputerSystem
Win32_PhysicalMedia      Win32_DiskPartition
```

Filtre les associations d'après un nom de rôle :

```
$Filtre="WHERE Role = Antecedent ClassDefsOnly"
([wmiSearcher]("$Requete$Filtre")).Get()
```

```
Win32_DiskDriveToDiskPartition
```

Les classes **ManagementObjet** et **ManagementClass** proposent des surcharges des méthodes *GetRelated*, *GetRelatedClass*, *GetRelationShips* et *GetRelationShipClasses* produisant le même résultat.

```
$C=[wmiclass]"Win32_DiskDrive"
$c.PSBase.GetRelated()
```

```
#ras
$c.PSBase.GetRelationships()
```

```
#ras
$c.PSBase.GetRelatedClasses()
```

```
Win32_DiskPartition                Win32_PhysicalMedia
$c.PSBase.GetRelationshipClasses()
```

```
Win32_DiskDriveToDiskPartition     Win32_DiskDrivePhysicalMedia
```

9.3 Provider de vues

Il existe d'autres formes d'association, plutôt une relation *mère-filles*, par exemple entre un process et un thread :

```
$ps = Gwmi Win32_Process | ?{$_.Name -eq 'powershell.exe'}
$id = $ps.ProcessId
Gwmi Win32_Thread | ?{$_.ProcessHandle -eq $id}
```

Mais ici c'est à vous de retrouver les instances filles dépendantes, à l'aide d'une propriété commune, on construit donc une jointure.

Wmi propose un provider spécialisé dans la construction de ce type de vue, il se nomme ***MS_VIEW_INSTANCE_PROVIDER***.

9.3.1 Création d'une classe de jointure

Pour nos tests créeons un espace de nom dédié, nommé *TestVues* :

```
$ClassNS = [wmiclass]"root:__Namespace"
$NS = $ClassNS.CreateInstance()
 $NS.Name = "TestVues"
$NS.put()
```

Important : pensez à contrôler la sécurité sur l'espace de nom créé, voir le chapitre *Modifier la sécurité sur le référentiel WMI* .

Le provider que nous allons utiliser n'est pas enregistré dans notre espace de nom, enregistrons-le en créant les instances suivantes :

```
$ClassDataProv= [wmiclass]"root\TestVues:__Win32Provider"
$DataProv= $ClassDataProv.CreateInstance()
 $DataProv.Name = "MS_VIEW_INSTANCE_PROVIDER";
 $DataProv.ClsId = "{AA70DDF4-E11C-11D1-ABB0-00C04FD9159E}"
 $DataProv.ImpersonationLevel = 1
 $DataProv.PerUserInitialization = $True
 $DataProv.HostingModel = "NetworkServiceHost"
$Result=$DataProv.put()
```

Sous PowerShell V1 l'accès aux propriétés systèmes de la nouvelle instance WMI nécessite de forcer l'adaptation de l'objet WMI encapsulé dans un objet dotnet, ceci est corrigé avec la version de PowerShell v2 livrée avec la RC1 de Windows Seven :

```
$DataProv|SELECT __* > $NULL
```

Sans cette instruction, sous PS v1 et la v2 CTP 3, l'accès en lecture de la propriété *__path* renvoie une valeur nulle, ici on utilisera la valeur de retour de l'appel à ***$DataProv.put()***

```
$ClassProvReg= [wmiclass]"root\TestVues:__InstanceProviderRegistration"
$ProvReg= $ClassProvReg.CreateInstance()
  #C'est une référence, on pointe sur le chemin de l'instance WMI
  # et pas sur la variable PowerShell
 $ProvReg.Provider =$Result.Path
```

```
 $ProvReg.SupportsPut = $True
 $ProvReg.SupportsGet = $True
 $ProvReg.SupportsDelete = $True
 $ProvReg.SupportsEnumeration =$True
 $ProvReg.QuerySupportLevels =@("WQL:UnarySelect")
$ProvReg.put()
$ClassMethodProvReg=[wmiclass]"root\TestVues:__MethodProviderRegistration"
$MethodProvReg = $ClassMethodProvReg.CreateInstance()
$MethodProvReg.Provider =$Result.Path
$MethodProvReg.put()
```

La déclaration d'une définition de classe WMI utilise la syntaxe MOF. Le fichier texte créé doit ensuite être compilé à l'aide du programme *MofComp.exe* :

```
Type JoinedPartitionDisk.mof
#pragma namespace("\\\\.\\root\\TestVues")

[
 JoinOn("Win32_DiskPartition.DiskIndex = Win32_DiskDrive.Index"),
 ViewSources{
          "SELECT DiskIndex, Index, DeviceID FROM Win32_DiskPartition" ,
          "SELECT Index, Caption, Model FROM Win32_DiskDrive"},
 ViewSpaces{
          "\\\\.\\root\\cimv2",
          "\\\\.\\root\\cimv2"},
 dynamic: ToInstance,
 provider("MS_VIEW_INSTANCE_PROVIDER")
]

class JoinedPartitionDisk
{
[PropertySources{"DiskIndex", "Index"}] UInt32 ID;
[PropertySources{"", "Caption"}] String Caption;
[PropertySources{"DeviceID", ""},Key] String DeviceId;
[PropertySources{"", "Model"}] String Model;
};
MofComp JoinedPartitionDisk.Mof
 #Affichage des instances de la nouvelle classe
gwmi -namespace "root\TestVues" -class "JoinedPartitionDisk"
```

Nous créons donc ici une classe de jointure entre un disque dur et ses partitions.

Note :

- ✓ Les propriétés utilisées dans le qualifier *JoinOn* doivent être de même type,
- ✓ chaque entrée du qualifier *ViewSpaces* référence l'espace de nom où est déclaré la classe,
- ✓ au moins une propriété doit être une clé : *[PropertySources{"DeviceID", ""},Key]*.

En fin de tests vous pourrez supprimer cet espace de nom par l'appel à la méthode *Delete()*.

Il existe deux autres types de vue, la vue d'association qui permet la création de classe à partir de classes se trouvant dans des espaces de nom différent, et la vue d'union qui permet d'unir une ou plusieurs instances de différentes classes au sein d'une classe.

Voir aussi :

Linking Classes Together :

http://msdn.microsoft.com/en-us/library/aa392265(VS.85).aspx

Creating an Association View Class

http://nukz.net/reference/wmi/hh/wmisdk/viewprov_6ab7.htm

Liste des qualificateurs spécifiques au provider de vues :

http://msdn.microsoft.com/en-us/library/aa392897(VS.85).aspx

10 La gestion d'événement

Un autre des intérêts de WMI est d'offrir une gestion d'événements systèmes sans avoir à produire du code de bas niveau, le langage de requête WQL couplé à des classes dotnet suffiront.

Reportez-vous au chapitre 3.8 de l'introduction WMI, pour des explications détaillées sur le fonctionnement de ces événements.

Les événements intrinsèques dérivent de la classe **__InstanceOperationEvent**, ceux de type extrinsèque de la classe **__ExtrinsicEvent**, toutes les classes d'événement dérivent de la classe **_Event.**

Il est possible de surveiller la création de fichier, la modification d'une clé de registre, etc.

10.1 Gestion d'événement intrinsèque synchrone, client d'événement provisoire

On utilise à cette fin la classe **ManagementEventWatcher** qui d'après la documentation :

« Permet de s'abonner à des notifications d'événement temporaires basées sur une requête d'événement spécifiée. »

La requête WQL d'événement se construit de la manière suivante :

- ✓ Les événements ciblés sont les instances de la classe '***__InstanceCreationEvent***',
- ✓ on recherche, toutes les secondes, seulement ceux émis par la classe **Win32_Process,**
- ✓ la propriété *TargetInstance* contient l'objet créé,

```
 #On est informé lors de la création d'un processus
$Query = "SELECT * FROM __InstanceCreationEvent`
        WITHIN 1 `
        WHERE Targetinstance ISA 'Win32_Process'"
```

Sous dotnet on peut utiliser la classe WqlEventQuery :

```
$query =New-object System.Management.WqlEventQuery(
```

```
            "__InstanceCreationEvent",
            [TimeSpan]"0:0:1",
           'TargetInstance isa "Win32_Process"')
```

L'abonnement à l'événement se fait ainsi :

```
$watcher =new-object System.Management.ManagementEventWatcher
$watcher.Query = $query
```

Reste ensuite à récupérer les notifications d'événement synchrones par l'appel à la méthode *WaitForNextEvent()* :

```
$evenement = $watcher.WaitForNextEvent()
Write-Warning ("Le processus a été créé {0}. Son path est : {1}" -F
$evenement.TargetInstance.Name, $evenement.TargetInstance.ExecutablePath)
  #Annule l'abonnement
$Watcher.Stop()
 #Libére les ressources
$Watcher.Dispose()
```

Cet appel, synchrone, bloque l'exécution du script tant qu'un événement, du type recherché, n'est pas détecté par WMI. On ne pourra donc exécuter qu'une seule recherche d'événement à la fois.

La classe **Win32_ProcessStartTrace** peut également être utilisée, la différence réside dans la construction de la requête et les informations proposées par cette classe :

```
$query =New-object System.Management.WqlEventQuery(
         "Win32_ProcessStartTrace",
         [TimeSpan]"0:0:1",
         $null)
```

Enfin, sachez que les instances de ce type d'événement ne sont pas créées dans le référentiel WMI mais en mémoire.

```
(Gwmi __InstanceCreationEvent) -eq $null
#True
(Gwmi Win32_ProcessStartTrace) -eq $null
#True
```

Pour éviter de bloquer l'interface il reste possible d'effectuer du semi-synchrone au sein d'une boucle comme le montre cet exemple :

http://blogs.msdn.com/powershell/archive/2007/01/28/working-with-wmi-events.aspx

10.1.1 À propos de classe de l'événement reçu

La méthode *WaitForNextEvent()* renvoi un objet WMI aux possibilités restreintes, car il est de la classe **ManagementBaseObject**, tous comme les possibles champs référençant des objets.

Par exemple la classe d'événement WMI **__InstanceOperationEvent** possède un champ nommé *TargetInstance*. Comme ce type d'événement s'applique à de nombreuses classes, ce champ doit pouvoir héberger n'importe quel type d'objet, de ce fait son type wmi est *Object*.

Ceci implique que sous dotnet/PowerShell on récupère ce champ en tant qu'objet de la classe **ManagementBaseObject** et pas de la classe **ManagementObject**.

Si on souhaite manipuler ce champ comme un objet de la classe **ManagementObject** on doit le relire dans le référentiel :

```
$Instance=[wmi]"$( $evenement.TargetInstance.__relpath)"
```

Dans tous les cas l'instance WMI est identique.

De plus les méthodes de conversion de date ne sont pas ajoutées à l'objet provenant de l'eventwatcher car sa classe est **ManagementBaseObject**, le fichier d'adaptation de type *C:\WINDOWS\system32\windowspowershell\v1.0\types.ps1xml* ne concerne que les objets de la classe **ManagementObject**.

Le fichier *System.Management.ManagementBaseObject.types.ps1xml* fournit dans les sources de ce tutoriel propose de combler cette lacune.

Ajoutez la ligne suivante dans votre profile PowerShell :

```
 #Placez-vous auparavant dans le répertoire contenant le fichier
 # $PSProfile=$Home+"\Mes documents\WindowsPowerShell"
 # $PSScripts=$PSProfile+"\Scripts"
 # cd $PSScripts
Update-FormatData -prepend `
System.Management.ManagementBaseObject.types.ps1xml
```

Enfin sachez également qu'un événement est délivré à tous les clients abonnés. C'est-à-dire que si vous exécutez le même script de gestion d'événement WMI dans 2 sessions PowerShell, les deux sessions recevront l'événement ciblé. La requête WQL peut être différente, mais porter au moins sur la même classe d'événement ou sur une de ces classes dérivées.

10.1.2 Surveiller un ensemble d'événements

On peut également surveiller les événements de type **__InstanceOperationEvent**, toutes les opérations de création, de modification et de suppression d'instance particulière seront prises en comptes. Le nom de la classe d'événement permettra de les différencier :

```
$query =New-object System.Management.WqlEventQuery(
           "__InstanceOperationEvent",
           [TimeSpan]"0:0:1",
           'TargetInstance isa "Win32_Process"')
$watcher =new-object System.Management.ManagementEventWatcher
$watcher.Query = $query
$e = $watcher.WaitForNextEvent()
$e.__Class
$e.TargetInstance.__Class
```

10.1.3 Surveiller un sous-ensemble d'événements

On peut effectuer une recherche avancée pour ne récupérer que les créations et les suppressions :

```
$query =New-object System.Management.WqlEventQuery(
           "__InstanceOperationEvent",
        [TimeSpan]"0:0:1",
       '(__CLASS="__InstanceCreationEvent" or
         __CLASS="__InstanceDeletionEvent") and
        TargetInstance isa "Win32_Process"')
```

Le reste du code est identique, seules la requête et l'analyse du contenu de l'événement reçu diffèrent.

10.1.4 Surveiller un sous-ensemble d'événements sur plusieurs classes

Pour ce faire, on ajoute un filtre supplémentaire dans la clause WHERE :

```
$query =New-object System.Management.WqlEventQuery(
           "__InstanceOperationEvent",
        [TimeSpan]"0:0:1",
       '(__CLASS="__InstanceCreationEvent"
         or
         __CLASS="__InstanceDeletionEvent"
         ) and
        (TargetInstance isa "Win32_Process"
        or
        TargetInstance isa "Win32_NTLogEvent"
        )
       ')
```

Pour cet exemple l'appel à *WaitForNextEvent()* générera une erreur :

```
Exception lors de l'appel de « WaitForNextEvent » avec « 0 » argument(s) : « Accès refusé »
```

La consultation de la documentation de la classe **Win32_NTLogEvent** nous indique que l'application doit avoir le privilège ***SeSecurityPrivilege*** d'activé.

Ajoutons le code adéquat :

```
 #On réutilise la variable $Query
$watcher =new-object System.Management.ManagementEventWatcher
$watcher.Query = $query
 $Path =New-object System.Management.ManagementPath("\\.\root\cimv2")
 $watcher.Scope = new-object  System.Management.ManagementScope($Path)
  #Nécessaire
 $watcher.PSbase.Scope.Options.EnablePrivileges = $true

$e = $watcher.WaitForNextEvent()
```

```
$e.__Class
$e.TargetInstance.__Class
```

Pour créer une entrée dans l'eventlog nommé *Application*, on doit ajouter au préalable une nouvelle source de la manière suivante :

```
[System.Diagnostics.EventLog]::CreateEventSource("TestEvent",'Application')
```

Ensuite exécutez le code suivant dans une autre session PowerShell :

```
$Event=new-object System.Diagnostics.EventLog("Application")
$Event.Source="TestEvent"
$Event.WriteEntry("Test WMI Event",
                  [System.Diagnostics.EventLogEntryType]::Information)
```

Ce type de requête n'est pas habituel, mais WQL peut nous aider à repousser un peu les limites des événements synchrones.

NOTES

L'usage des mots clés AND et OR est limité dans une requête WQL. Une requête trop complexe provoquera l'erreur WBEM_E_QUOTA_VIOLATION.

WMI vérifie à intervalle régulier (clause WQL *WITHIN*) si un événement donné c'est produit, si c'est le cas il est mis en cache avant d'être délivré aux clients. Ce mécanisme appelé pooling peut entrainer une surcharge CPU sur le poste concerné. De plus si le type d'événement se produit fréquemment le client peut être inondé d'événements, dans ce cas, et si possible, l'usage d'une clause WQL *WHERE* réduira le nombre d'événements reçu.

Pour d'autres type d'événement l'usage de la clause WQL *GROUP* limitera l'envoi d'événement vers le client. L'exemple suivant :

```
SELECT * FROM __InstanceCreationEvent
  WHERE TargetInstance ISA "Win32_NTLogEvent"
  GROUP WITHIN 600 BY TargetInstance.SourceName
  HAVING NumberOfEvents > 25
```

regroupe tous les événements *Win32_NTLogEvent*, provenant d'une même source, reçus pendant 10 minutes (600 secondes), groupe n'excédant pas 25 occurrences

Ici WMI reçoit et traite 25 événements, le client un seul.

Détails des clauses WQL Having et Group :

http://msdn.microsoft.com/en-us/library/aa394606(VS.85).aspx

10.2 Gestion d'événement extrinsèque, client d'événement permanent

Un client d'événement permanent persiste dans le référentiel WMI, même en cas de reboot, et ne nécessite plus, une fois la requête créée, la présence d'un client puisque c'est un provider WMI qui joue le rôle du client.

On laisse le référentiel WMI s'occuper de tout, plus besoin de lancer un script PowerShell pour scruter un type d'événement. Ainsi, certaines alertes peuvent être automatisées.

Il en existe quelques un de préinstallés, vous trouverez la liste ici :

http://msdn.microsoft.com/en-us/library/aa392395(VS.85).aspx

Si on souhaite exécuter un script en réponse à un événement, il y a deux provider qui peuvent répondre à ce besoin, **ActiveScriptEventConsumer** et **CommandLineEventConsumer.**

Le premier exécute directement du code VBScript ou JScript, mais point de PowerShell, le second exécute un programme externe, par exemple PowerShell.exe.

Voyons comment procéder à partir des informations fournies sur ces pages du SDK WMI,

Running a Program from the Command Line Based on an Event :

http://msdn.microsoft.com/en-us/library/aa393249(VS.85).aspx

Implementing Cross-Namespace Permanent Event Subscriptions

http://msdn.microsoft.com/en-us/library/aa390873(VS.85).aspx

10.2.1 Création d'un client basé sur le provider CommandLineEventConsumer

Ce chapitre reprend et complète le blog suivant (adaptant un exemple du SDK) :

http://www.streamline-it-solutions.co.uk/blog/post/Launching-PowerShell-Scripts-in-Response-to-WMI-Events.aspx

Nous devons créer plusieurs instances, la première de la classe du provider **CommandLineEventConsumer** afin de déclarer le programme à exécuter lors de la réception d'un événement :

```
$ScriptFile="C:\Temp\TestWMICmdEvt.ps1"
$Cmdln="powershell.exe -command $ScriptFile %TargetInstance.Handle%"
$Path="C:\WINDOWS\system32\windowspowershell\v1.0\powershell.exe"
$ConsumerClass = [wmiclass]"\root\subscription:CommandLineEventConsumer"
$EventConsumer = $ConsumerClass.CreateInstance()
$EventConsumer.Name = "PowerShell"
$EventConsumer.CommandLineTemplate =$Cmdln
$EventConsumer.ExecutablePath=$path
$EventConsumer.ForceOffFeedback=$False
$EventConsumer.ForceOnFeedback=$True
```

```
$EventConsumer.RunInteractively=$True
$EventConsumer.ShowWindowCommand=1
$EventConsumer.UseDefaultErrorMode=$True
```

Vous noterez que la propriété *CommandLineTemplate* référence le contenu d'une propriété d'une instance WMI, ce formalisme est similaire à celui des variables d'environnement. Seules les propriétés possédant le qualificateur *template* peuvent les utiliser :

```
$ConsumerClass.psbase.Properties|
 Where {$_.qualifiers|Where {$_.name -eq "Template"} }
```

Le détail des templates : http://msdn.microsoft.com/en-us/library/aa393954(VS.85).aspx

Le script TestWMICmdEvt.ps1 peut contenir le traitement que vous voulez, il reçoit un seul paramètre de type integer.

La seconde instance est de la classe **__EventFilter** afin de définir la requête de l'événement à scruter :

```
$FiltreClass = [wmiclass]"\root\subscription:__EventFilter"
$Filtre = $FiltreClass.CreateInstance()
$Filtre.Name = "CreationProcess"
$Filtre.Querylanguage = "WQL"
$Filtre.Query = "select * from __InstanceCreationEvent within 5 where
targetinstance isa 'Win32_Process' and targetinstance.name=Notepad.exe"
$Filtre.EventNamespace = "root\cimv2"
$ResultFiltre=$Filtre.Put()
```

La troisième, et dernière instance, de la classe **__FilterToConsumerBinding** qui est une classe d'association référençant les deux précédentes instances :

```
$AssociationClass=[wmiclass]"\root\subscription:__FilterToConsumerBinding"
$Filtre_EventConsumer = $AssociationClass.createinstance()
$Filtre_EventConsumer.Consumer = $ResultEventConsumer.Path
$Filtre_EventConsumer.Filter = $ResultFiltre.Path
$Filtre_EventConsumer.put()
```

Avec cette approche quelques problèmes se posent. Le premier est que WMI est exécuté avec le compte *AUTORITE NT\SYSTEM* ce compte ne possède pas de profile PowerShell, on doit donc en copier un, si besoin est, dans le répertoire suivant (sous Windows XP) :

C:\WINDOWS\system32\config\systemprofile

De plus, on doit préciser pour des questions de sécurité qui peut envoyer des événements et qui peut y accéder. http://msdn.microsoft.com/en-us/library/aa393016(VS.85).aspx

Les événements sont déclenchés même si l'utilisateur ne dispose pas des droits, dans ce cas WMI rejette la demande est émet une trace dans son fichier de logs.

Ensuite la requête WMI doit remonter le moins d'événements possible. Par exemple, comme le client est persistant, si vous précisez un nom de classe uniquement :

```
where targetinstance isa 'Win32_Process'
```

le client déclenchera automatiquement, lors de la phase de boot, un événement pour chaque nouveau process créé par le compte *AUTORITE NT\SYSTEM.*

Soyez également vigilant aux droits d'accès placés sur le fichier du programme exécuté au sein de ce provider.

Pour retrouver la liste des souscriptions déclarées sur ce provider :

```
gwmi "NTEventLogEventConsumer" -namespace "root\subscription"
```

10.3 Un trojan avec WMI !

Vous trouverez dans le document *The_Moth_Trojan.pdf* les détails sur ce type de virus basé sur le détournement de l'event consumer WMI. Cette présentation a été réalisée par la société *www.security-assessment.com*.

http://djtechnocrat.blogspot.com/2008/10/moth-trojan-uses-wmi-event-consumers.html

10.4 Exécuter une requête d'événement WQL asynchrone

Sous PowerShell V1 cela n'est pas possible à moins d'utiliser les cmdlets du projet PSEventing :

http://pseventing.codeplex.com

10.4.1 Comment surveiller simultanément plusieurs classes d'événements WMI ?

Puisque les appels asynchrones ne sont pas possibles et que les appels synchrones sont limités, on peut modifier le script suivant déjà cité :

http://blogs.msdn.com/powershell/archive/2007/01/28/working-with-wmi-events.aspx

Un objet de la classe **ManagementEventWatcher** ne surveille qu'un type d'événement, pour en surveiller plusieurs on doit associer un objet à chaque type d'événement.

Ce script paramètre un timeout d'écoute sur l'eventwatcher afin qu'il rende la main même si aucun événement attendu ne survient pendant le laps de temps défini.

Scripts concernés : TestWMI_Event-Surveillance.ps1, TestWMI_Event-CréationEvent.ps1

Comme WMI place dans une mémoire tampon les événements reçus on est assuré de ne pas perdre d'événements, sous réserve que WMI puisse traiter tous ces événements (cf. pooling).

Ce n'est pas l'appel à la méthode Start() de l'eventwatcher qui démarre la mise en tampon mais le premier appel à *WaitForNextEvent()*.

Le code suivant est donc possible :

```
...
  while ($true) {
    trap [System.Management.ManagementException] {continue}
     #Chaque ManagementEventWatcher attend pendant 2 secondes
    $e=$watchProcess.WaitForNextEvent()
```

```
    $e2=$watchTempDir.WaitForNextEvent()
    $e3=$watchEventLog.WaitForNextEvent()
...
```

Ainsi une fois abonné on reçoit bien tous les événements même si l'eventwatcher n'est pas en écoute au moment où un événement attendu survient.

Le principe étant qu'on attend, pour chaque eventwatcher, un événement 2 secondes avant que WaitForNextEvent() ne se termine, si un événement survient pendant ce laps de temps on le traite, ensuite on passe à l'eventwatcher suivant et ainsi de suite.

Dans notre exemple on surveille trois événements WMI :

- la création de process,
- la création d'une entrée dans un eventlog,
- la création de fichier dans le répertoire c:\temp

Une chose importante, l'arrêt et la finalisation des eventwatchers :

```
$watchProcess.Stop()
$watchProcess.Dispose()
```

On informe WMI qu'on se désabonne puis on libère les ressources utilisées (objet COM). Sans cela on peut provoquer l'erreur WBEM_E_QUOTA_VIOLATION lors de l'appel à Start().

Parfois ce sera l'erreur Acces denied ou Serveur RPC inaccessible. L'arrêt du service WMI, suivi de son redémarrage réglera le plus souvent le problème.

Voir aussi :

http://blogs.technet.com/askperf/archive/2008/09/16/memory-and-handle-quotas-in-the-wmi-provider-service.aspx

Dans le script de surveillance la fonction New-EventWatcher évite de dupliquer le code de création des eventwatchers.

Une fois ce script exécuté dans une console PowerShell, on doit générer des événements dans une seconde console PowerShell, la première étant dans une boucle d'attente.

On crée 10 process, 10 fichiers texte et 10 entrées dans l'eventlog :

```
1..10|% {Start-Process Notepad.exe;"test" > "c:\temp\PSTest$_.txt";New-EventLogEntryType}
```

On peut ainsi voir comment réagit le script de surveillance fasse à une «rafale» d'événement.

Une solution telle que PSEventing est certes plus élégante et d'une conception plus appropriée à la surveillance, mais cette solution est native et démontre certaines limites de PowerShell.

Note :

Je n'ai pas testé l'exécution de ce script sur plusieurs jours ce qui permettrait de vérifier sur le long terme s'il provoque des erreurs de quotas WMI.

10.5 Forwarding et corrélation d'événements

Le forwarding peut être traduit par réexpédition ce qui permet de réémettre un événement vers une autre machine. Un provider dédié est disponible uniquement sous Windows XP. Il est remplacé par Winrm sous Windows 2008.

Forwarding d'événement sous Winrm :

http://blogs.technet.com/otto/archive/2008/07/08/quick-and-dirty-enterprise-eventing-for-windows.aspx

Sous PS v2 ctp3, la gestion d'événement permet l'event forwarding, c'est-à-dire d'informer un client qu'un événement a eu lieu sur une machine distante.

http://blogs.msdn.com/powershell/archive/2008/06/11/powershell-eventing-quickstart.aspx

La corrélation d'événement permet les requêtes de type :

Le client X est intéressé par des événements de type A, mais seulement si un événement de type B s'est produit dans les 5 dernières minutes.

Ce provider n'étant disponible que sous Windows XP, consultez la documentation sur MSDN.

11 Créer ses propres événements WMI à l'aide de .NET

Le framework dotnet 2.0 facilite la création d'événements WMI pouvant être déclenchés dans une application, cela permet d'intégrer des événements de gestion destinés par exemple aux administrateurs.

Un exemple de classe d'événement (voir les sources dans le répertoire WMIEvent) :

```
public class PoshOperationEvent :
        System.Management.Instrumentation.BaseEvent
    {   // Version simplifiée
        public PoshTransmitter Actor;
        public Int32 ProcessID ;
        public Byte[] RunSpaceInstanceId ;
        public string EventName ;

        public PoshOperationEvent(String EventName, Int32 ProcessHandle, Guid InstanceId)
        {
            if (Process.GetProcessById(ProcessHandle) == null)
            {     // Le process peut ne pas ou ne plus exister.
                 throw new PoshException("Le handle de process n'existe pas.");
            }

            if (InstanceId != null)
            { this.RunSpaceInstanceId = InstanceId.ToByteArray(); }

            this.EventName = EventName;
            this.ProcessID = ProcessHandle;
            this.Actor = PoshTransmitter.Unknown;
        }
       //…
    }
```

On peut utiliser cette possibilité pour émettre et recevoir des messages entre différents process ou runspace PowerShell en s'appuyant sur un mécanisme WMI.

Pour utiliser cet assembly signé, copiez-le dans le *GAC* puis installez-le à l'aide du programme ***InstallUtil.exe*** :

```
set-alias InstallUtil `
 $env:windir\Microsoft.NET\Framework\v2.0.50727\installutil
$AssemblyPath=*A MODIFIER*
cd $AssemblyPath
InstallUtil Wmievent.dll
```

La première installation d'un tel événement met en place si besoin le provider de gestion .NET :

```
Installation du schéma WMI : commencée
Inscription de l'assembly : WMIEvent_SN__Version_1.0.0.0
Vérification de l'existence de l'espace de noms : root\MicrosoftWmiNet
Vérification de l'existence de la classe : root\MicrosoftWmiNet:WMINET_Instrumentation
Vérification de l'existence de la classe : CREATING  root\MicrosoftWmiNet:WMINET_Instrumentation
Vérification de l'existence de la classe : root\MicrosoftWmiNet:WMINET_InstrumentedNamespaces
Vérification de l'existence de la classe : CREATING
root\MicrosoftWmiNet:WMINET_InstrumentedNamespaces
Vérification de l'existence de la classe : root\MicrosoftWmiNet:WMINET_Naming
Vérification de l'existence de la classe : CREATING  root\MicrosoftWmiNet:WMINET_Naming
Vérification de l'existence de l'espace de noms : Root\Default
Vérification de l'existence de la classe : Root\Default:WMINET_Instrumentation
Vérification de l'existence de la classe : CREATING  Root\Default:WMINET_Instrumentation
Vérification de l'existence de la classe : Root\Default:WMINET_InstrumentedAssembly
Vérification de l'existence de la classe : CREATING  Root\Default:WMINET_InstrumentedAssembly
Vérification de l'existence de la classe : Root\Default:MSFT_DecoupledProvider
Vérification de l'existence de la classe : CREATING  Root\Default:MSFT_DecoupledProvider
Vérification de l'existence de la classe : Root\Default:WMINET_ManagedAssemblyProvider
Vérification de l'existence de la classe : CREATING  Root\Default:WMINET_ManagedAssemblyProvider
```

Une fois ceci fait la création de l'événement et son déclenchement se font ainsi :

```
[reflection.assembly]::loadwithpartialname("WMIEvent")
 #Nom de l'événement WMI
$EventName="Process"
$RS=([System.Management.Automation.Runspaces.Runspace]::DefaultRunSpace).I
nstanceID
$Actor=[WMIEvent.PoshTransmissionActor]::PowerShell
 #Création de l'évenement WMI
$StopWatching=New-object WMIEvent.PoshStopWatchingEvent($EventName,
                                                        $pid,
                                                        $RS,
                                                        $Actor)
 #Déclenchement de l'événement WMI
$Event.Fire()
```

Notre classe d'événement *PoshStopWatchingEvent*, dérivée de la classe ***PoshOperationEvent***, contient le membre EventName, celui-ci sera utilisé pour différencier plusieurs « types » d'événement.

Précédemment nous utilisions la propriété __CLASS pour différencier le type de la classe d'événement, on utilisera le même principe :

```
switch ($PoshEvent.__Class)
 {
    "PoshStopWatchingEvent" {
    switch -regex ($PoshEvent.Eventname)
     {
       "^Process$"     {"Arrêt de la surveillance des process."}
       "^AllWatching$" {"Arrêt de toutes les surveillances."}
…
```

Chaque abonné écoute l'événement par la requête suivante :

```
Select * from PoshOperationEvent Within 1
```

Le déclenchement de l'événement se fait dans la session PowerShell, mais sa gestion et sa distribution aux abonnés sont prisent en charge par les mécanismes de WMI.

Puisque dans les données de l'événement WMI on connaît le processID de l'émetteur, on peut le manipuler dans une autre session PowerShell :

```
$query =New-object System.Management.WqlEventQuery(
        "PoshOperationEvent",
        [TimeSpan]"0:0:1")

 # Notre événement WMI est créé dans l'espace de nom 'root\Default'
 # pas dans celui par défaut 'root\CIMV2'
$Scope =new-object System.Management.ManagementScope("root\Default")

$watcher =new-object System.Management.ManagementEventWatcher
$watcher.Query = $query
$watcher.Scope=$Scope
$e = $watcher.WaitForNextEvent()
(get-process -id $e.ProcessID).CloseMainWindow()
```

Il est possible de terminer le process PowerShell.exe déclenchant l'événement par l'instruction *exit*, mais en réalité l'objectif ici n'est pas de terminer le process ayant émis l'événement, mais d'être averti d'une fin de traitement. C'est-à-dire savoir qu'il s'est passé quelque chose.

11.1 Envoi de messages entre deux instances de PowerShell

L'envoi de message de ce type n'a pas de contrainte, on le crée puis on l'envoie, en revanche le client abonné à cet événement doit nécessairement avoir une gestion de message à l'aide d'un eventwatcher.

Scripts concernés :

TestWMIdotnetEvent-Surveillance.ps1, TestWMIdotnetEvent-CréationEvent.ps1

On peut reprendre la mécanique utilisée pour surveiller simultanément plusieurs classes d'événements WMI. On ajoute un eventwatcher pour gérer nos événements WMI dotnet :

```
 $Query=New-object System.Management.WqlEventQuery("PoshOperationEvent",`
  [TimeSpan]"0:0:01")
 $watchPoshEvent=New-EventWatcher $Query "WatchPoshEvent" "Root\Default"`
 -Start
```

Ensuite on souhaite signaler au script de surveillance soit :

- l'arrêt de la fin de la surveillance des process,
- le redémarrage de la surveillance des process,
- ou l'arrêt complet du script de surveillance.

Il nous faut modifier la gestion des eventwatcher afin de les contrôler plus finement :

```
 $PoshEvent=$watchPoshEvent.WaitForNextEvent()
 if ($watchProcess -ne $null)
  {$e=$watchProcess.WaitForNextEvent()}
 if ($watchTempDir -ne $null)
 …
```

L'arrêt d'un eventwatcher :

```
 if ($watchProcess -ne $null)
  {
    $watchProcess.Stop()
    $watchProcess.Dispose()
    $watchProcess=$null
    Write-Warning "Arrêt de la surveillance du process Notepad."
   …
```

Son redémarrage :

```
if ($watchProcess -ne $null)
 { Write-Warning "La surveillance du process Notepad est toujours active."}
else
 {
  $Query=New-object System.Management.WqlEventQuery(`
    "__InstanceCreationEvent",
    [TimeSpan]"0:0:1",
    'TargetInstance isa "Win32_Process" and TargetInstance.Name = "notepad.exe"')
  $watchProcess=New-EventWatcher $Query "WatchProcess" -Start
  Write-Warning "Redémarrage de la surveillance du process Notepad."
 }
 …
```

Une fois notre événement géré on doit **impérativement** prendre en charge sa suppression :

```
$PoshEvent=$null
```

Sinon il reste disponible dans le cache WMI du client et lui sera de nouveau présenté, indépendamment des autres clients abonnés.

Ainsi on peut émettre des messages d'une console PowerShell **A** vers une console PowerShell **B**.

Dans ces exemples de gestion d'événements WMI dotnet on ne tient pas compte des informations de process ou de runspace, on peut donc exécuter plusieurs instances de la surveillance, chacune recevra les mêmes messages.

Pour vos tests vous pouvez aussi utiliser le script Test-Winform-WMIDotnet-CréationEvent.ps1.

11.2 Runspace

La gestion d'événement asynchrone n'étant pas possible, la surveillance de plusieurs événements WMI dans une même session de PowerShell V1 peut également se faire au sein de runspace.

On pourrait également coupler l'usage de runspace avec les traitements de masse, par exemple envisager une requête WMI par runspace, dans ce cas le problème de timeout serait moins pénalisant. Cet aspect nécessiterait d'y consacrer un tutoriel.

Pour ce qui est de la construction des runspaces vous pouvez consulter ces liens :

http://jtruher.spaces.live.com/blog/cns!7143DA6E51A2628D!130.entry

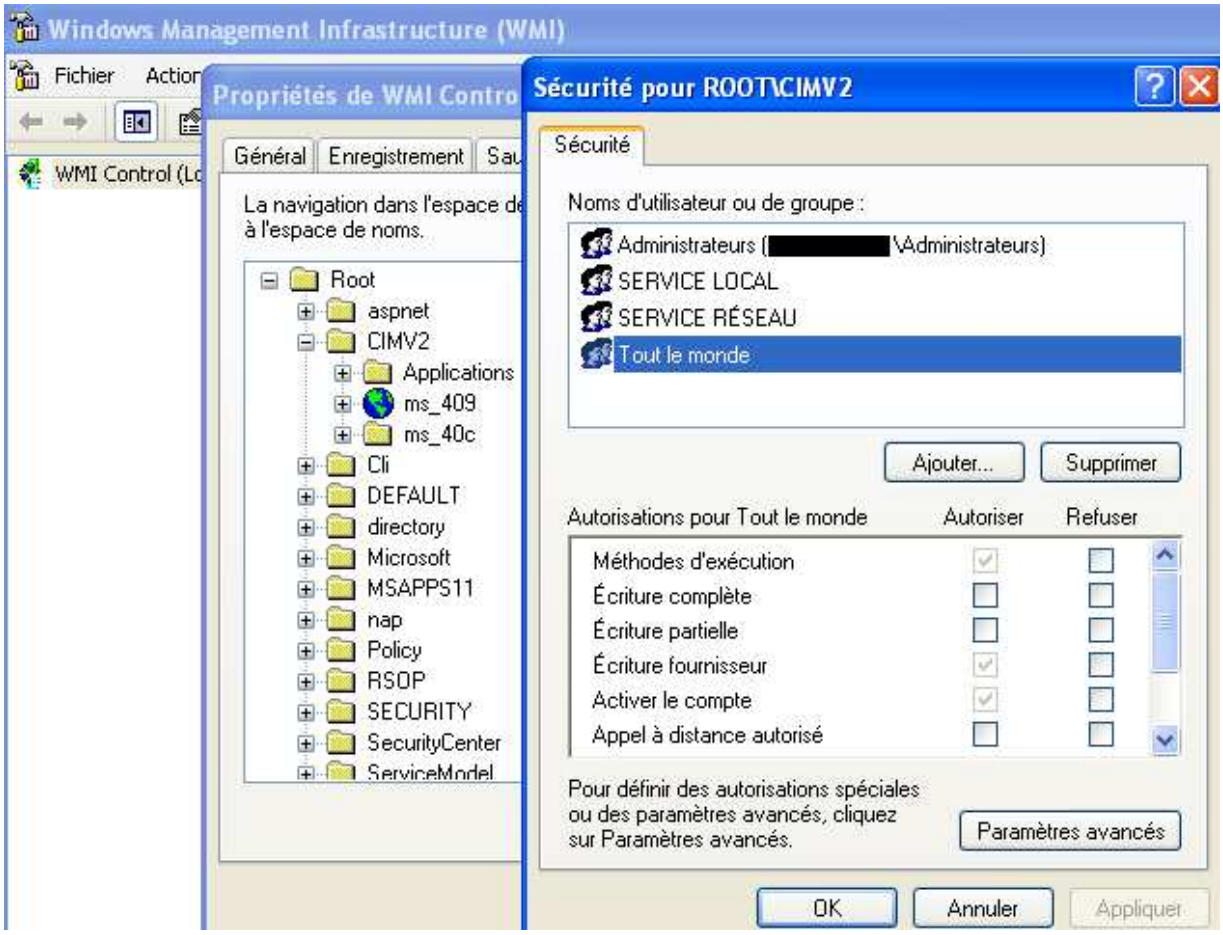
http://thepowershellguy.com/blogs/gaurhoth/archive/2007/10/08/an-example-of-how-to-use-new-taskpool.aspx

http://www.jansveld.net/powershell/2008/12/split-job-093/

http://www.jansveld.net/powershell/2008/06/split-job-version-0-9/

12 Modifier la sécurité sur le référentiel WMI

Utilisez *Wmimgmt.msc*. Sélectionnez le menu *Action* puis *Propriétés*.



Voir aussi la définition des permissions :

http://technet.microsoft.com/en-us/library/cc787533(WS.10).aspx

13 Outils

The WMI Diagnosis Utility-- Version 2.0

http://www.microsoft.com/downloads/details.aspx?FamilyID=d7ba3cd6-18d1-4d05-b11e-4c64192ae97d&displaylang=en

Voir aussi sur le sujet :

http://blogs.msdn.com/wmi/archive/2006/05/12/596266.aspx

WMI Code Creator v1.0

http://www.microsoft.com/downloads/details.aspx?FamilyID=2cc30a64-ea15-4661-8da4-55bbc145c30e&DisplayLang=en

WMI Diagnosis Utility

http://www.microsoft.com/technet/scriptcenter/topics/help/wmidiag.mspx

Il existe de nombreuses archives de traitements WMI réalisés en VBscript qui sont sources d'informations importantes et sont classés par thèmes ou par type d'opérations.

WMI Scripting Examples

http://www.microsoft.com/downloads/details.aspx?FamilyID=B4CB2678-DAFB-4E30-B2DA-B8814FE2DA5A&displaylang=en

14 Les évolutions de PowerShell V2 concernant WMI

Windows PowerShell 2.0 propose 4 nouveaux cmdlets dédiés à WMI :

- Invoke-WMIMethod (http://technet.microsoft.com/en-us/library/dd315300.aspx)
- Remove-WMIObject (http://technet.microsoft.com/en-us/library/dd347605.aspx)
- Set-WMIInstance (http://technet.microsoft.com/en-us/library/dd315391.aspx)
- Register-WmiEvent (http://technet.microsoft.com/en-us/library/dd315242.aspx)

Le SDK de Powershell propose désormais une classe dérivée de PSCmdlet afin de spécialiser des cmdlets, la classe **WMIBaseCmdlet** :

« *Serves as a base class for cmdlet that use WMI connection options.* »

D'autres cmdlets utilisent en interne des classes WMI :

Stop-Process, Debug-Process (attache un debugger à un process), Get-HotFix, Test-Connection, Stop-Computer, Start-Computer et Restore-Computer.

Cet article (http://www.microsoft.com/technet/scriptcenter/topics/winpsh/wmiin2.mspx) passe en revue ces nouveaux cmdlets propres à WMI, certains ne sont que des facilités d'écriture. Leur usage permet de réduire le nombre d'instructions nécessaire à la manipulation d'instance WMI.

Par exemple la suite de commande suivante :

```
$VarEnvironment= ([wmiclass]"Win32_Environment").CreateInstance()
$VarEnvironment.UserName = [System.Environment]::UserName
$VarEnvironment.Name = "VariableWMI"
$VarEnvironment.VariableValue = "Variable d' environnement crée avecWMI "
$VarEnvironment.Put()
```

Devient :

```
 #On crée une variable d'environnement si elle n'existe pas
$VarEnvironment=Set-WmiInstance -class win32_environment `
 -argument @{
     Name="TestVarWmi";
     VariableValue="testvalue";
   UserName="$env:ComputerName\\$Env:UserName"
 }
```

```
gwmi win32_environment
 #Modifie l'instance existante
Set-WmiInstance -InputObject $VarEnvironment `
 -argument @{VariableValue="Modifietestvalue"} `
 -PutType UpdateOnly
gwmi win32_environment
 #Supprime l'instance
Remove-WMIObject -InputObject $VarEnvironment
```

Le cmdlet **Get-WMIObject** propose de nouveaux paramètres afin de faciliter la gestion de l'authentification sur des machines distantes. Ces nouveaux paramètres sont :

-Impersonation, -Authentication, -Locale, -EnableAllPrivileges, -Amended, -DirectRead, -Authority.

Le cmdlet Get-WmiObject de la CTP 3 (http://technet.microsoft.com/en-us/library/dd315295.aspx) offre de nouvelles fonctionnalités telles que la recherche récursive dans un espace de nom ou l'exécution de l'opération en tâche de fond, voir le paramètre *–AsJob.*

Il autorise également les caractères spéciaux de remplacement tels que *, ? ou [] :

```
Get-WMIObject -List -namespace "root" -class "*disk*" –recurse
```

Cette commande renvoie :

```
   NameSpace: ROOT\CIMV2

Name                                Methods              Properties
----                                -------              ----------
CIM_LogicalDisk                     {SetPowerState, R...  {Access, Availability, BlockSize, Caption...}
...
Win32_PerfRawData_PerfDisk_ ...     {}                   {AvgDiskBytesPerRead,...


   NameSpace: ROOT\WMI

Name                                Methods              Properties
----                                -------              ----------
MSDiskDriver                        {}                   {}
...
SystemConfig_V2_PhyDisk             {}                   {BootDriveLetter, BytesPerSector, Cylinders,...}
```

Enfin, l'accès aux membres de l'objet encapsulé ne nécessite plus l'usage de *.psbase*.

Get-Member lui nécessitera toujours son usage.

De plus, WinRM peut être combiné avec WMI ce qui règle les problèmes de connexion au travers d'un firewall :

http://blogs.msdn.com/wmi/archive/2009/01/17/accessing-wmi-data-via-winrm.aspx

Voir aussi *What's new in WMI for Windows 7*

http://blogs.msdn.com/wmi/archive/2009/08/08/what-s-new-in-wmi-for-windows-7.aspx

14.1 Gestion des événements asynchrone

```
$query =New-object System.Management.WqlEventQuery `
  "__InstanceCreationEvent",([TimeSpan]"0:0:1"),"TargetInstance isa
'Win32_Process'"

$watcher =new-object System.Management.ManagementEventWatcher
$watcher.Query = $query

 #Abonnement à l'événement EventArrived proposé par
 # la classe ManagementEventWatcher
 #Il est du type EventArrivedEventHandler
Register-ObjectEvent $watcher `
                     -EventName "EventArrived" `
                     -SourceIdentifier "WMI.ProcessCreated" `
                     -Action {
     # Sender = $args[0]
     # EventArguments =: $args[1]
  Write-Warning ("Le process à été créé {0}. Son path est : {1}" -F `
                 $args[1].NewEvent.TargetInstance.Name,`
                 $args[1].NewEvent.TargetInstance.ExecutablePath)
  } # -Action
```

Chaque création de process affichera le détail de l'objet *$args[1].NewEvent.TargetInstance.*

L'équipe de développement de PowerShell a pensé à simplifier cette tâche en proposant le cmdlet **Register-WmiEvent**. Le code suivant ne nécessite plus l'usage de l'objet **ManagementEventWatcher** :

```
Register-WMIEvent -query "Select * From __InstanceCreationEvent within 3
Where TargetInstance ISA 'Win32_Process'" `
 -sourceIdentifier "NewProcess" `
 -action {Write-Host "A new process has started."}
```

Vous pouvez à son sujet consulter le tutoriel suivant :

http://www.microsoft.com/technet/scriptcenter/topics/winpsh/events.mspx

15 Conclusion

À la différence du VBScript, l'usage de WMI sous PowerShell est simplifié pour les tâches les plus courantes. En revanche les tâches nécessitant un paramétrage plus fin devront être effectuées à l'aide des classes .NET. Les possibilités offertes par PowerShell v1 limitent quelque peu les traitements WMI, alors que le VBScript «colle» au plus prés des API de scripting WMI.

En «bricolant » on peut en dépasser quelques une, heureusement les nouvelles fonctionnalités de la version 2 amélioreront l'usage de WMI sous PowerShell. Les plus importantes, comme les

événements aperçus dans le chapitre 12.1, nécessiteront d'être approfondi dans des tutoriels dédiés.

16 Liens

WMI Reference

http://msdn.microsoft.com/en-us/library/aa394554.aspx

En savoir plus sur WMI .NET

http://msdn.microsoft.com/fr-fr/library/ms257341(VS.80).aspx

Cette rubrique contient des liens vers des rubriques qui expliquent plus en détail les différentes zones de WMI .NET, indiquent comment utiliser WMI .NET et décrivent les différents aspects de l'utilisation de WMI dans le Framework dotNET.

Vue d'ensemble de WMI .NET

http://msdn.microsoft.com/fr-fr/library/ms257340(VS.80).aspx

Tracing WMI Activity

http://msdn.microsoft.com/en-us/library/aa826686(VS.85).aspx

Seul semble concerné les postes Windows Vista et supérieur ainsi que Windows Server 2008.

WMIFAQ

http://www.microsoft.com/technet/scriptcenter/resources/wmifaq.mspx

Un ouvrage de référence sur WMI (vbscript)

Understanding WMI Scripting, Alain Lissoir, éditeur Digital Press. ISBN : 1-55558-266-4

Leveraging WMI Scripting - Operating Systems, éditeur Digital Press.ISBN : 1-55558-299-0

Doc WMI en ligne

http://nukz.net/reference/wmi/index.html

Sur Technet vous pouvez également consulter les articles de Don Jones sur WMI :

http://technet.microsoft.com/fr-fr/magazine/2009.02.windowspowershell.aspx

Le blog de l'équipe WMI

http://blogs.msdn.com/wmi/

Quelques articles autour des performances du 'moteur' WMI

http://blogs.technet.com/askperf/archive/tags/WMI/default.aspx

Par exemple : WMI Debug Logging, Memory and Handle Quotas in the WMI Provider Service, Splitting up WMI Providers for Troubleshooting,…

Windows Remote Management and WMI

http://msdn.microsoft.com/en-us/library/aa384463(VS.85).aspx