Chapitre 3
Création de modules

1. Introduction

Le mot "module" fait généralement penser à "modulaire". On a l'image d'un élément de base, massif, à partir duquel gravitent d'autres éléments de plus petite taille qui viennent compléter ses fonctionnalités. Ces éléments peuvent être retirés, agrémentés, ajoutés, voire transférés.

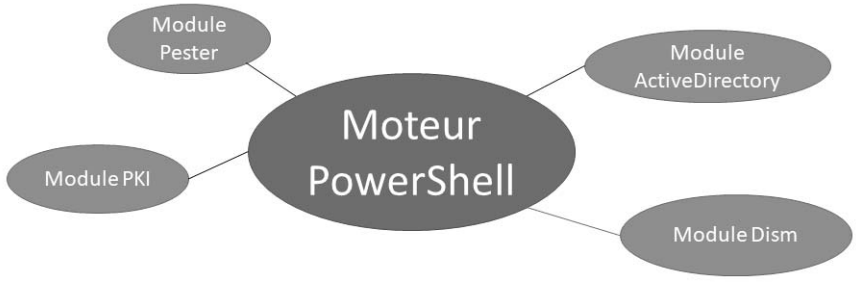

Illustration de la modulalité sous PowerShell

Il en est ainsi des modules PowerShell, même si ces derniers sont plus que de simples ajouts. En effet, l'ensemble des commandes PowerShell, notamment les commandes de base, est contenu dans des modules. Les modules ont fait leur apparition avec PowerShell 2.0. Ils ont supplanté les *snapins*, apparus dans la première version de PowerShell. Les snapins sont encore utilisés, mais Microsoft conseille l'utilisation des modules, bien plus simples à mettre en place.

Les modules permettent d'organiser correctement le code conçu. Mais surtout, ils facilitent le partage du code avec des collègues, voire avec la communauté entière.

Remarque

À l'heure où ces lignes sont écrites, Microsoft a recensé 11554 modules publiés sur sa plateforme de téléchargement PowerShell Gallery (voir le chapitre Gestion des modules et des packages). Cela peut donner une idée de leur importance réelle et de l'engouement qu'ils peuvent susciter.

De quels types de fichiers sont-ils constitués ? Et que peuvent-ils contenir ?

- PSM1 : fichier script. Il s'agit du fichier principal du module, son seul emploi suffit à définir un module. Il est exécuté pour charger l'ensemble des fonctions.

  Il peut également contenir des classes, des ressources DSC ou encore des variables.
- DLL : fichier binaire généralement compilé à partir de C#. Il peut contenir des applets de commandes, des providers, des ressources DSC, des variables ou encore des classes.
- PS1 : fichier script. Ce fichier déclare généralement des fonctions, des variables ou des classes.
- PSD1 : manifeste de module. Il s'agit d'un fichier permettant de renseigner un grand nombre d'éléments concernant le module. Il sera détaillé plus loin dans ce chapitre.
- PSRC : fichier de capacité de rôle JEA (*Just Enough Administration*). Ce type de fichier est abordé dans le chapitre Gestion à distance avancée.

Il existe plusieurs types de modules : module script, module binaire, module dynamique. Ils seront tous abordés dans les sections suivantes.

2. Module dynamique

La seconde possibilité pour créer un module est la commande `New-Module`. Elle crée un module dit dynamique. Il est chargé en mémoire et existe le temps de la session PowerShell. Cela signifie qu'il n'est pas écrit sur le disque, et donc invisible via la commande `Get-Module` (nous verrons plus loin qu'il existe un moyen de le faire apparaître). Les fonctions créées restent toutefois visibles par la commande `Get-Command`.

Arguments de la commande `New-Module` :

Paramètre	Description
`ArgumentList`	Indique un tableau de valeurs à passer au paramètre `-ScriptBlock`.
`AsCustomObject`	Renvoie le module sous forme de `CustomObject` lors de sa création. Il peut ainsi être assigné dans une variable. Les fonctions ou cmdlets exportées se présentent sous forme de méthode de cet objet.
`Cmdlet`	Filtre les cmdlets qui sont exportées du module. Pour cela, il est nécessaire d'indiquer une liste avec le nom des cmdlets à exporter. Par défaut, l'ensemble des cmdlets est exporté. Les *wildcards* sont supportées.
`Function`	Filtre les fonctions qui sont exportées à partir du `-ScriptBlock`. Même syntaxe que le paramètre `-Cmdlet`.
`Name`	Indique un nom de module. Étant donné que cela est obligatoire, si la valeur est nulle ou vide, PowerShell génère un nom de configuration aléatoire. Son nom débute alors par "`_DynamicModule`". Il sera suivi d'un GUID.
`ReturnResult`	Retourne la sortie du bloc de script, en plus de la création du module.
`ScriptBlock`	Indique le contenu du module dynamique. Ce bloc est délimité par des accolades. Les fonctions doivent être séparées par un saut de ligne ou un point-virgule. On peut dire qu'il s'agit de l'équivalent du fichier PSM1 pour un module script.

Deux fonctions vont maintenant être créées et intégrées dans le `-ScriptBlock` du module dynamique. Une première, `Get-PSInstalledHotFix`, qui permet de lister les mises à jour installées sur un poste. Et une seconde, `Rename-PSWindowUITitle`, qui a pour but de renommer la fenêtre PowerShell. Cela peut s'avérer pratique lorsqu'une multitude de fenêtres sont ouvertes et qu'il est difficile de s'y retrouver. Enfin, un alias, `gpih`, est également créé. Il pointe sur la fonction `Get-PSInstalledHotFix`.

Création du module dynamique

```
PS > New-Module -ScriptBlock {
   Function Get-PSInstalledHotFix {
        Get-WmiObject -Class Win32_QuickFixEngineering |
Select HotFixID,InstalledOn,Description
```

```
    }
    Function Rename-PSWindowUITitle ($NewName) {
        $host.UI.RawUI.WindowTitle = $NewName
    }
    Set-Alias -Name gpih -Value Get-PSInstalledHotFix
}

ModuleType Version  Name                        ExportedCommands
---------- -------  ----                        ----------------
Script     0 0       __DynamicModule_9a4c... {Get-PSInstalledHo...
```

Le paramètre `-Name` n'a pas été renseigné. Par conséquent, PowerShell a généré un nom aléatoire.

Pour lister les commandes

```
PS > Get-Command -Module "__DynamicModule_*"

CommandType  Name                   Version  Source
-----------  ----                   -------  ------
Function     Get-PSInstalledHotFix  0.0      __DynamicModule_...
Function     Rename-PSWindowUITitle 0.0      __DynamicModule_...
```

Les deux commandes issues du module sont bien présentes. Peut-on en dire autant de l'alias ?

```
PS >  get-alias gpih
get-alias : Cette commande ne trouve pas d'alias correspondant,
car l'alias avec name « gpih » n'existe pas.
Au caractère Ligne:1 : 1
+ get-alias gpih
+ ~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (gpih:String)
[Get-Alias], ItemNotFoundException
    + FullyQualifiedErrorId : ItemNotFoundException,Microsoft.
PowerShell.Commands.GetAliasCommand
```

On constate qu'une exception est générée, et par conséquent que l'alias n'a pas été exporté avec les fonctions. En effet, les modules dynamiques n'exportent pas les variables et les alias. Toutefois, il est possible de forcer ce comportement en utilisant la commande `Export-ModuleMember` :

```
PS > New-Module -ScriptBlock {
    Function Get-PSInstalledHotFix {
        Get-WmiObject -Class Win32_QuickFixEngineering |
Select HotFixID,InstalledOn,Description
    }
    Function Rename-PSWindowUITitle ($NewName) {
        $host.UI.RawUI.WindowTitle = $NewName
```

```
    }
    Set-Alias -Name gpih -Value Get-PSInstalledHotFix
    Export-ModuleMember -Alias gpih `
      -Function Get-PSInstalledHotFix,Rename-PSWindowUITitle
}
PS > get-alias gpih

CommandType  Name                                Source
-----------  ----                                ------
Alias        gpih -> Get-PSInstalledHotFix       __DynamicModule_9...
```

Nous reviendrons sur la commande `Export-ModuleMember` un peu plus loin dans ce chapitre.

Il a été dit précédemment que la commande `Get-Module` n'est pas dans la capacité de voir un module dynamique. L'astuce est d'ajouter la cmdlet `Import-Module` à gauche de la création du module, séparée par un pipeline :

```
PS > New-Module -ScriptBlock {…} | Import-Module
PS C:\Users\Nicol> Get-Module

ModuleType Version     Name                 ExportedCommands
---------- -------     ----                 ----------------
Script     0 0         __DynamicModule_...  {Get-PSInstalledHot...
```

3. Module binaire

Les modules binaires sont un type de modules assez spécifiques. Ils contiennent du code compilé à partir d'un langage .NET. On peut citer en exemple C#, ainsi que VB.NET, ou encore ASP.NET. Attention toutefois, même si un module binaire est proche d'un snapin par sa conception, ils sont tous les deux différents. L'installation d'un snapin nécessite des droits administrateur, ce qui n'est pas le cas d'un module binaire. Une simple copie suffit, tout comme les modules script (sauf si l'on copie le module dans un dossier nécessitant une élévation de privilèges, comme Program Files).

La conception de cmdlets à travers un langage .NET est très intéressante. Elle permet d'avoir un temps d'exécution supérieur comparé aux fonctions PowerShell. Toutefois, leur découverte ne relève pas de ce livre. En effet, cela s'adresse à des développeurs. Mais de manière générale, il est fortement conseillé de s'intéresser de près au framework .NET, surtout si l'on cherche constamment la performance dans les scripts et fonctions PowerShell. Le .NET est extrêmement riche, avec des classes d'objets qui font gagner un temps non négligeable.

4. Module script

Les modules de type script sont les plus connus de par leur facilité et rapidité de création. Ils sont également simples à déployer, et plus faciles d'accès.

La création d'un module est relativement simple. Il peut d'ailleurs être stocké à différents endroits. Pour cela, il faut se référer au contenu de la variable d'environnement `$Env:PSModulePath`. Si le module créé n'est pas contenu dans un des répertoires de cette variable, alors il ne sera pas découvert automatiquement par PowerShell. Par ailleurs, il est tout à fait possible de modifier cette variable pour y ajouter ses propres chemins d'accès.

Remarque

Attention : si vous modifiez cette variable, il est très important de ne pas supprimer les valeurs par défaut qui sont déjà présentes.

Si l'on décide de se passer de `$Env:PSModulePath`, il est toujours possible d'importer le module manuellement, en indiquant chemin d'accès complet à l'applet de commande `Import-Module`. Pour connaître le contenu de la variable `$Env:PSModulePath` :

```
PS > $Env:PSModulePath.Split(';')
C:\Users\Nicol\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
```

Une fois l'emplacement trouvé, il suffit de créer un dossier qui porte le nom du module, puis un fichier PSM1. Il doit porter le même nom que le dossier. Ce fichier va contenir le code PowerShell, voire faire appel à des scripts PS1.

Exemple de création du module

```
PS > New-Item -Path $Home\Documents\WindowsPowerShell\Modules `
             -Name MyModuleTools -Type directory


PS > New-Item -Name MyModuleTools.psm1 -Type File `
-Path $Home\Documents\WindowsPowerShell\Modules\MyModuleTools
```

Pour le moment, le module est vide. On va donc lui ajouter quelques fonctions utiles. Il est également intéressant d'y inclure une fonction qui fait appel à une seconde fonction, assez longue, utilisée régulièrement.

Remarque

Les linuxiens sous Bash sont habitués à créer des alias pour appliquer un comportement par défaut à une commande. Sous PowerShell, les alias n'ont pas cette utilité. Il est nécessaire de déclarer une nouvelle fonction qui fera office d'alias et appellera la fonction ou la commande que l'on veut raccourcir. Ces nouvelles fonctions sont généralement appelées fonctions proxy.

Dans le cas présent, on va simplement ajouter deux fonctions dans le fichier PSM1 :

– La fonction `Get-PSSerialNumber` : elle récupère le numéro de série de l'ordinateur.
– La fonction `Get-PSFolderSiz` : elle détermine la taille d'un dossier avec l'ensemble des fichiers qui le composent. La valeur renvoyée est en mégaoctets.

Code des fonctions

```
Function Get-PSSerialNumber {
    (Get-CIMInstance -ClassName Win32_Bios).SerialNumber
}

Function Get-PSFolderSize {
    Param(
        [ValidateScript({Test-Path $_})]
        [String]$Path
    )
    (Get-ChildItem -Path $Path -Recurse -File | `
    Measure-Object -Property Length -Sum ).Sum /1MB
}
```

Il est maintenant possible de constater la prise en compte du module dans la découverte automatique des modules par PowerShell :

```
PS > Get-Module -ListAvailable

    Répertoire : C:\Users\Nicolas\Documents\WindowsPowerShell\Modules
ModuleType Version Name                    ExportedCommands
---------- ------- ----                    ----------------
Script     0.0     MyModuleTools           {Get-PSFolder...

    Répertoire : C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules


ModuleType Version   Name                  ExportedCommands
---------- -------   ----                  ----------------
Manifest   1.0.0.0   AppBackgroundTask     {Disable-AppBackgro...
Manifest   2.0.0.0   AppLocker             {Get-AppLockerFileI...
Manifest   1.0.0.0   AppvClient            {Add-AppvClientConn...
...
```

Comme indiqué précédemment, il n'est pas nécessaire d'importer le module pour utiliser les fonctions que l'on vient de créer :

```
PS > Get-PSFolderSize -Path .
10682,433052063
```

4.1 Conversion d'un script en module

La conversion d'un script en module PowerShell est loin d'être compliquée. Il suffit de modifier l'extension du fichier. Veillez cependant à bien inclure le code dans des fonctions PowerShell. Nous verrons plus loin qu'il est possible d'exporter seulement quelques éléments sur l'ensemble du code.

Exemple de renommage

```
PS > Rename-Item -Path MyFunction.ps1 -NewName MyFunction.psm1
```

Pour la suite, nous reprenons les indications de la partie précédente. En fonction de la politique d'importation des modules, il faudra importer le nouveau module manuellement ou non avec l'aide de l'applet de commande `Import-Module`.

```
PS > Import-Module ./MyFunction.psm1
```

4.2 Convention de nommage

`Verbe-Nom` est la convention de nommage des fonctions préconisée par Microsoft.

Dans le cas où le nom de la fonction ne respecte pas cette convention, un message d'avertissement apparaît lors du chargement du module.

Pour l'exemple, on va franciser le nom de la commande `Get-PSFolderSize` : `Prendre-PSTailleDossier`.

Importation du module

```
PS > import-module MyModuleTools -Verbose
COMMENTAIRES : Importation de la fonction « Get-PSSerialNumber ».
AVERTISSEMENT : Les noms de certaines commandes importées du
module « MyModuleTools » contiennent des verbes non approuvés qui
peuvent les rendre moins détectables. Pour trouver les commandes
comportant des verbes non approuvés, re-exécutez la commande
Import-Module avec le paramètre -Verbose.
 Pour obtenir la liste des verbes approuvés, tapez Get-Verb.
COMMENTAIRES : La commande « Prendre-PSTailleDossier » dans le
module « MyModuleTools » a été importée, mais étant donné que son nom
ne contient pas de verbe approuvé, il se peut qu'elle soit difficile
à trouver. Pour obtenir la liste des verbes approuvés, tapez Get-Verb.
COMMENTAIRES : Importation de la fonction « Prendre-PSTailleDossie ».
```