Les modules PowerShell.

Par Laurent Dardenne, le 17 septembre 2012.

Ce tutoriel aborde l'usage, l'organisation et la conception de code Powershell hébergé dans un module.

Spécial dédicace à Anthony :-)

Les fichiers sources

Version 1.0

Chapitres

1 Prérequis

Pour ce tutoriel j'ai indiqué tous niveaux étant donné que l'on peut se contenter d'utiliser des modules, le développement de modules nécessitant comme prérequis des connaissances sur la conception et l'écriture de script, la compréhension de la notion de portée, voir de quelques mécanisme .NET.

2 La notion de module

Au travers des modules Powershell (version 2 et supérieure), les concepteurs du langage offre la possibilité d'assembler des données et du code, facilitant le découpage et l'organisation de vos traitements. Un module peut être utilisée autant par un administrateur automatisant une tâche que par un développeur de cmdlet, pour ce dernier un module lui permet de faciliter le déploiement et l'usage de son code par le premier.

A la différence de Powershell version 1, un module évite l'usage du programme *Installutil.exe* pour enregistrer un snapin, c'est-à-dire une dll hébergeant des cmdlets, et ne requiert plus de droit administrateur pour installer de nouveaux cmdlet. Un déploiement *xCopy* suffit.

A partir du moment où on peut regrouper des traitements en fonction de leur utilisation on permet la réutilisation de code tout en le spécialisant. Il existe par exemple un module pour administrer Office 365, un autre pour gérer BitLocker, etc.

Un des autres avantages d'un module, et à la différence d'un assembly dotnet, est que l'on peut le supprimer de la mémoire une fois le traitement associé terminé. Là où un assembly dotnet, une fois chargé dans son domaine d'application (Appdomain), ne peut plus être déchargé.

Un module binaire peut également charger automatiquement un provider Powershell.

Note : la prochaine version 3 de Powershell n'utilisera plus de snapins pour charger ses cmdlets de base, mais des modules.

2.1 Le module, un script comme un autre ?

Le plus souvent un module est un fichier texte, similaire à un script, mais dont l'extension est *.psm1*, en revanche Powershell l'utilise de la manière suivante :

- un module ne peut être chargé en dotsource, on doit utiliser le cmdlet **Import-Module**,
- une fois chargé, il est persistant comme une fonction peut l'être via le provider *Function*, bien qu'il n'existe pas de provider *Module*,
- les éléments (alias, fonction, variable,…) définis dans le module peuvent être déclarés accessibles ou pas dans la session Powershell important le module,
- un module peut importer ou imbriquer d'autres modules (nested module),
- il peut être, ainsi que ses dépendances, supprimé de la mémoire. Pour décharger un module, on doit utiliser le cmdlet **Remove-Module**,
- on peut également, à partir d'un scriptblock, créer un module dynamique,

- un module peut être un module binaire liè à une dll, dans ce cas on utilisera un manifeste de module, un fichier texte dont le contenu respecte une syntaxe particulière.

3 Usages et création d'un module

Utilisons le script .\New-FileModule.ps1 présent dans répertoire des scripts de démonstration.

Celui-ci crée, dans le répertoire C:\Temp, un fichier de script dont l'extension est ***.psm1*** :

```
# cd répertoire d'installation des scripts du tutoriel

.\New-FileModule.ps1
Dir C:\temp\my*.psm1
Type C:\temp\mymodule.psm1
```

Le code du script contient une déclaration d'une fonction et d'un d'alias :

```
Function Get-Files{
 Dir C:\Windows
}

New-Alias glf Get-Files
```

3.1 Lister les modules chargés en mémoire

Pour obtenir la liste des modules publics chargés en mémoire, on utilisera le cmdlet **Get-Module** *:*

```
Get-Module
```

On peut constater que la création d'un fichier de module n'est pas liée à son chargement en mémoire, le module *mymodule.psm1* n'est pas listé par l'appel du cmdlet **Get-Module**.

3.2 Lister les modules disponibles

Powershell recherche les modules dans les répertoires contenus dans la variable d'environnement *$Env:PSModulePath :*

```
$Env:PSModulePath -Split ';'
```

Par défaut la variable d'environnement PSModulePath référence les répertoires *$PsHome\Modules* et *$Env:UserProfile\WindowsPowerShell\Modules*

```
C:\Documents and Settings\UserName\Mes documents\WindowsPowerShell\Modules
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\
```

Le répertoire du profile utilisateur est ajouté par le runtime.

L'installation de Powershell ne crée pas de profile utilisateur, c'est à vous de le créer et d'y ajouter le répertoire nommé *Modules*.

Pour obtenir la liste des modules installés dans les répertoires contenus dans la variable d'environnement *PSModulePath*, on utilisera le paramètre *ListAvailaible* du cmdlet **Get-Module** :

```
Get-Module –ListAvailable
```

Notre cmdlet étant dans un répertoire inconnu du chemin de recherche de Powershell, il n'est pas listé.

3.3 Importer un module

Dés qu'un module est dans un des répertoires de recherche un simple appel à **Import-Module** suffit :

```
import-module BitsTransfer
Get-Module
```

L'affichage sera au minimum :

```
ModuleType  Name           ExportedCommands
----------  ----           ----------------
Manifest    BitsTransfer   {Start-BitsTransfer, Remove-BitsTransfer, Resume-BitsTransfer,...
```

Par défaut Import-Module n'emet aucune donnée, si vous souhaitez récupérer des informations du module importé, utilisez le paramètre **-Passthru**.

Essayons de charger notre module en cours de développement :

```
Import-Module MyModule.psm1
```

```
Import-Module : Le module « mymodule.psm1 » spécifié n'a pas été chargé, car aucun fichier de module valide n'a été trouvé dans un répertoire de module.
```

Pour charger notre module, en cours de développement, il nous faut spécifier son chemin d'accès :

```
Import-Module C:\temp\mymodule.psm1
#ou
Cd c:\temp
Import-Module "$Pwd\mymodule.psm1"
#ou Import-Module ".\mymodule.psm1"
```

Le chargement est effectif sans que powershell nous informe de la réussite de l'opération :

```
Get-Module
```

```
ModuleType  Name           ExportedCommands
----------  ----           ----------------
Manifest    BitsTransfer   {Start-BitsTransfer, Remove-BitsTransfer, Resume-BitsTransfer,...
Script      mymodule       Get-Files
```

Pour être informé de l'opération de chargement on peut utiliser le paramètre *Verbose* :

```
Import-Module "$Pwd\mymodule.psm1" -Verbose
```

```
COMMENTAIRES : Chargement du module à partir du chemin « C:\temp\mymodule.psm1 ».
COMMENTAIRES : Exportation de la fonction « Get-Files ».
COMMENTAIRES : Importation de la fonction « Get-Files ».
```

Les commentaires nous informent que la fonction *Get-Files* est exportée du module et importée dans le provider de fonction :

```
dir Function:get-Files
```

```
CommandType  Name        Definition
-----------  ----        ----------
Function     Get-Files   ...
```

Par défaut les alias ne sont pas exportés :

```
Dir Alias:glm
```

```
Get-ChildItem : Impossible de trouver le chemin d'accès « glm », car il n'existe pas.
```

Nous aborderons par la suite ce comportement.

Note : La version 3 de Powershell proposera un mode de chargement automatique de module lors d'une recherche de commande effectuée à l'aide du cmdlet **Get-Command**.

3.4 Supprimer un module

Pour supprimer un module de la mémoire, on utilisera le cmdlet **Remove-Module** :

```
Remove-Module MyModule
```

Ici nul besoin de préciser le chemin d'accès, seul le nom du module sans extension suffit pour le décharger.

Note :

Si toutefois il existait, avant le chargement d'un module, une ou des fonctions ayant le même nom que celle exportées par le module, elles sont remplacées lors de l'import du module.

L'appel au cmdlet **Remove-Module** supprime ses fonctions exportée, mais ne restaure pas l'état précédent. Il y a donc ici à notre insu une modification de code.

3.5 Forcer le chargement d'un module

Lors du développement d'un module on est amené au cycle d'opérations suivantes :

- chargement du module en mémoire,
- tests du code du module,
- suppression du module de la mémoire,
- corrections du code du module,
- *nouveau cycle.*

Dans ce cas l'appel suivant suffit

```
Import-Module "$Pwd\mymodule.psm1" -Force
```

Le paramètre *–Force* supprime le module puis le réimporte. Lors d'appels successifs à **Import-Module** portant sur un même module, et sans préciser le paramètre *Force*, seul le premier appel est effectif les suivants n'ont aucun effet, le cmdlet **Import-Module** ne fait rien, à part constater l'existence du module en mémoire.

Pour lister les cdmlets dédié aux modules :

```
Get-Command *-*module*
```

3.6 Ajouter un chemin de recherche de module

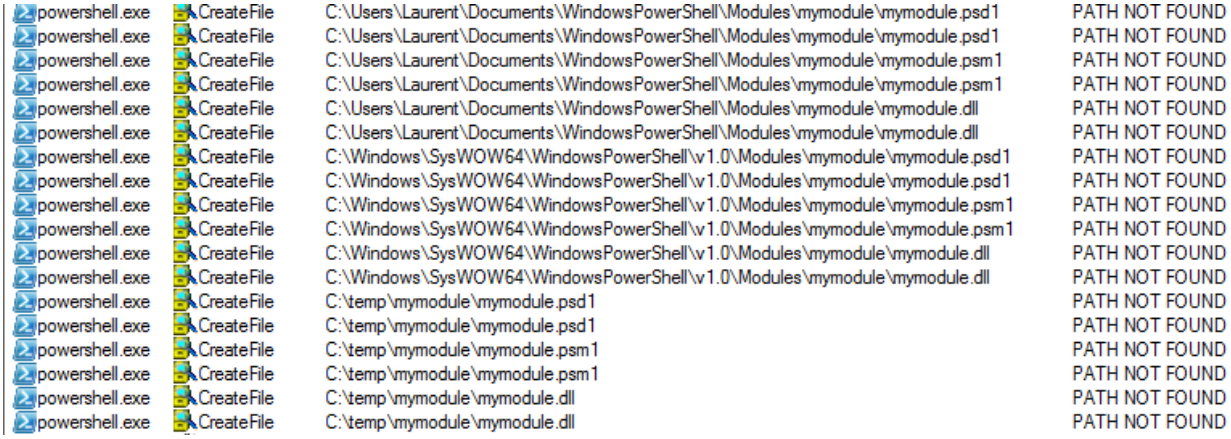
Il est possible de modifier temporairement la variable d'environnement *PSModulePath* :

```
$env:PSModulePath = $env:PSModulePath + ";c:\temp"
Import-Module mymodule
```

```
Import-Module : Le module « mymodule » spécifié n'a pas été chargé, car aucun fichier de module valide n'a été trouvé dans un répertoire de module.
```

Cette déclaration de chemin de recherche ne fonctionne pas, car le nouveau module doit être hébergé dans un sous répertoire de C:\Temp dont le nom est identique au nom du module.

Comme nous le montre l'utilitaire Process Monitor :



powershell.exe	CreateFile	C:\Users\Laurent\Documents\WindowsPowerShell\Modules\mymodule\mymodule.psd1	PATH NOT FOUND
powershell.exe	CreateFile	C:\Users\Laurent\Documents\WindowsPowerShell\Modules\mymodule\mymodule.psd1	PATH NOT FOUND
powershell.exe	CreateFile	C:\Users\Laurent\Documents\WindowsPowerShell\Modules\mymodule\mymodule.psm1	PATH NOT FOUND
powershell.exe	CreateFile	C:\Users\Laurent\Documents\WindowsPowerShell\Modules\mymodule\mymodule.psm1	PATH NOT FOUND
powershell.exe	CreateFile	C:\Users\Laurent\Documents\WindowsPowerShell\Modules\mymodule\mymodule.dll	PATH NOT FOUND
powershell.exe	CreateFile	C:\Users\Laurent\Documents\WindowsPowerShell\Modules\mymodule\mymodule.dll	PATH NOT FOUND
powershell.exe	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0\Modules\mymodule\mymodule.psd1	PATH NOT FOUND
powershell.exe	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0\Modules\mymodule\mymodule.psd1	PATH NOT FOUND
powershell.exe	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0\Modules\mymodule\mymodule.psm1	PATH NOT FOUND
powershell.exe	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0\Modules\mymodule\mymodule.psm1	PATH NOT FOUND
powershell.exe	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0\Modules\mymodule\mymodule.dll	PATH NOT FOUND
powershell.exe	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0\Modules\mymodule\mymodule.dll	PATH NOT FOUND
powershell.exe	CreateFile	C:\temp\mymodule\mymodule.psd1	PATH NOT FOUND
powershell.exe	CreateFile	C:\temp\mymodule\mymodule.psd1	PATH NOT FOUND
powershell.exe	CreateFile	C:\temp\mymodule\mymodule.psm1	PATH NOT FOUND
powershell.exe	CreateFile	C:\temp\mymodule\mymodule.psm1	PATH NOT FOUND
powershell.exe	CreateFile	C:\temp\mymodule\mymodule.dll	PATH NOT FOUND
powershell.exe	CreateFile	C:\temp\mymodule\mymodule.dll	PATH NOT FOUND

Powershell parcourt chaque nom de chemin inséré dans la variable $Env:PSModulePath en lui ajoutant le nom de répertoire du module à importer.

On doit donc respecter la structure suivante :

- ✓ Nom du répertoire parent : C:\Temp
- ✓ Nom du répertoire du module : C:\Temp\MyModule

Ce dernier est identique au nom du fichier du module, que celui-ci soit un fichier .psm1 ou un fichier manifeste .psd1.

Le chemin complet est donc : C:\Temp\MyModule\MyModule.psm1

```
Remove-Module myModule
$env:PSModulePath = $env:PSModulePath + ";C:\Temp"
```

Placez-vous dans le répertoire des scripts de démos :

```
# cd répertoire d'installation des scripts du tutoriel
.\New-FileModuleV2.ps1
Dir C:\temp\MyModule\my*.psm1
Type C:\temp\MyModule\mymodule.psm1

Import-Module mymodule
```

De cette manière la présence du nom de répertoire n'est plus nécessaire, on délégue au mécanisme de recherche le soin de retrouver notre module.

Nous verrons plus avant comment, à partir du code du module, charger des scripts externes en utilisant des chemins relatifs.

Si aucune extension n'est précisé **Import-Module** recherche un fichier *ModuleName.psd1*, puis *ModuleName.psm1* et enfin *ModuleName.dll*.

Voir ce script d'installation de module, *Install-Module* : http://poshcode.org/?show=1875

Consulter également ce post sur l'import de module à partir d'un partage :

http://huddledmasses.org/how-to-import-binary-modules-from-network-shares/

Ainsi que les recommendations suivantes concernant l'installation de modules :

http://msdn.microsoft.com/en-us/library/windows/desktop/dd878350(v=vs.85).aspx

3.6.1 Configurer la variable PSModulePath

Pour modifier la variable système PSModulePath de façon durable, vous pouvez utiliser les instructions suivantes :

```
$InstallModulePath="$env:ProgramFiles\Societe\Modules"
$CurrentValue=[Environment]::GetEnvironmentVariable("PSModulePath", "Machine")
if (($CurrentValue -Split ';') -notcontains  $InstallModulePath)
{
   Write-Host "Modification de la variable d'environnement PsModulePath" -fore Green
   [Environment]::SetEnvironmentVariable("PSModulePath", $CurrentValue + ";$InstallModulePath", "Machine")
  }
else
 {Write-Warning "La variable d'environnement PsModulePath est déjà renseignée pour MonModule." }
```

Notez que cette variable ne modifie pas l'environnement des process courants, pas même le sien.

Si vous souhaitez paramètrer le process courant, c'est-à-dire la session powershell, vous devrez combiner les deux approches.

3.7 Exporter des membres d'un module

Par défaut toutes les fonctions et tous les filtres sont exportés du module, c'est-à-dire visible dans la portée de l'appelant. Par défaut les alias, les variables et les cmdlets ne sont pas exportées.

La déclaration de module du fichier *New-FileModuleV2.ps1* démontre ce comportement :

```
Dir function:Get-Files #fonction publique
Dir alias:glf #alias privé
```

```
Get-ChildItem : Impossible de trouver le chemin d'accès « glf », car il n'existe pas.
```

Pour préciser les membres à exporter on utilisera le cmdlet **Export-ModuleMember**, celui-ci ne peut être appelé qu'au sein du code d'un module, son usage dans un autre contexte provoquera une erreur.

A partir du moment où on utilise ce cmdlet dans le code du module, le comportement par défaut ne s'applique plus, c'est à dire que l'on devra préciser tous les membres que l'on souhaite exporter.

La déclaration de module du fichier *New-FileModuleV3.ps1* exporte seulement les alias :

```
Remove-Module myModule
```

Placez-vous dans le répertoire des scripts de démos :

```
.\New-FileModuleV3.ps1
Import-Module mymodule
Dir function:Get-Files|ft Module* #fonction privéee
Get-ChildItem : Impossible de trouver le chemin d'accès « get-files », car il n'existe pas.
Dir alias:glf #alias public
```

La déclaration de module du fichier *New-FileModuleV4.ps1* exporte les alias et les fonctions :

```
Export-ModuleMember -function Get-Files -alias glf
```

Placez-vous dans le répertoire des scripts de démos :

```
# cd répertoire d'installation des scripts du tutoriel
Remove-Module myModule
.\New-FileModuleV4.ps1
Import-Module mymodule
Dir alias:glf #alias public
```

Pour ne pas exporter un membre, il suffit de ne pas le préciser dans la liste d'export.

Les métadonnées des membres exportés contiennent une référence au module auquel elles appartiennent :

```
Dir function:Get-Files|ft Module* #fonction publique
```

3.7.1 Portée d'une variable

Si dans un module vous implémentez par exemple un compteur, la variable portant sa valeur doit utiliser la portée ***Script:*** sinon sa portée sera celle de la fonction courante :

```
$Compteur=-7
$Script:Compteur=-7
function CompteurOK{
    Write-Host "AVANT valeur du compteur OK : $Compteur $Script:Compteur"
    $script:Compteur++
    Write-Host "APRES valeur du compteur OK : $Compteur $Script:Compteur"
 }
 function CompteurNOK{
    Write-Host "AVANT valeur du compteur NOK : $Compteur $Script:Compteur"
```

```
    $Compteur++
    Write-Host "APRES valeur du compteur NOK : $Compteur $Script:Compteur"
 }

Export-ModuleMember CompteurNOK,CompteurOK
```

Placez vous dans le répertoire des exemples et charger le module Scope.psm1 :

```
Import-Module "$pwd\Scope.psm1"
```

Le résultat de différents appels croisés :

```
CompteurnOK #Modification locale, aucune modification globale
            #dans la portée du module
            #le contenu de $Script:Compteur n'a pas changé
```

```
AVANT valeur du compteur NOK : -7 -7
APRES valeur du compteur NOK : -6 -7
```

```
CompteurOK  #Modification globale dans la portée du module
            #le contenu de $Script:Compteur a changé
```

```
AVANT valeur du compteur OK : -7 -7
APRES valeur du compteur OK : -6 -6
```

```
CompteurOK #Modification globale dans la portée du module
           #le contenu de $Script:Compteur a changé
```

```
AVANT valeur du compteur OK : -6 -6
APRES valeur du compteur OK : -5 -5
```

```
CompteurnOK #Affiche la valeur courante de la variable $Compteur,
            #par défaut celle contenue dans $Script:Compteur
            #Aucune modification globale dans la portée du module,
            #le code modifie une variable locale.
            #Le contenu de $Script:Compteur n'a pas changé
```

```
AVANT valeur du compteur NOK : -5 -5
APRES valeur du compteur NOK : -4 -5
```

```
CompteurnOK #Idem
```

```
AVANT valeur du compteur NOK : -5 -5
APRES valeur du compteur NOK : -4 -5
```

```
CompteurOK #Modification globale dans la portée du module,
           #le contenu de $Script:Compteur a changé
```

```
AVANT valeur du compteur OK : -5 -5
APRES valeur du compteur OK : -4 -4
...
```

L'export de la variable ayant la portée ***Script:*** vous permettra d'accéder au compteur dans la session courante :

```
Export-ModuleMember CompteurNOK,CompteurOK -variable Compteur
```

Celle-ci sera supprimée lors de l'appel Remove-Module.

3.7.2 Exporter des membres d'un module

Si deux modules exportent des membres portant des noms identiques, que ce soit des variables ou des fonctions, et que l'import du module se fasse dans la même portée alors la dernière déclaration primera. L'usage de préfixe dans le nommage des membres évitera le plus souvent ce problème.

3.7.2.1 Préfixer automatiquement les commandes exportées

On peut dés la conception se prémunir des collisions de nom, soit utiliser le mécanisme dédié proposé par le cmdlet **Import-Module**. Le paramètre *–Prefix* ajoute un préfixe à la partie nom des cdmlets ou fonctions exportées :

```
Import-Module mymodule -Prefix My
Get-Module MyModule
```

```
ModuleType  Name        ExportedCommands
----------  ----        ----------------
Script      MyModule    Get-Files
```

```
Get-Command  Get-My*
```

```
CommandType     Name            Definition
-----------     ----            ----------
Function        Get-MyFiles     ...
```

Notez que dans les données renvoyées par le cmdlet **Get-Module** (cf. ExportedCommands), le nom de la fonction exportée n'est pas préfixé, mais elle est bien accessible par le nom *Get-MyFiles*, la fonction *Get-Files* n'étant pas déclarée dans la session Powershell.

4 Développement d'un module de script

Nous avons vu comment construire le chemin d'accès d'un module, voyons comment le récupérer dans le code du module

4.1 L'objet Module

Il est possible d'obtenir des métadonnées d'un module lors de son import :

```
$myModule=Import-Module MyModule -Passthru
```

Ou lors de sa recherche :

```
$myModule=Get-Module MyModule
```

Affichons son type:

```
$myModule.GetType().fullname
```

```
System.Management.Automation.PSModuleInfo
```

Et ses propriétés :

```
$MyModule|Select-Object *
```

```
ExportedCommands    : {Get-Files}
Name                : mymodule
Path                : C:\temp\mymodule\mymodule.psm1
Description         :
Guid                : 00000000-0000-0000-0000-000000000000
ModuleBase          : C:\temp\mymodule
PrivateData         :
Version             : 0.0
ModuleType          : Script
AccessMode          : ReadWrite
ExportedFunctions   : {[Get-Files, Get-Files]}
ExportedCmdlets     : {}
NestedModules       : {}
RequiredModules     : {}
ExportedVariables   : {}
ExportedAliases     : {[glf, glf]}
SessionState        : System.Management.Automation.SessionState
OnRemove            :
ExportedFormatFiles : {}
ExportedTypeFiles   : {}
```

Pour ce module certaines propriétés ne sont pas renseignées, car elles ne sont pas utilisées, par exemple *NestedModules*, ou bien contiennent des valeurs par défaut, par exemple V*ersion.*

La plupart des propriétés correspondent à des champs pouvant être déclarés dans un fichier dit manifeste de module, que nous verrons plus avant dans ce tutoriel.

Un autre intérêt de récupèrer un objet de type *System.Management.Automation.PSModuleInfo* est d'accéder au contexte du module, afin d'exécuter du code dans la portée du module :

```
&$myModule Get-Variable Name
```

Cette appel exécute le cmdlet dans la portée du module est accède à la variable privée nommée *Name*. On peut également exécuter une fonction privée :

```
& $m.ExportedFunctions.Private
```

Bien que ces appels soient possibles, il est préférable de les éviter dans votre code, à moins d'avoir une très bonne raison !

Voir aussi : http://blogs.msdn.com/b/powershell/archive/2009/06/03/peering-into-script-modules.aspx

4.2 Retrouver le chemin du module

Lors du chargement d'un module, son code peut référencer la variable automatique *$PSScriptRoot*. Celle-ci contient le nom du chemin du module, il est donc possible de charger des scripts externes ou des fichiers de ressources (ps1xml, psd1, etc), voir de sauvegarder des données dans un répertoire dédié à chaque module :

```
$Name=$MyInvocation.MyCommand.ScriptBlock.Module.Name
Write-Host "Chargement du module [$Name] à partir du répertoire : $PSScriptRoot"
Remove-Module myModule
```

Placez-vous dans le répertoire des scripts de démos :

```
.\New-FileModuleV5.ps1
Import-Module mymodule
```

```
Chargement du module [mymodule)] à partir du répertoire : C:\temp\mymodule
```

4.3 Code d'initialisation

Il n'existe pas de section de code dédié à l'initialisation, vous pouvez débuter l'initialisation de votre module dés la première ligne de code : création de variable, de raccourcis, d'alias de chargement de dll ou de script externe…

Nous verrons que certaines de ces opérations pourront être déplacées dans un fichier manifeste.

4.4 Code de finalisation

Il n'existe pas de section de code dédié à la finalisation, par contre l'objet module propose une propriété nommée *OnRemove* de type scriptblock :

```
Get-Module mymodule|gm -Name on*
```

```
   TypeName: System.Management.Automation.PSModuleInfo

Name        MemberType   Definition
----        ----------   ----------
OnRemove    Property     System.Management.Automation.ScriptBlock OnRemove {get;set;}
```

Le code contenu est exécuté avant la libération mémoire. Vous pouvez y placer du code de finalisation (libération de ressources, nettoyage de fichier temporaire, etc.)

Voici comment procéder :

```
#Le code de la propriété 'OnRemove' est appelé lors de
#la suppression du module.
$MyInvocation.MyCommand.ScriptBlock.Module.OnRemove = { OnRemoveMyModule }
```

Par convention vous pouvez préfixer le nom d'une fonction par *OnRemove* suivi du nom de module *MyModule,* **celle-ci doit rester dans la portée privé du module** :

```
function OnRemoveMyModule {
  Write-Host "Supprime le module $Name" –fore Green
}
```

```
Remove-Module myModule
```

Placez-vous dans le répertoire des scripts de démos :

```
.\New-FileModuleV6.ps1
Import-Module mymodule
```

```
Chargement du module [mymodule)] à partir du répertoire : C:\temp\mymodule
Remove-Module myModule
```

```
Finalise le module
```

Visualisons le comportement du paramètre *–Force* :

```
Import-Module mymodule -Force
```

```
Chargement du module [mymodule)] à partir du répertoire : C:\temp\mymodule
Import-Module mymodule -force
Finalise le module
Chargement du module [mymodule)] à partir du répertoire : C:\temp\mymodule
```

Le code de la propriété *OnRemove* est bien appelé avant le code "d'initialisation" du module et seulement si le module est présent en mémoire.

Voir aussi :

http://connect.microsoft.com/PowerShell/feedback/details/559651/powershell-doesnt-invoke-module-onremove-event-handlers-as-it-shuts-down

http://connect.microsoft.com/PowerShell/feedback/details/498983/v2-modules-cannot-remove-module-that-exposes-a-read-only-variable

4.5 Localisation d'un module

Pour adapter les messages selon la langue d'installation du poste, on utilisera le cmdlet **ConvertFrom-StringData** qui s'appuie sur la convention de construction d'aborescence des fichiers d'aide.

Pour l'aide en Français on crée le répertoire suivant : *C:\Temp\MyModule\fr-FR*

Puis le fichier *MyModuleLocalizedData.psd1* qui contient les lignes de code suivantes :

```
ConvertFrom-StringData @'
 MsgInitialise=Chargement du module [{0}] à partir du répertoire: {1}
 MsgFinalise=Finalise le module {0}
'@
```

Notez que son nom n'est pas normé. Ensuite dans le code d'initialisation du module, on ajoutera l'appel du chargement des données localisées :

```
Import-LocalizedData -BindingVariable MessageTable `
                     -Filename MyModuleLocalizedData.psd1 `
                     -EA Stop
```

Reste à construire le message paramétré :

```
Write-Host ($MessageTable.MsgInitialise -F $Name,$PSScriptRoot)
```

Testons

```
Remove-Module myModule
```

Placez-vous dans le répertoire des scripts de démos :

```
.\New-FileModuleV7.ps1
Import-Module mymodule
```

```
Chargement du module [mymodule)] à partir du répertoire : C:\temp\mymodule
```

On obtient bien le même message, mais cette fois-ci il s'adapte à la langue courante (sous réserve de fournir le fichier adéquat dans le répertoire associé à la langue courante).

Voir aussi :

National Language Support (NLS) API Reference :

http://msdn.microsoft.com/fr-fr/goglobal/bb896001.aspx

Using-Culture, fonction modifiant temporairement la culture :

http://blogs.msdn.com/b/powershell/archive/2006/04/25/583235.aspx

4.6 Convention de nommage de fonction

Ajoutons manuellement une fonction dans notre module :

```
Function Fait-Untruc {
    Write-host "fait qq chose…"
}
```

Cette fonction ne respecte pas la convention de nommage préconisée par Microsoft, à savoir ***Verbe-Nom*** : http://msdn.microsoft.com/en-us/library/ms714428(VS.85).aspx

Celle des paramètres n'est pas vérifiée, mais autant la respecter :

http://msdn.microsoft.com/en-us/library/dd878352(VS.85).aspx

L'import provoquera l'affichage de l'avertissement suivant :

```
Import-Module mymodule -Force
```

```
AVERTISSEMENT : Certains noms de commandes importés contiennent des verbes non approuvés qui les rendent moins détectables. Utilisez le paramètre Verbose pour plus d'informations ou tapez Get-Verb pour obtenir la liste des verbes approuvés.
```

Le détail :

```
Import-Module mymodule -Force -Verbose
```

```
AVERTISSEMENT : Certains noms de commandes importés contiennent des verbes non approuvés qui les rendent moins détectables. Utilisez le paramètre Verbose pour plus d'informations ou tapez Get-Verb pour obtenir la liste des verbes approuvés.
COMMENTAIRES : Le nom de commande « Fait-Untruc » contient un verbe non approuvé qui le rend moins détectable.
…
```

Le paramètre -*Verbose* affiche le nom de la fonction qui utilise un verbe inconnu.

4.7 Module en lecture seule

On peut vouloir protéger le module d'une libération accidentelle, pour cela on configure son mode d'accès en lecture seule :

```
$MyInvocation.MyCommand.ScriptBlock.Module.AccessMode="ReadOnly"
```

Désormais, le module ne pourra être supprimé qu'en précisant le paramétre -*Force* lors de l'appel du cmdlet **Remove-Module**.

4.8 Fichier d'aide de module

Pour réaliser l'aide en ligne d'un module on peut s'appuyer sur les fonctionnalités existantes des fonctions avancées (cf. about_Comment_Based_Help). L'inconvénient de cette approche est

qu'elle ne permet pas de localiser le texte de l'aide. Pour localiser l'aide on doit utiliser un fichier XML externe comme indiqué ici :

http://msdn.microsoft.com/en-us/library/dd878343(v=vs.85).aspx

Vous trouverez quelques explications complémentaires dans ce post :

http://stackoverflow.com/questions/1432717/powershell-v2-external-maml-help

On peut créer un squelette de fichier MAML en utilisant la fonction suivante :

http://poshcode.org/3197

Vous trouverez une version améliorée et un exemple d'utilisation dans le répertoire de démos nommée *Help-MAML*.

Comme il n'existe pas d'éditeur dédié au fichier d'aide MAML destiné à un script ou à un module, un outil tel que XMLpad pourra vous aider.

4.9 Chargement de scripts externes

Vous pouvez construire votre module en utilisant des scripts .ps1 externes et les charger en dot source dans la portée du module. L'exemple suivant utilise un script par fonction à exporter :

```
$ScriptsPath="$PSScriptRoot\Scripts"
$Local:Scripts=Get-Childitem $ScriptsPath|Where {$_.Extension -eq '.ps1'}
if ($Local:Scripts -ne $null)
{
   Write-Verbose "`n`rImport des scripts externes du répertoire :`r`n
$ScriptsPath"
   $Local:FName=""
   $ExportedFunctions=$Local:Scripts|
     Foreach {
        try{
          $Local:FName=split-path $_.Fullname -leaf
          Write-Verbose "$FName "
           # Charge la function dans la portée du module
          . $_.Fullname
           #Fonction à exporter
           Write-warning "$($_.Name)"
           $_.Name -replace '.ps1$',""
        } catch {
          #[System.Management.Automation.PSSecurityException]
         throw "Chargement impossible du fichier : $(split-path $FName -
leaf)`r`n$_ "
        }
    }
}
else {Write-Verbose "`n`rAucun scripts externes."}
```

```
Write-warning "$ExportedFunctions"
Remove-Variable File,FName

Export-ModuleMember -Function $ExportedFunction
```

Ces scripts peuvent exécuter des traitements ou déclarer des fonctions. Libre à vous d'adapter le code précédent en utilisant des warning ou des Write-Error au lieu de déclencher des exceptions.

Voir aussi : http://connect.microsoft.com/PowerShell/feedback/details/746643/myinvocation-mycommand-path-is-null-when-importing-a-script-file-from-a-module-file-psm1

Note : le système d'aide ne semble pas fonctionner lorsque les fonctions des scripts importés référence un fichier d'aide commun au module :

```
Function Test-ProviderConstraint{
# .ExternalHelp ValidationsArgument-Help.xml
…
```

Si la fonction est intégrée au corps du module, le système d'aide fonctionnera.

4.10 Gestion des erreurs

Le module suivant vous permettra de générer des exceptions spécifiques à un module tout en renseignant l'error record associé : http://poshcode.org/1573

5 Module dynamique

Ce type de module permet de s'affranchir de l'usage d'un fichier, on peut donc construire un module à la volée à l'aide du cmdlet **New-Module**. L'exemple suivant renvoi un objet module mais ne le charge pas :

```
New-Module Test {$i=10; Export-ModuleMember -var i}
```

Bien qu'on n'utilise pas de fichier, on doit tout de même explicitement charger un module dynamique :

```
New-Module Test {$i=10; Export-ModuleMember -var i} | Import-Module
```

5.1 Un module comme Objet

Le cmdlet **New-Module** permet également de renvoyer un objet personnalisé (PSObject) :

```
$Object=New-Module Test {
   $Private=-1;
   $Public=10;
  Export-ModuleMember -variable Public
 } -AsCustomObject
```

Notez que le module utilisé en arrière plan pour construire l'objet personnalisé n'est pas accessible en tant que module. Le cmdlet **Get-Module** ne peut y accèder.

Soyez attentif au fait qu'un object ainsi construit nécessite tout de même beaucoup plus de mémoire qu'un objet personnalisé construit avec **New-Object**. Cela reste une possibilité

d'implémenter simplement des membres privés et un 'finaliseur', sans avoir à utiliser du C# ou du VB.NET.

La construction d'un module comme objet peut également se faire sur un appel à **Import-module** :

```
$Object=Import-Module MonModule –AsCustomObject
```

Dans ce cas le nommage Verbe-Nom nécessitera une syntaxe particulière lors de l'appel des méthodes de l'objet (methodes basées sur les fonctions exportées) :

```
$Object."Get-File"()
```

5.2 Ajout de membre personnalisé

Si vous ajoutez des membres personnalisés, seuls les membres exportés par le module seront accessible dans le code associé aux membres personnalisés :

```
Add-Member –Input $Object -membertype ScriptProperty -Name Nombre -value {
     $this.Private}  -secondvalue { $this.Private=Args[0] }
```

L'appel à **Add-Member** crée bien une nouvelle propriété, mais le code n'accèdera pas à la variable interne au module nommée *Private.*

L'inverse fonctionnera :

```
$Object=New-Module Test {
  $Public=10;
  Function Test{
   Write-host "La variable Externe= $($this.Externe)"
  }
  Export-ModuleMember -variable Public –Function *
 } –AsCustomObject
$Object.Test()
$Object=$Object|
    Add-Member -membertype ScriptProperty -Name Externe -value {
     "Membre externe"} –passthru
$Object.Test()
```

On constate que la variable ***$this*** est bien créée dans le PSObject 'basé module'.

6 Manifeste de module

Un manifeste attache des métadonnées à un module, celles-ci décrivent ses fonctionnalités et permettent également de contrôler son versionning, ses prérequis et de spécifier des éléments à charger automatiquement. Un manifeste de module est optionnel, sauf si vous comptez importer un assembly installé dans le cache global des assemblies (GAC).

Le fichier manifeste porte le même nom que le module, mais son extension est .psd1, et doit être dans le même répertoire que le module auquel il se référe. Un manifeste est construit autour d'une hashtable normée, lors de son chargement le runtime exécute du code d'initialisation à partir de ses clés.

6.1 Liste des clés d'un fichier manifeste

Nom de clé	Usage
ModuleToProcess	Indique le fichier primaire (*root*) du module. Peut être le nom de fichier d'un module de script (. PSM1) ou d'un module binaire (.DLL).
ModuleList	Liste d'inventaire des fichiers du module. Note : *Ces fichiers ne sont pas automatiquement chargés.*
ModuleVersion	Numéro de version du module. Seule cette clé est requise dans la hashtable constituant le manifeste.
GUID	Identifiant unique du module.
Author	Noms des auteurs.
CompanyName	Nom de l'entreprise.
Copyright	Information de copyright ou de copyleft.
Description	Description du module.
PowerShellVersion	Indique la version minimum du moteur PowerShell pouvant fonctionner avec ce module. Valeur admise 1.0 ou 2.0. Note : *La version 3 de Powershell autorisera la valeur 3.0.*
PowerShellHostName	Indique le nom du host Windows PowerShell minimum pouvant fonctionner avec ce module (cf. *$Host.Name*).
PowerShellHostVersion	Indique la version minimum du host Windows PowerShell pouvant fonctionner avec ce module.
DotNetFrameworkVersion	Indique la version du Framework .NET requise par le module.
CLRVersion	Indique la version du Common Language Runtime (CLR) du Framework .NET requise par le module.
ProcessorArchitecture	Indique l'architecture processeur requise par le module. Les valeurs valides sont x86, AMD64, IA64, et None (inconnue ou non spécifiée).
RequiredModules	Indique les modules devant être présents dans l'état de session appelante. Si les modules indiqués n'y sont pas, l'import du module échouera. Note : *Les modules cités ne sont pas importés, Powershell vérifie seulement que ces modules sont déjà importés.*

RequiredAssemblies	Indique le ou les fichiers .DLL utilisé par le module. Windows PowerShell charge les assemblies spécifiés avant le chargement des fichiers de types ou de formats, avant l'import de modules imbriqués ou avant l'import du fichier du module spécifié dans la clé *ModuleToProcess*.
ScriptsToProcess	Indique les fichiers de script (. Ps1) exécutés dans l'état de session de l'appelant lorsque le module est importé. Vous pouvez utiliser cette clé pour préparer un environnement. Note : *Pour spécifier des scripts qui s'exécutent dans l'état de session du module, utilisez la clé NestedModules ou charger vos scripts au sein du module.*
TypesToProcess	Liste des fichiers de type à charger. cf. cmdlet ***Update-TypeData***.
FormatsToProcess	Liste des fichiers de formatage à charger. De portée globale. cf. cmdlet ***Update-FormatData***.
NestedModules	Liste ordonnées des modules imbriqués à importer dans l'état de session du module courant.
FunctionsToExport	Liste des fonctions à exporter.
CmdletsToExport	Liste des cmdlets à exporter. Peut contenir des caractères génériques.
VariablesToExport	Liste des variables à exporter.
AliasesToExport	Liste des alias à exporter.
FileList	Liste d'inventaire des fichiers du module. Notez que ces fichiers ne sont pas automatiquement chargés.
PrivateData	Indique les données passées au module lors de son import.

Vous trouverez des détails supplémentaires dans l'aide en ligne du cmdlet **New-ModuleManifest**, qui comme son nom l'indique permet de créer un nouveau fichier manifeste :

http://technet.microsoft.com/en-us/library/dd819477.aspx

Vous trouverez également des détails supplémentaires sur MDSN Library :
http://msdn.microsoft.com/en-us/library/dd878337(VS.85).aspx

6.2 Précisions sur le comportement et l'usage de certaines clés

6.2.1 Export keys

Comme indiqué dans les articles suivants :

http://www.itidea.nl/index.php/powershell-export-modulemember-vs-export-keys-in-manifest/

http://www.itidea.nl/index.php/powershell-export-functions-variables-and-aliases-with-wildcards/

Les clés *ToExport modifient le comportement du cmdlet **Export-ModuleMember**, dans le cas où le manifeste et le module comporte une déclaration d'export, l'instruction du manifeste sera prioritaire.

6.2.2 ModuleVersion

Le cmdlet **Import-Module** permet de charger un module d'après son numéro de version porté par la clé *Moduleversion*. Comme nous l'avons vu précédement lors de la construction de l'installation d'un module, son répertoire ne peut contenir qu'un seul nom de fichier de manifeste ayant le même nom que le module.

Pour gérer ce versioning il nous faut installer le nouveau module dans un autre répertoire puis le référencer dans la variable d'environnement *$Env:PSModulePath*.

Voir aussi :

http://huddledmasses.org/working-with-multiple-versions-of-powershell-modules/

http://msdn.microsoft.com/en-us/library/windows/desktop/dd878350(v=vs.85).aspx

Note : Le code suivant est possible, sous réserve de l'installation des deux dll dans le GAC :

```
# cd répertoire d'installation des scripts du tutoriel
Cd "fichiers\Manifest GAC assemblies\v1"
ipmo .\Log4Poshv1.psd1
cd ..\v2
ipmo .\Log4Poshv2.psd1
```

Le chargement des deux versions est accepté, ainsi que le chargement des deux versions d'un même assembly :

```
True    v2.0.50727     C:\Windows\assembly\GAC_MSIL\log4net\1.2.10.0__1b44e1d426115821\log4net.dll
True    v2.0.50727     C:\Windows\assembly\GAC_MSIL\log4net\1.2.11.0__669e0ddf0bb1aa2a\log4net.dll


[MTA] C:\temp> get-module -name log*

ModuleType Name                     ExportedCommands
---------- ----                     ----------------
Script     Log4Poshv1               {Get-LogRootLogger, Get-LogLoggerRepository, Get-LogLoggers,
Script     Log4Poshv2               {Get-LogRootLogger, Get-LogLoggerRepository, Get-LogLoggers,
```

Microsoft recommande d'**éviter de charger plusieurs versions d'un assembly dans le même contexte** :

http://msdn.microsoft.com/fr-fr/library/dd153782.aspx#avoid_loading_multiple_versions

6.2.3 GUID

Pour générer un GUID, on procédera ainsi :

```
[GUID]::NewGuid()
```

```
Guid
----
80051b71-cb62-4d03-98a0-6cefab9bf2b7
```

6.2.4 PowerShellHostVersion

Le terme américain ‘host’ référence ici une application utilisant en interne le runtime Powershell.

Cette clé ne référence donc pas seulement la console Powershell mais de nombreuses applications. Ceci dit, je laisse le soin à Jeffrey Snover de vous expliquer la subtilité de cette clé :

http://blogs.msdn.com/b/powershell/archive/2010/01/23/powershellhostversion-wtf.aspx

6.2.5 RequiredModules

Cette clé référence un tableau contenant une ou plusieurs hashtable. Chaque hashtable détaille le module requis :

```
RequiredModules=@(
 @{ModuleName="HyperV";GUID="E9223F85-3735-4641-A14F-91522CF6180A";
ModuleVersion="2.1.0.0"}
 @{ModuleName="Orchestrator";GUID="5dff7c60-fdd4-4b63-989d-dffc84a7b631";
ModuleVersion="1.0.0.0"}
)
```

Voir aussi ce post : http://huddledmasses.org/another-module-manifest-gotcha/

Cette clé ne charge pas les modules référencés, mais teste uniquement leur présence.

Pour charger un module requis dans le code de votre module, vous devez procéder ainsi :

```
#By Joel 'Jaykul' Bennett
if (!(Get-Module Dependency)) { ## Or check for the cmdlets you need
    ## Load it nested, and we'll automatically remove it during clean up
    Import-Module Dependency -ErrorAction Stop
}

# if you wanted to check for versions ...
if (!(Get-Module Dependency | Where { $_.Version -ge "2.5" )) {
    ## Load version 2.5 (or newer), or die
    Import-Module Dependency -Version 2.5 -ErrorAction Stop
}
```

6.2.6 RequiredAssemblies

Cette clé charge, dans le runspace courant (une session ou un job), les dlls utilisées par le module. Si aucun chemin n'existe Powershell chargera la dll à partir du cache globale des assemblies (GAC).

Pour charger une dll à partir du répertoire du module utilisez l'instruction suivante :

```
RequiredAssemblies=Join-Path $psScriptRoot
"Microsoft.BackgroundIntelligentTransfer.Management.Interop.dll"
```

Note : On utilise ici un ensemble restreint du langage Powershell documenté dans le fichier suivant :

```
get-help about_data_section
```

Celui-ci est également détaillé dans le chapitre 'Appendice D' du document suivant :

http://www.manning.com/payette2/WPSiA_APPENDIXES.pdf

Chargement à partir du GAC

S'il existe plusieurs versions d'un même assembly, Powershell chargera la dll ayant le numéro de version supérieure :

```
RequiredAssemblies="log4net"
```

Pour charger un assembly par son numéro de version, spécifiez le nom complet de la dll (*FullName*) :

```
RequiredAssemblies="log4net, Version=1.2.10.0, Culture=neutral,
PublicKeyToken=1b44e1d42611582
```

Note : Une dll ne contenant pas de déclaration de cmdlet peut tout à fait être chargée. Bien que l'on se retrouve avec un module binaire dont la dll n'a rien à voir avec la notion de module Powershell.

6.2.7 NestedModules

Les modules précisés dans cette clé sont chargés AVANT que ne soit chargé les modules contenus dans la clé *ModuleToProcess*.

Si une erreur survient lors du chargement d'un des modules imbriqués, le module appelant ne sera pas chargé.

6.2.8 PrivateData

Les données privées ne sont pas accessible dans le code d'initalisation du module, mais par le code des fonctions. Difficile de savoir si c'est un bug ou le comportement attendu.

7 Modules imbriqués

Un module peut imbriquer l'import d'un ou plusieurs modules.

L'instruction suivante charge le module nommé ***Test*** qui importe en interne le module nommé ***Nested***.

Placez-vous dans le répertoire des scripts de démos :

```
# cd répertoire d'installation des scripts du tutoriel
Cd ImbricationDeModules
$M=ipmo "$pwd\Test" –Passthru
```

```
Charge le module imbriqué (Nested)
Charge le module primaire(Test)
```

On peut déjà remarquer que le module imbriqué est chargé en premier. Affichons les informations des modules présents en mémoire :

```
Get-Module
```

```
ModuleType Name         ExportedCommands
---------- ----         ----------------
Script     Pscx         {Invoke-AdoCommand, Set-Writable, New-MSMQueue, Write-GZip,...}
Script     Test         {}
```

Le module imbriqué nommé *Nested* n'est pas listé par le cmdlet **Get-Module,** examinons les données du module portées par la variable $M, particulièrement le membre NestedModules :

```
$M|Select NestedModules
```

```
NestedModules
-------------
{Nested}
```

Ce membre contient la liste des modules imbriqués au sein du module nommée *Test* :

```
$M.NestedModules.GetType().FullName
```

```
System.Collections.ObjectModel.ReadOnlyCollection`1[[System.Management.Automation.PSModuleInfo,
System.Management.Automation, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]]
```

On peut donc interroger les métadonnées de ses modules imbriqués :

```
$M.NestedModules[0].SessionState
```

7.1.1 Ordre de chargement

Rechargeons notre module primaire :

```
$M=ipmo "$pwd\Test" –Force -Passthru
```

```
Finalise le module primaire Test
Finalise le module imbriqué Nested
Charge le module imbriqué (Nested)
Charge le module primaire(Test)
```

On constate que le scriptblock porté par la propriété *OnRemove* est appellé en premier sur le module primaire puis sur les modules imbriqués, dans l'ordre inverse du chargement.

L'ordre de chargement des modules imbriqués respecte l'ordre de déclaration indiqué dans la clé ***NestedModules*** du manifeste associé au module :

```
# ModuleToProcess
NestedModules = @('.\Nested\Nested.psd1','.\Nested2\Nested2.psd1')
```

Vous pouvez tester les manifestes suivants *Test0.psd1, Test1.psd1, Test2.psd1* en les renommant en *Test.psd1*.

Les fonctions exportées par le module imbriqué sont vues comme appartenant au module appelant. Si le module imbriqué n'exporte aucune fonction, le module appelant n'en verra aucune.

8 Développement d'un module binaire

Les modules binaires remplacent les snap-ins utilisés dans la version 1 de Powershell, ils sont associés à un fichier manifeste et constitués d'une ou plusieurs dlls contenant des cmdlets.

Vous trouverez ci-dessous les liens d'un tutoriel en deux parties rédigé par *Adam Driscoll* développeur et leader technique chez Quest Software.

Building Binary PowerShell Modules – Part 1 – Getting Started

http://csharpening.net/?p=738#

Building Binary PowerShell Modules – Part 2 – Design Principles and Other Guidelines :

http://csharpening.net/?p=853#

Une video, 'Write Script and Binary Modules':

http://technet.microsoft.com/en-us/edge/Video/ff711007

Pour télécharger la dll utilisée dans cette vidéo : http://wasp.codeplex.com/

Dans cette vidéo on peut voir Bruce Payette mettre en oeuvre la construction à la volée d'un module binaire :

```
$Source=@"
namespace MyDynamicBinaryModule
{
  using System.Management.Automation;
  [Cmdlet("Invoke","Hello")]
  public class InvokeHelloCommand : PSCmdlet
  {
     protected override void ProcessRecord()
     {
        WriteObject("Hello there, Bruce!");
     }
  }
}
```

```
"@

Add-Type -PassThru -TypeDefinition $Source|
 Foreach-Object {$_.Assembly}|
 Import-Module -Verbose –PassThru
Get-Module ; Invoke-Hello
```

8.1 Masquage de cmdlet (shadowing)

L'usage de **Remove-Module** sur un module binaire masque le(s) cmdlet(s) exporté(s) et ne supprime pas de la mémoire la dll associée. En cas de rechargement du module, la dll étant toujours en mémoire, PowerShell démasquera le(s) cmdlet(s) précédement masqué(s).

8.2 Rappel des types de module

Il existe plusieurs types de module, information que l'on retrouve lors de l'appel à Get-Module

Type de module	Extension	Description
Module de script	.PSM1	Un module dont les membres sont définis dans un fichier module de script (.psm1). Peut être associé ou non à un fichier manifeste. *Note* : Un module créé dynamiquement est similaire à un module de script, bien que celui-ci ne soit pas asssocié à un fichier.
Module binaire	. PSD1 + .DLL .PSD1 +.PSM1 + .DLL	Un module dont les membres sont définis dans un assembly dotnet (. dll). Un module binaire ne nécessite qu'un fichier .psd1 et un fichier .dll contenant du code compilé.
Manifeste de module	.PSD1 .PSD1 +.PSM1	Un module qui est défini par un fichier manifeste de module (. PSD1) dont la clé *ModulesToProces* est vide. Note : On peut charger une dll qui ne contient pas de cmdlets.
Module dynamique	Aucune	Module crée dynamiquement, n'est pas créé à partir d'un fichier.

Note : La version 3 proposera deux types de module supplémentaires nommés CIM et Workflow. Voir l'aide en ligne sur MSDN, *PSModuleInfo.ModuleType* :

http://msdn.microsoft.com/en-us/library/windows/desktop/system.management.automation.moduletype(v=vs.85).aspx

9 État de session Windows PowerShell

La documentation de Microsoft nous indique que l'état de session (session state) se réfère à la configuration actuelle d'une session Windows PowerShell ou d'un module. Une session Windows PowerShell est l'environnement opérationnel qui est utilisé soit par l'utilisateur de manière interactive en ligne de commande soit par programmation au sein d'une application hôte.

Du point de vue d'un développeur, une session Windows PowerShell se réfère à la durée de vie d'un runspace, pour un administrateur le plus souvent une instance de la console Windows PowerShell ou un job.

Une session concerne les variables, les alias, les fonctions, les drivers, les providers, déclarés dans le runspace. On peut, à partir de la console, obtenir le détail de cet état en consultant la variable automatique suivante :

```
$ExecutionContext.SessionState
```

```
Drive                           : System.Management.Automation.DriveManagementIntrinsics
Provider                        : System.Management.Automation.CmdletProviderManagementIntrinsics
Path                            : System.Management.Automation.PathIntrinsics
PSVariable                      : System.Management.Automation.PSVariableIntrinsics
LanguageMode                    : FullLanguage
UseFullLanguageModeInDebugger : False
Scripts                         : {*}
Applications                    : {*}
Module                          :
InvokeProvider                  : System.Management.Automation.ProviderIntrinsics
InvokeCommand                   : System.Management.Automation.CommandInvocationIntrinsics
```

Chaque runspace posséde un état de session qui lui est propre.

Voir aussi : http://msdn.microsoft.com/en-us/library/ms714451(VS.85).aspx

9.1 L'état de session d'un module

Un module ajoute un niveau d'état imbriqué dans l'état de session d'un runspace.Un état de session de module est créé chaque fois qu'un module ou un module imbriqué est importé dans la session. Quand un module exporte un élément comme un cmdlet, une fonction ou une varaible, une référence à cet élément est ajoutée à l'état de session globale de la session. Toutefois, lorsque l'élément est exécuté, il est exécuté dans l'état de session du module.

Si on exécute dans un module l'instruction précédente, on constate que le membre *Module* est renseigné :

```
$ExecutionContext.SessionState
```

```
Drive                           : System.Management.Automation.DriveManagementIntrinsics
Provider                        : System.Management.Automation.CmdletProviderManagementIntrinsics
Path                            : System.Management.Automation.PathIntrinsics
PSVariable                      : System.Management.Automation.PSVariableIntrinsics
LanguageMode                    : FullLanguage
UseFullLanguageModeInDebugger : False
Scripts                         : {*}
Applications                    : {*}
Module                          : MyModule
InvokeProvider                  : System.Management.Automation.ProviderIntrinsics
InvokeCommand                   : System.Management.Automation.CommandInvocationIntrinsics
```

L'état de session d'un module lui permet, de manière simplifiée, de créer des objets Powershell (variables, alias, fonctions,…) publics, mais surtout privés. Cet état nécessite un mécanisme de cloisonnement impliquant une notion de portée, bien que celle-ci ne soit pas explicitement définie comme peut l'être la portée nommée *global.*

Cet état implique également un contexte d'exécution, puisqu'il y a imbrication d'état de session, la question que l'on se posera à un moment ou à un autre est :

Dans quel contexte est exécuté mon code, dans l'état de session du runspace ou dans celui du module ?

9.2 Des données à portée de main

Par défaut si un objet n'existe pas dans la portée du module, la recherche s'effectuera dans la portée globale.

Afin de mettre en évidence ce problème de portée, la fonction de l'exemple suivant utilise un bloc de script nommé ***sb*** et une variable nommée ***Datas***.

Ouvrez une nouvelle console Powershell, placez-vous dans le répertoire des scripts de démos et chargez le module EtatSession :

```
# cd répertoire d'installation des scripts du tutoriel
ipmo ".\Gotcha\EtatSession.psm1"
```

Ce module est chargé dans la session courante. Nous utiliserons dans un premier temps la fonction suivante :

```
function TestData {
  param(
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [ValidateNotNull()]
      $Datas,
        [Parameter(Position=1, Mandatory=$true)]
        [ValidateNotNullOrEmpty()]
      [Scriptblock] $sb
  )
    &$sb
} #TestData
```

Le code de cette fonction exécute un bloc de script pour chaque objet reçu en paramètre.

Déclarons, dans la portée de la session PowerShell, une variable nommée **Info** :

```
$Info=New-object psobject -property @{nom="Jean"; age=25}
```

On passera en paramètre un script block référençant la variable *Info*.

```
TestData -Datas $Info -sb {Write-Host $Info.Nom -fore Green }
```

```
Jean
```

Le résultat affiché correspond bien au contenu du membre 'nom' de la variable *Info*.

Maintenant utilisons le paramètre **-Datas** dans le code du script block, désormais on y référence la variable interne à la fonction TestData, c'est-à-dire **$Datas**.

```
TestData -Datas $Info -sb {Write-Host $Datas.Nom -fore Green }
```

```
#
```

L'appel à Write-host n'affiche rien, ce comportement est dû à l'inexistence de la variable *$Datas,* que l'on peut vérifier ainsi :

```
TestData -Datas $Info -sb {Get-Variable Datas}
```

```
Get-Variable : Impossible de trouver une variable assortie du nom « Datas ».
```

Pourtant notre fonction ***TestData*** est définie dans notre module et celle-ci définie bien un paramètre nommé *-Datas*, on s'attend donc à référencer un variable existante, ce qui n'est pas le cas.

Créons une variable *$Datas* dans la session appelante, et exécutons de nouveau l'appel précédent :

```
#déclare $Datas dans la porté courante
$Datas=New-object psobject -property @{nom="Pierre"; age=35}
TestData -Datas $Info -sb {Write-Host $Datas.Nom -fore Green }
```

```
Pierre
```

Cette fois-ci le code fonctionne, tout du moins il affiche un résultat, malheureusement on s'attend à afficher le contenu de la variable *$Infos* (Jean), et pas celui de la variable *$Datas* (Pierre).

Ce qui se passe est que le script block est déclaré dans l'état de session de l'appelant, la console, et pas dans celle du module appelé :

```
$sb={Write-Host $Datas.Nom -fore Green}
TestData -Datas $Info -sb $sb
```

```
Pierre
```

On peut le constater en affichant les propriétés du scriptblock :

```
$sb|select *
```

```
IsFilter     : False
StartPosition : System.Management.Automation.PSToken
File         :
Attributes   : {}
Module       :
```

Le membre nommée *Module* n'est pas renseigné, ce qui signifie qu'il référence la session d'état de l'appelant.

Essayons cette fois avec la fonction TestData2 :

```
function TestData2 {
  param(
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [ValidateNotNull()]
      $Datas,
        [Parameter(Position=1, Mandatory=$true)]
        [ValidateNotNullOrEmpty()]
      [Scriptblock] $sb
  )
    $Sb|select *
    &$sb
} #TestData2
```

qui affiche les informations de son paramètre **–Sb**.

```
TestData2 -Datas $Info -sb $sb
```

Ici aussi, bien que l'on soit dans la portée du module, le membre nommée *Module* est vide :

```
IsFilter     : False
StartPosition : System.Management.Automation.PSToken
File         :
Attributes   : {}
Module       :
Pierre
```

9.2.1 Liaison de scriptblock à une session de module

La solution consiste à lier explicitement le code de notre scriptblock à l'état de session du module à l'aide de la méthode **NewBoundScriptBlock** :

```
$SbBounded=$MyInvocation.MyCommand.ScriptBlock.Module.NewBoundScriptBlock($sb)
&$SbBounded
```

Essayons avec une troisième fonction, TestData3 :

```
function TestData3 {
  param(
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [ValidateNotNull()]
      $Datas,
        [Parameter(Position=1, Mandatory=$true)]
        [ValidateNotNullOrEmpty()]
      [Scriptblock] $sb
  )
     #On lie explicitement le scriptblock dans la portée du module,
     #sinon la variable $Datas est recherchée dans la portée de l'appelant
     # ce qui générerait une erreur : VariableIsUndefined
```

```
  &($MyInvocation.MyCommand.ScriptBlock.Module.NewBoundScriptBlock($sb))

} #TestData3
```

La construction de l'appel est identique :

```
TestData3 -data $Info -sb {Write-Host $Datas.Nom -fore Green}
```

```
Jean
```

Cette fois le résultat est bien celui attendu. Cette méthode crée un scriptblock dans l'état de session du module, son code y recherchera en premier lieu les variables référencées, si toutefois une variable référencée n'existe pas, la recherche se fera dans l'appelant, comme le montre la fonction TestData4 :

```
$DataSession=New-object psobject -property @{nom="Claude"; age=45}
$Info=New-object psobject -property @{nom="Jean"; age=25}
TestData4 -datas $Info -sb {Write-Host $Info.Nom -fore Green; Write-Host
$DataSession.Nom -fore White }
```

```
IsFilter      : False
StartPosition : System.Management.Automation.PSToken
File          :
Attributes    : {}
Module        : EtatSession
Jean
Claude
```

Le fait de lier le code d'un scriptblock au module modifie l'ordre de recherche des variables qu'il référence.

9.2.2 Accès à l'état de session de l'appelant

Dans le tutoriel sur les fonctions avancées :

http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/Les-fonctions-et-scripts-avancees-sous-PowerShell-version-2

Vous trouverez, au chapitre 5.11.2, un exemple de récupération du contenu d'une variable automatique **déclarée dans la portée de l'appelant.**

On peut donc adresser soit la portée du module soit celle de la session primaire (la console).

Consulter la démo présente dans le répertoire nommé 'Régles de validation'.

9.3 Paramètrer un module à l'aide d'arguments

Le cmdlet **Import-Module** ne permet pas de spécifier sur la ligne de commande un nom de paramètre déclaré dans la clause *Param* :

```
#Syntaxe impossible
Import-Module MyModule.psm1 -DataPath "C:\temp" -Switch
```

On paramètrera le module à l'aide du paramètre *ArgumentList*, puis dans le code du module, on accédera aux objets passés via cet argument en utilisant les noms des paramètres de la clause *Param* :

```
#Syntaxe d'appel à disposition
```

```
Import-Module MyModule.psm1 -ArgumentList "C:\temp",$True
```

Placez-vous dans le répertoire des scripts de démos :

```
.\New-FileModuleArgumentList.ps1
```

Premier appel sans préciser le paramétre *ArgumentList :*

```
Import-Module mymodule
```

```
DataPath= Switch=False
```

Second appel en précisant le paramétre *ArgumentList :*

```
Import-Module mymodule -ArgumentList C:\temp,$True -force
```

```
DataPath=C:\temp Switch=True
```

Voir aussi cette astuce pour valider les arguments :

http://www.nivot.org/post/2009/10/22/PowerShell20ModuleInitializers.aspx

Note : Il reste possible de déclarer des attributs sur les paramètres du module, bien que le message émit en cas d'erreur soit dans ce cas plus proche d'un message concernant une variable contrainte qu'un message d'erreur sur un paramètre, comme c'est le cas avec l'astuce indiqué.

9.4 Import d'un module dans la session globale

Il peut être nécessaire de charger un module dans l'état de session globale, pour cela le cmdlet **Import-Module** propose l'argument *Global*, de type Switch. Cette possibilité ne s'applique qu'aux modules de script.

10 Module et remoting implicite

Powershell offre, au travers d'un proxy, la possibilité d'utiliser en local un module installé sur une machine distante, je vous laisse consulter ces tutoriaux en Anglais :

http://technet.microsoft.com/en-us/magazine/ff720181.aspx

http://www.mikepfeiffer.net/2011/09/how-to-manage-your-exchange-2010-organization-with-powershell-implicit-remoting-over-the-internet/

Un tutoriel dédié au remoting :

http://www.ravichaganti.com/blog/?page_id=1301

11 Conclusion

L'usage de module ne présente pas de difficulté particulère, en revanche dés que l'on aborde le développement de module on se confronte rapidement avec le problème récurrent sous Powershell, à savoir quel comportement propose l'implémentation. La seconde édition de l'ouvrage de Bruce Payette *PowerShell in action* couvre ce sujet sur plus de 70 pages, et comporte des informations que l'on ne trouve pas par ailleurs.

Les multiples liens compilés dans ce document donnent un aperçu des problèmes de documentation rencontrés dés que l'on sort des sentiers battus.

Je pense que ce tutoriel contient suffisament d'informations permettant à la majorité des lecteurs de bien débuter sur le sujet tout en évitant quelques pièges, mais je lui laisse le soin de pratiquer ce sujet ☺

A vos modules ! Prêt ?