Le versionning de module sous Powershell

Par Laurent Dardenne, le 14/07/2016.

Version 1.0

Niveau

Conçu avec Powershell version 5.0.10240.16384 - Windows 10 64 bits.

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

Les fichiers sources :
http://ottomatt.pagesperso-orange.fr/Data/Tutoriaux/Powershell/VersionningDeModule/Sources.zip

Chapitres

1 Versionner

En informatique une version est « *l'état d'un logiciel mis à la disposition des utilisateurs, comportant les corrections et améliorations apportées à l'état précédent.* ».

Un numéro de version est semblable à un repère temporel attaché à un élément : OS, logiciel, fichier, objet, etc. Il permet de déterminer un état parmi d'autres.

Une mise à jour fait passer dans un nouvel état, le plus souvent le plus récent, l'inverse étant nommé restauration, c'est-à-dire une remise dans un état acceptable. La nouveauté n'étant pas sans danger ☺.

Aborder cette notion de version implique de préciser ce qu'est un numéro de version et de savoir quand et comment changer de numéro version, le document ‘gestion sémantique de version’ offre une première réponse.

Une autre question liée est de recenser les implications liés aux modifications, ce blog contient un document nous donnant quelques indications, bien qu'il soit spécifique au développement C#.

Ce dernier document contient des définitions en anglais de terme liés au versionning tels que *binary compatibility, backwards compatibility* ou encore *forwards compatibility*.

Dans ce présent document je me limite à la gestion de version d'un module sous Powershell. La gestion de version de module présentée ici concerne Powershell version 3 et supérieure.

Notez qu'un gestionnaire de source est nécessaire pour le suivi des versions.

A voir également The numerology of the build, redux et Version Numbering for All Releases .

1.1 Version de module

Le versionnage (*versioning*) n'est pas qu'un numéro de version, c'est également un mécanisme de récupération de métadonnées, nos informations de version.

Celui-ci peut également être lié à un mécanisme de gestion de dépendances qui peut aller du simple contrôle de prérequis à leurs chargement automatique.

Les scripts et fonctions Powershell ne sont, pour le moment, pas versionnés nativement, ils ne possèdent donc pas d'informations de version accessibles au travers du cmdlet **Get-Command**.

Seul un module possède un numéro de version *via* son manifeste qui attache des métadonnées à un module.

A partir de la version 5 de Powershell la classe de métadonnée ***CommandInfo*** propose la propriété *Version*, mais le contenu de celle-ci dépend d'un module qui héberge la commande. Le numéro de version d'une fonction peut donc contenir *$null* ou le numéro de version du module l'hébergeant.

2 Versionner un module

Afin d'éviter des doublons, des modules d'auteurs différent, mais portant le même nom, Powershell ajoute la notion d'identité.

2.1 Version et identité

Si on utilise un seul module nommé *Computer* son nom suffit.

Pour le versionner, on doit associer son nom et un numéro de version sous réserve de créer un manifeste de module :

```
@{
   RootModule = 'Computer.psm1'
   ModuleVersion = '1.0'
}
```

Dans un manifeste, seule la clé *ModuleVersion* est requise.

Pour différencier deux modules nommés *Computer* ayant la même version mais d'auteurs différents on doit leur ajouter une troisième information unique afin de les différencier :

```
@{
  RootModule = 'Computer.psm1'
  ModuleVersion = '1.0'
  GUID = 'a5d7c151-56cf-40a4-839f-0019898eb324'
}
```

La clé 'GUID' du manifeste de chaque module remplie ce rôle, chaque nouvelle version d'un module utilisera le même GUID, seul son numéro de version sera incrémenté :

```
@{
  RootModule = 'Computer.psm1'
  ModuleVersion = '1.1'                          # Différent
  GUID = 'a5d7c151-56cf-40a4-839f-0019898eb324' # Identique
}
```

L'identité d'un module est donc constituée des champs suivants :

- ModuleName, de type **[String]**
- ModuleVersion, de type **[Version]**
- GUID, de type **[Guid]**

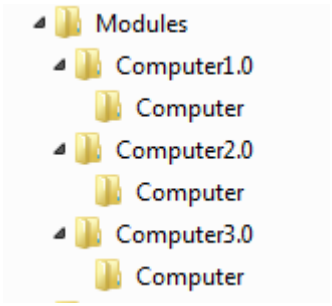
Depuis la version 3 Powershell héberge ces informations dans la classe ModuleSpecification.

Note : Seule la version livrée avec Powershell 5.0 propose la propriété '*RequiredVersion*'.

2.2 Installation de module versionné

Lors de l'appel à **Import-Module**, Powershell recherche le ou les modules demandés dans les répertoires précisés dans la variable d'environnement *%PSModulePath%*.

Une fois notre module 'Computer' versionné, son installation doit respecter la structure suivante



Le répertoire ***Modules*** contient toutes les versions de notre module, chaque version est hébergée dans un répertoire ***ComputerX.Y.Z*** et chacun de ces répertoires contient un sous répertoire '***Computer***'. Cette structure d'hébergement de module est bien formée.

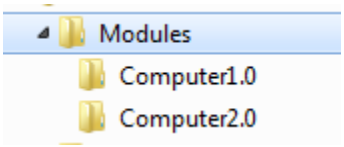
Pour chaque version installée, la variable d'environnement *%PSModulePath%* doit contenir le nom de chemin de chaque version :

Votre_CheminComplet\Modules\Computer1.0 ; Votre_CheminComplet\Modules\Computer2.0 …

On peut utiliser le répertoire de modules de Powershell ou un répertorie spécifique :

Votre_CheminComplet\NomEntreprise\Computer1.0 …

Si la structure d'hébergement de module est mal formée :



Ce qui donne par exemple le chemin d'accès suivant :

```
…\Modules\Computer1.0\Computer.psm1
```

L'import du module échouera :

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

Notez que le nommage du nom de répertoire tel que *ComputerX.Y.Z* n'est pas normé, il pourrait très bien se nommer *MaVersion1.0* :

```
…\Modules\Maversion1.0\Computer\Computer.psm1
```

Avec le premier nommage on sait de quoi on parle avec le second un peu moins. Ici **c'est le manifeste du module qui porte le numéro de version** pas le nom du répertoire l'hébergeant :

```
…\Modules\Maversion1.0\Computer\Computer.psd1
```

L'inconvénient majeur est que pour chaque version d'un module l'on doit ajouter le nom de son répertoire d'accès dans la variable d'environnement *%PSModulePath%*.

2.3 Chargement par un nom

Si on ne versionne pas un module l'usage de son nom suffit :

```
cd « VotreRépertoire de source\Modules »
.\DemoComputer.ps1          # Modifie $env:PSmodulePath
Import-Module 'Computer'
Get-ComputerVersion
Version 1.0
Get-Module
ModuleType Version    Name        ExportedCommands
---------- -------    ----        ----------------
Script     1.0        computer    Get-ComputerVersion
```

Ici Powershell charge le premier module de nom '*Computer*' trouvé dans $env:PSModulePath. Le comportement de l'autoloading de module est identique. L'ordre de déclaration des chemins des différentes versions d'un module influence donc cette commande.

Si dans le script *DemoComputer.ps1* on inverse l'ordre des chemins dans $env:PSModulePath :

Votre_CheminComplet\Modules\Computer2.0 ; Votre_CheminComplet\Modules\Computer1.0 …

La même suite d'instructions renverra la version 2.0.

2.4 Chargement d'une version spécifiée

Attention, l'usage du paramètre *–Version* ne remplit pas cette fonction, l'exemple ci-dessous utilise les chemins ordonnés suivants :

..\Modules\Computer1.0 ; ..\Modules\Computer3.0 ; ..\Modules\Computer2.0 …

```
# !!!! Nouvelle session
cd « VotreRépertoire de source\Modules »
.\DemoComputer.ps1
Import-Module 'Computer' –Version 2.0
Get-ComputerVersion
Version 3.0
Get-Module
ModuleType Version    Name        ExportedCommands
---------- -------    ----        ----------------
Script     3.0        computer    Get-ComputerVersion
```

Le paramètre *–Version* est un alias du paramètre **-MinimumVersion** qui charge le premier module dont le numéro de version est égal ou supérieure au numéro de version indiqué.

Ici aussi l'ordre de déclaration des chemins influence cette commande.

2.4.1 Le paramètre RequiredVersion

Pour charger une version spécifique indépendamment de l'ordre des chemins, on utilisera le paramètre *–RequiredVersion* :

```
# !!!! Créez une nouvelle session Powershell
cd « VotreRépertoire de source\Modules »
.\DemoComputer.ps1
Import-Module 'Computer' –RequiredVersion 2.0
Get-ComputerVersion
Version 2.0
Get-Module
ModuleType Version    Name        ExportedCommands
---------- -------    ----        ----------------
Script     2.0        computer    Get-ComputerVersion
```

On couple définitivement ce code avec un numéro de version, en revanche un code utilisant le paramètre *-MinimumVersion* chargera automatiquement la dernière version du module **sous réserve d'ordonner les chemins de recherche du plus grand au plus petit** :

..\Modules\Computer3.0 ; ..\Modules\Computer2.0 ; ..\Modules\Computer1.0 …

2.5 Chargement d'une version identifiée

Le cas de doublons de module nommé à l'identique et de même version n'est pas couvert par le paramètre *–RequiredVersion*. Une fois encore l'ordre de déclaration des chemins influencera cette commande.

Pour ce faire sous Powershell version 5, le cmdlet Import-Module contient un nouveau paramètre nommé *–FullyQualifiedName*.

2.5.1 Le paramètre FullyQualifiedName

Ce paramètre utilise la notion d'identité vue précédemment afin de lever toute ambiguïté sur le module concerné. Il attend une hashtable déclarant les clés suivantes :

```
@{
   ModuleName = "modulename";
   ModuleVersion = "version_number";
   RequiredVersion = "version_number";
   Guid = "GUID"
}
```

Seule la clé *Guid* est optionnelle. Les clés *ModuleVersion* et *RequiredVersion sont* mutuellement exclusives. Ces clés correspondent aux propriétés de la classe ModuleSpecification.

Ainsi on est assuré de charger le module voulu :

```
# !!!! Nouvelle session
cd « VotreRépertoire de source\Modules »
.\DemoComputer.ps1
$FQN=@{
  ModuleName = 'Computer'
  ModuleVersion = '1.0'
  GUID = 'a5d7c151-56cf-40a4-839f-0019898eb324'
}
Import-Module –FullyQualifiedName $FQN –PassThru |Select GUid,Version
Guid                                 Version
----                                 -------
a5d7c151-56cf-40a4-839f-0019898eb324 3.0
```

On charge bien le module avec le GUID demandé, mais la version n'est pas celle demandée. Le comportement lié à la clé *ModuleVersion* est identique à celui du paramètre ***-MinimumVersion***. La version chargée dépend de l'ordre des chemins de recherche déclaré dans le Path.

Pour charger le module avec le GUID demandé et la version demandée, il faut utiliser la clé *RequiredVersion* :

```
$FQN=@{
  ModuleName = 'Computer'
  RequiredVersion = '1.0'
  GUID = 'a5d7c151-56cf-40a4-839f-0019898eb324'
}
Import-Module –FullyQualifiedName $FQN –PassThru |Select GUid,Version
Guid                                 Version
----                                 -------
a5d7c151-56cf-40a4-839f-0019898eb324 1.0
```

Cette fois, et sans modifier l'ordre des path, on charge la version demandée

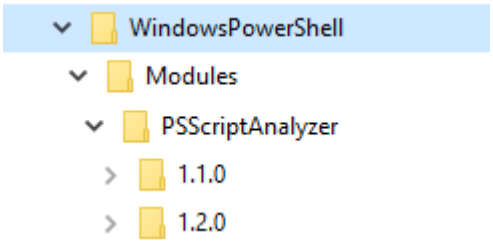
Notez que le paramétrage suivant est similaire au comportement du paramètre –***RequiredVersion*** vu précédemment :

```
$FQN=@{
  ModuleName = 'Computer'
  ModuleVersion = '1.0'
}
Import-Module –FullyQualifiedName $FQN
```

L'omission de la clé *GUID* fait que cette commande se basera sur l'ordre de déclaration des chemins.

2.6 Installation versionnée avec la version 5

Powershell v5 est livrée avec le module PackageManagement, celui-ci crée la structure suivante pour les modules versionnés :



Cette structure simplifie l'installation, chaque répertoire d'un module héberge directement ses versions : *..\Modules\PSScriptAnalyzer\1.1.0\PSScriptAnalyzer.psm1*

Le nom du répertoire doit pouvoir être converti en un numéro de version :

```
[Version]'1.0'

Major  Minor  Build  Revision
-----  -----  -----  --------
1      0      -1     -1

[Version]'1'
Cannot convert value "1" to type "System.Version". Error: "Version string
portion was too short or too long."
[Version]'Module1.0'
Cannot convert value "Module1.0" to type "System.Version". Error: "Input
string was not in a correct format."
```

Attention, le nom du répertoire doit être strictement identique au numéro de version indiqué dans le manifeste. Le numéro de version 1.1.0 est différent de 1.1 :

```
[Version]'1.1.0'

Major  Minor  Build  Revision
-----  -----  -----  --------
1      1      0      -1

[Version]'1.1'
Major  Minor  Build  Revision
-----  -----  -----  --------
1      1      -1     -1
```

Tout comme 1.0 est différent de 1.0.0.0

MSDN : « The value of T:System.Version properties that have not been explicitly assigned a value is undefined (-1). »

Si on ajoute un répertoire, sans contenu, dont le nom respecte cette convention, par exemple 4.0, l'outil *ProcessMonitor.exe* nous informe que Powershell y recherche un manifeste :

```
..\Modules\PSScriptAnalyzer\4.0\PSScriptAnalyzer.psd1    NAME NOT FOUND
..\Modules\PSScriptAnalyzer\1.2.0\PSScriptAnalyzer.psd1  SUCCESS
```

Cette recherche, chargera donc toujours la dernière version d'un module.

On peut aussi utiliser :

```
Trace-command -Name PathResolution -Expression {ipmo Computer } -PSHost
```

S'il existe dans le path deux répertoires de recherche contenant chacun des versions différentes, cette recherche suit l'ordre de déclaration des chemins et s'arrête dès qu'elle trouve un manifeste pour le module demandé. Toutes les versions d'un module doivent donc être placées dans un seul répertoire. A moins d'utiliser le paramètre *–RequiredVersion* qui parcourt tous les répertoires de recherche déclarés.

La version 4 de Powershell ne supporte pas cette structure, si on ajoute le chemin de recherche :

Votre_CheminComplet\ Modules\PSScriptAnalyzer\1.1.0\

Celle-ci recherche le chemin suivant :

..\Modules\PSScriptAnalyzer\1.1.0\PSScriptAnalyzer\PSScriptAnalyzer.psd1

Note :

La gestion de package (PackageManagement/OneGet) disponible sous la version 4 ne gére pas le versionning des noms de répertoire, il écrase la version précédente ce qui, je suppose, lui évite de modifier la variable d'environnement %Path%.

2.7 Récapitulatif

La colonne ‘Arborescence’ indique le mode d’installation d’un module versionné :

PS	Arborescence	Paramètre	Comportement
V4	v4	*Aucun : AutoLoading*	On charge le premier module trouvé. La version et le GUID dépendent de l'ordre du path.
V4	v4	Name	On charge le premier module trouvé. La version et le GUID dépendent de l'ordre du path.
V4	v4	MinimumVersion	On charge le premier module trouvé dont la version est supérieure ou égal à celle indiquée. La version et le GUID dépendent de l'ordre du path.
V4	v4	RequiredVersion	Pas de gestion des doublons. On charge la version demandée. Le GUID dépend de l'ordre du path.
V5	v4	FullyQualifiedName : clé '*ModuleVersion*'	**Gestion des doublons**. On charge le module du GUID demandé. La version dépend de l'ordre du path.
V5	v4	FullyQualifiedName : clé '*RequiredVersion*'.	Gestion des doublons. On charge la version demandée avec le GUID demandé.
V5	v5	*Aucun : AutoLoading*	On charge la derniére version. Le GUID dépend de l'ordre du path.
V5	v5	Name	On charge la dernière version. Le GUID dépend de l'ordre du path.
V5	v5	MinimumVersion	On charge la dernière version. Le GUID dépend de l'ordre du path.
V5	v5	RequiredVersion	Pas de gestion des doublons. On charge la version demandée. Le GUID dépend de l'ordre du path.
V5	v5	FullyQualifiedName : clé '*ModuleVersion*'.	**Gestion des doublons**. On charge le module du GUID demandé. On charge la dernière version.
V5	v5	FullyQualifiedName : clé '*RequiredVersion*'	**Gestion des doublons**. On charge la version demandée du GUID demandé.

2.8 Dépendances

Il est possible de créer une dépendance entre deux modules via un manifeste et également entre un script et un ou plusieurs modules à l'aide de la clause #Requires placée en début de script :

```
#Requires -Modules
@{ModuleName="Computer";ModuleVersion='1.0';GUID='a5d7c151-56cf-40a4-839f-
0019898eb324'}
Get-ComputerVersion
Get-Module Computer |Select GUid,Version
```

On obtient bien le chargement du module demandé :

```
cd « VotreRépertoire de source\Modules »
.\DemoRequires.ps1
Version 1.0

Guid                                 Version
----                                 -------
a5d7c151-56cf-40a4-839f-0019898eb324 1.0
```

Mais cette clause dépend de l'ordre des chemins de recherche dans le path, on peut donc se retrouver avec une autre version du module :

```
Version 3.0

Guid                                 Version
----                                 -------
a5d7c151-56cf-40a4-839f-0019898eb324 3.0
```

La hashtable spécifiée est convertie en une instance de la classe ModuleSpecification, ce qui fait que seule la version 5 de Powershell supporte la clé *RequiredVersion*.

Attention, cette instruction pose un problème si on utilise des modules de même nom mais d'identité différente, voir ce bug ainsi que l'archive 'VotreRépertoire de source \Tests\Test-bug-requires.zip'

Le répertoire « VotreRépertoire de source\Tests » contient des scripts Pester détaillant les différents comportements abordés précédement.

Le script principal se nomme *MainTestModules.ps1*.

2.9 Attribut Obsolete()

Afin d'indiquer des éléments obsolètes la version 5 propose l'attribut **[Obsolete()].**

L'élément ainsi marqué dans une nouvelle version fonctionne toujours, mais Powershell averti (*Warning*) l'utilisateur ou le concepteur qu'un élément plus récente est disponible.

Notez que la présence de cet attribut ne déclenche pas d'erreur de parsing sous Powershell version 4 et inférieure, ni l'affichage d'un avertissement. Cet attribut fonctionne pour un script, une fonction ou un paramètre.

2.9.1 Script

La présence de la clause *Param*, même sans paramètre, est obligatoire :

```
@'
 [Obsolete()]
 param()
 write-host "Test script"
'@ > c:\temp\Test.ps1
c:\temp\Test.ps1
WARNING: The command 'Test.ps1' is obsolete.
Test script
```

2.9.2 Fonction

La présence de la clause *Param*, même sans paramètre, est obligatoire. Il est possible de préciser un message d'avertissement additionnel :

```
function Test {
  [Obsolete('Utilisez la nouvelle version.')]
 Param() #mandatory
 Write-host "Test function"
}
Test
WARNING: The command 'Test' is obsolete. Utilisez la nouvelle version.
Test function
```

2.9.3 Paramètre

Il est possible de déprécier un paramètre :

```
function Test {
 param (
    [Obsolete()] $Name
 )
 Write-host "Test function parameter"
}
```

```
Test
Test function parameter
Test -Name x
WARNING: Parameter 'Name' is obsolete.
Test function parameter
```

Sachez que la présence du paramètre *error* :

```
[System.ObsoleteAttribute]::new
OverloadDefinitions
-------------------
System.ObsoleteAttribute new()
System.ObsoleteAttribute new(string message)
System.ObsoleteAttribute new(string message, bool error)
```

N'est pas pris en compte sous Powershell, uniquement en C#. Dans ce cas l'exception déclenchée est du type *[System.Management.Automation.PSInvalidOperationException]*.

En revanche, le comportement de cet attribut dépend du contenu de la variable de préférence ***$WarningPreference*** :

```
$WarningPreference='Stop'
Test -Name x
WARNING: Parameter 'Name' is obsolete.
Test : The running command stopped because the preference variable
"WarningPreference" or common parameter is set to
Stop: Parameter 'Name' is obsolete.
```

3 Versionning d'instance

Sous Powershell version 5 un module est à mon avis le plus approprié pour contenir une définition de classe native. Le plus souvent un traitement distant récupère simplement des données, si on souhaite réhydrater (reconstruire) une instance, la classe utilisée doit être identique sur le(s) distant(s) et le local.

On peut envisager d'utiliser DSC pour s'assurer que chaque nœud possède les versions concernées, mais le contrôle de version doit se faire dans le code de nos traitements.

Je vous propose dans ce chapitre d'étudier un contrôle de version pour les instances de classes Powershell hébergées dans un module. Puisqu'un module propose déjà un mécanisme de versionning on peut en réutiliser une partie, la condition est que chaque instance possède les informations de version du module qui héberge sa classe. Pour coupler cette information de version on est tenté d'ajouter une propriété de classe, mais celle-ci n'est pas accessible par l'instance. Mais surtout lors de la transmission Powershell/WSMan sérialise un objet et pas sa classe.

Une classe Powershell ne dispose pas de possibilité de versionning, pour le moment elles sont toutes de version 1.0. Et surtout on ne récupère pas les métadonnées de la classe.

La solution que je vous propose est d'utiliser le système d'extension de type (ETS). Ce n'est pas un mécanisme de sérialisation avec tolérance de version, loin de là ☺.

3.1 Principe

Chaque module utilisant cette approche doit déclarer son identité via un fichier de manifeste.

Ce qui permet de contrôler les versions des instances et si besoin de charger le module pour réhydrater les objets désérialisés. **On gère le versionning d'une association** (Instance, Module), une classe PowerShell version 5 pourra ainsi être réhydratée dans le contexte de son module.

Notez que le code d'un manifeste ne peut utiliser la variable $PSVersionTable.

3.1.1 Côté émetteur

Nous avons vu précédemment que, depuis la version 3**,** Powershell héberge les informations de version d'un module dans la classe [ModuleSpecification](). Il suffit donc d'ajouter à chaque instance que l'on souhaite versionner une propriété contenant ces informations :

```
$SB={
  Add-Type @'
using System;
 namespace Test
 {
   public class MaClasse
    {
        public string Message="Versionning";
        public void ShowMessage()
        {
         Console.WriteLine("Test de versionning");
        }
    }
}
'@
$object=new-object 'Test.MaClasse'
 Update-TypeData  -TypeName 'Test.MaClasse' -MemberType ScriptProperty `
                  -MemberName ModuleSpecification `
                  -SecondValue {Throw "ModuleSpecification is a read only
property."} `
                  -Value {
   Param()

   return [Microsoft.PowerShell.Commands.ModuleSpecification]@{
             ModuleName="TestMaClasse"
             ModuleVersion ='1.0'
             GUID='3ba397a3-c46a-4f3d-a6a2-95ce16156e50'
          }
  } -Force
  $Object
} #$sb
$Result=.$SB
```

Notez que la définition de la classe précédente ne contient pas de membre lié au versionning, celui-ci est ajouté via l'extension de type (Update-TypeData).

Ce qui permet de contrôler des instances de module tiers :

```
$Result
Message        ModuleSpecification
-------        -------------------
Versionning    @{ ModuleName = 'TestMaClasse'; Guid = '{3ba397a3-c46a-4...
```

Nous avons désormais nos informations de version associant notre instance à son module identifié :

```
Result.ModuleSpecification
Name           Guid                                   Version
----           ----                                   -------
TestMaClasse   3ba397a3-c46a-4f3d-a6a2-95ce16156e50   1.0
$Result.PSTypenames
Test.MaClasse …
```

Afin de simplifier je n'ai pas inséré cette définition de classe dans un module, de plus ce code utilise le cmdlet **Update-TypeData**, l'usage d'un fichier de type (.ps1xml) est préférable car lors du déchargement d'un module celui-ci supprime (implicitement) uniquement ce qu'il a ajouté, là où **Remove-TypeData** supprime toutes les entrées du type précisé.

Par exemple si votre module ajoute des membres à un type existant qui aurait été modifié par un autre module ou le profile utilisateur.

3.1.2 Côté récepteur

Cette propriété ajoutée à l'instance est par défaut sérialisée, l'association 'instance/module' persiste donc lors du remoting :

```
$Result=Start-job $sb|
 Wait-Job|
 Receive-job -AutoRemoveJob –Wait
$Result.PSTypenames
Deserialized.Test.MaClasse…
```

Depuis la version 5 Powershell réhydrate la classe *ModuleSpecification* :

```
$Result.ModuleSpecification
Name         Guid                                   Version RequiredVersion
----         ----                                   ------- ---------------
TestMaClasse 3ba397a3-c46a-4f3d-a6a2-95ce16156e50 1.0
```

Pour la version 4 on récupère une chaîne de caractères :

```
$Result.ModuleSpecification
@{ ModuleName = 'TestMaClasse'; Guid = '{3ba397a3-c46a-4f3d-a6a2-
95ce16156e50}' ; ModuleVersion = '1.0' }
```

On peut reconstruire l'instance ainsi :

```
$Regex= "@{ ModuleName = '(?<ModuleName>.[^']*)'; Guid =
'{(?<GUID>.[^}]*)}' ; ModuleVersion = '(?<ModuleVersion>.[^']*)' }"
if ($result.ModuleSpecification -match $Regex)
{
 $Matches.Remove(0)

$ModuleSpecification=[Microsoft.PowerShell.Commands.ModuleSpecification
]$Matches
}
```

Si l'usage du cmdlet **Invoke-Expression** ne vous inquiète pas, ceci est plus concis :

```
$s='[Microsoft.PowerShell.Commands.ModuleSpecification]'
$S +=$Result.ModuleSpecification
$o= Invoke-Expression $S
```

Une fois les données reçues on peut soit :

- charger le module, en ayant au préalable vérifié qu'aucun module associé à l'instance n'est présent en mémoire. Dans ce cas on tient compte de son nom et de son GUID uniquement :

```
$LoadedModules=@(
 Get-module -Name "TestMaClasse" |
  Where Guid -eq $Result.ModuleSpecification.Guid
)
if ($LoadedModules.Count -eq 0)
{ Import-Module -FullyQualifiedName $Result.ModuleSpecification }
```

- si un module de même nom et de même guid existe, contrôler les numéros de version entre une instance distante et une instance locale. Celle-ci provenant du même module (*Name, GUID*), mais de version différente (*Version*) :

```
elseif ("$(O.ModuleSpecification)" -ne "$(Result.ModuleSpecification)")
{ Throw "Version de module inadéquate : $(O.ModuleSpecification)"}
```

Sachez qu'il existe une classe **ModuleSpecificationComparer**, mais elle est 'internal', on teste donc l'égalité sur des chaînes.

Enfin s'il s'agit d'une instance sérialisé d'une classe Powershell ou C#, on peut la récréer avec le type d'origine, puisque désormais le type est accessible qu'il soit créé via un assembly/dll, *Add-Type* ou une classe native :

```
$Instance=[System.Management.Automation.LanguagePrimitives]::ConvertTo(
 $Result,
 [Test.maClasse], #Ou $InstanceType
 [System.Globalization.CultureInfo]::InvariantCulture
)
```

Etant donné la rédéfinition de la propriété '*ModuleSpecification'* basée sur un fichier de type, ETS ne peut gérer qu'une seule version puisqu'il est basé sur la propriété *PSTypenames* de chaque instance.

Notes :

Si, côté émetteur, on utilise ce principe sous Powershell v4, la propriété '*RequiredVersion*' n'existe pas, on utilisera donc la propriété '*ModuleVersion*', mais dans ce cas le comportement lors du chargement du module diffère. On gère bien les doublons en chargeant le module du GUID demandé, mais la version dépend de l'ordre du path.

Si, côté récepteur, on utilise ce principe sous Powershell v4, le cmdlet Cmdlet *Import-Module* ne proposant pas le paramètre ***–FullyQualifiedName*** on doit utiliser ***–RequiredVersion*** :

Emetteur	Récepteur	Propriété	Paramétre
v4	v4	ModuleVersion	-RequiredVersion
v4	v5	ModuleVersion* *Transformation possible	-FullyQualifiedName
v5	v4	ModuleVersion	-RequiredVersion
v5	v5	RequiredVersion	-FullyQualifiedName

Faute temps je n'ai pas testé le cas où l'instance est désérialisée par une classe dérivée de PSTypeConverter , ni étudié les combinaisons liées à l'usage du module PackageManagement.

Cette mécanique ne gère pas le chargement de modules côte à côte, sujet que nous aborderons plus avant.

3.2 Récupérer les informations de version

Les informations de version d'un module sont indisponibles lors de son chargement. On peut le constater en insérant le code suivant dans un des modules d'exemple du répertoire …\Sources\Modules :

```
#Computer.psm1

Function Get-Info {
 Write-Warning "Call Get-Info"
 Write-host "Name=$($ExecutionContext.SessionState.Module.Name)"
 Write-host "Version=$($ExecutionContext.SessionState.Module.Version)"
 Write-host "GUID=$($ExecutionContext.SessionState.Module.GUID)"
}
Get-Info
```

Le résultat lors du chargement du module :

```
Import-Module .\Computer.psd1
AVERTISSEMENT : Call Get-Info
Name=Computer
Version=0.0
GUID=00000000-0000-0000-0000-000000000000
```

Une fois le module chargé, les informations sont disponibles :

```
Get-Info
AVERTISSEMENT : Call Get-Info
Name=Computer
Version=1.0
GUID=a5d7c151-56cf-40a4-839f-0019898eb324
```

L'accès à ces informations de version nécessite de relire le manifeste du module, tout en évitant l'appel à une fonction d'initialisation une fois le module chargé.

Pour cela on utilise la classe d'attribut *ArgumentToConfigurationDataTransformationAttribute*.

Cet attribut attend un nom de fichier de configuration (.psd1) et transforme son contenu en une hashtable :

```
Function ConvertTo-ModuleSpecification {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory = $true)]

[Microsoft.PowerShell.DesiredStateConfiguration.ArgumentToConfigurationDataTransformation()]
        $data
    )


  return [Microsoft.PowerShell.Commands.ModuleSpecification]@{
            ModuleName=$Data.RootModule -Replace '.psm1'
            RequiredVersion =$Data.ModuleVersion
            GUID=$Data.Guid
          }
}#ConvertTo-ModuleSpecification
```

Seules les clés déclarées dans le manifeste se retrouvent dans la hashtable.

Si vous utilisez le cmdlet **Test-ModuleManifest** sachez que le contenu de la clé *RequiredModules* renseigne sa propriété 'Path' avec un nom de chemin inexistant. Ceci n'est pas un bug mais un comportement correct ☺

L'appel dans le module :

```
$Path=[System.IO.Path]::ChangeExtension($MyInvocation.MyCommand.ScriptBlock.Module.path, '.psd1')
$ModuleSpecification = ConvertTo-ModuleSpecification $Path
```

Désormais notre module contient une variable *$ModuleSpecification* mémorisant les informations de versions.

Note : La version finale de la fonction *ConvertTo-ModuleSpecification* est différente :

```
$ModuleSpecification = ConvertTo-ModuleSpecification -Data $Path
```

3.3 Créer le fichier de type

Pour créer automatiquement le fichier de type, on a besoin des informations de version du module (le chemin du manifeste) et du ou des noms des classes à versionner.

La fonction suivante crée le contenu du fichier de type :

```
Ipmo ToolsClass
$Params=@ {
 TypeName='Computer'
 Data='C:\Temp\VersionningInstance\Modules\Computer\2.0\Computer.psd1'
}
New-ModuleSpecificationMemberTypeData @params
<?xml version="1.0" encoding="utf-8"?>
<Types>
  <Type>
   <Name>Computer</Name>
   <Members>
     <ScriptProperty>
     <Name>ModuleSpecification</Name>
     <GetScriptBlock>
      [Microsoft.PowerShell.Commands.ModuleSpecification]@{
        ModuleName='Computer'
        RequiredVersion ='2.0'
        GUID='4b25a895-addd-45fd-82bc-b7b0ed80d3a6'
      }
…
```

Pour créer le fichier :

```
$path='C:\Temp\VersionningInstance\Modules\Computer\2.0\Computer.types.ps1xml'
New-ModuleSpecificationMemberTypeData @params|Set-Content $Path
```

Il reste à ajouter la clé suivante dans le fichier de manifeste du module concerné :

```
  TypesToProcess = @('Computer.types.ps1xml')
```

Certains scripts de démo ..*VersionningInstance\Demo-XX.ps1* nécessitent Powershell version 5.

4 Module côte à côte (side by side)

Cette notion de côte à côte est la possibilité d'utiliser simultanément plusieurs versions d'un module. Vous trouverez ici une démonstration complète. Dans les versions antérieures à la version 5, Powershell ne peut charger qu'une seule version d'un module.

Bien qu'il soit possible d'avoir plusieurs versions de module, le provider de fonction ne gérant pas le versionning, seule la dernière version d'une des fonctions communes sera accessible. Une possibilité est d'utiliser le paramètre *–Prefix* lors de l'import du module :

```
Ipmo -FullyQualifiedName  @{ModuleName='sxstest';Requiredversion='1.0'} -
Prefix v1
Ipmo -FullyQualifiedName  @{ModuleName='sxstest';Requiredversion='2.0'} -
Prefix v2
Get-Item function:Get-v[12]*
CommandType     Name                Version    Source
-----------     ----                -------    ------
Function        Get-v1SxSVersion    1.0        sxstest
Function        Get-v2SxSVersion    2.0        sxstest
```

Un autre point, tout en sachant que je n'ai pas exploré les différents types de module : workflow, DSC, CIM, concerne les possibles conflits :

- dans les ressources utilisées
    - le code natif, DLL ou Add-Type. Powershell peut charger deux versions d'une DLL sous réserve qu'elles possèdent un nom fort.
    - ETS, qui se base sur le PSTypeName d'une instance et pas le nom fort de sa classe, s'il existe. Notez qu'une collection générique référence le nom fort du type paramétré :

```
System.Collections.Generic.List`1[[System.Management.Automation.PSO
bject, System.Management.Automation, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35]]
```

- dans le code
    - les variables globales,
    - les classes natives qui obligent à manipuler des types et pas des raccourcis de type. De plus le nom complet du type référence le path d'installation du module :

```
System.Collections.Generic.List`1[[Computer,C__Temp_VersionningInst
ance_Modules_Computer_2.0_Computer.psm1, Version=0.0.0.0,
Culture=neutral, PublicKeyToken=null]]
```

    - les noms de répertoires utilisés, …

Le répertoire de démo *VersionAssemblies* contient deux scripts de test autour du versionning de DLL.

Cette possibilité semble donc limitée au module de type binaire (.psd1 + DLL versionnée)

Voir aussi : Exécution côte à côte dans le framework .NET