Introduction à Plaster

Par Laurent Dardenne, le 14/03/2017.

Conçu avec Powershell v5.1 Windows 7 64 bits.

Plaster version 1.0.1

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

Chapitres

1 Objectif

Dans l'aide en ligne du Plaster il est dit que : « *Plaster is a scaffolding engine for PowerShell*.»

Le terme anglais de *scaffolding* traduit en français par échafaudage porte un peu à confusion il me semble, car même si c'est au pied du mur que l'on voit le maçon, ici on ne démonte rien une fois le chantier terminé.

Plaster est un module Powershell d'aide à la création de l'ossature d'un projet, ses fondations.

Lors de l'écriture de mes premiers scripts Plaster je me suis souvent demandé si j'avais affaire à un outil de setup, de copie de fichier ou si une tâche Psake pourrait suffire.

Il y a bien la notion de répétition de tâche lié à ce terme d'échafaude, l'échafaudage en lui-même n'est que le script Plaster et pas le résultat de son exécution.

On peut considérer l'usage de Plaster similaire à la création d'un nouveau projet dans un EDI, par exemple Visual Studio crée le minimum nécessaire à la structuration d'un nouveau projet.

Lors de sa création l'assistant nous demande de saisir certaines informations communes au type de projet, le nom de la dll ou de l'exécutable, le nom de son répertoire, etc. Il peut également nous demander confirmation pour certaines options.

Plaster est une aide à la construction de projets, par exemple un script ou un module.

Dès le début l'équipe du projet prévoyait d'intégrer Plaster à VSCode et ainsi que la publication de template sur la galerie Powershell. Comme nous le verrons, cette fonctionnalité a eu une influence déterminante sur des choix d'implémentation.

2 Installation

Le module étant présent sur Powershell Gallery son installation est simplifiée :

```
Install-Module Plaster
```

Pour exécuter la démo du projet on doit récupérer le répertoire d'installation du module :

```
$M=Import-module Plaster -PassThru
cd $M.ModuleBase
cd Templates\NewPowerShellManifestModule
```

Celui-ci contient deux répertoires et deux fichiers :

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        06/02/2017     13:27                editor
d-----        06/02/2017     13:27                test
-a----        16/12/2016     11:33            323 Module.psm1
-a----        16/12/2016     11:33           3700 plasterManifest.xml
```

3 Exécution d'un manifeste dans la console

Créons un répertoire de test :

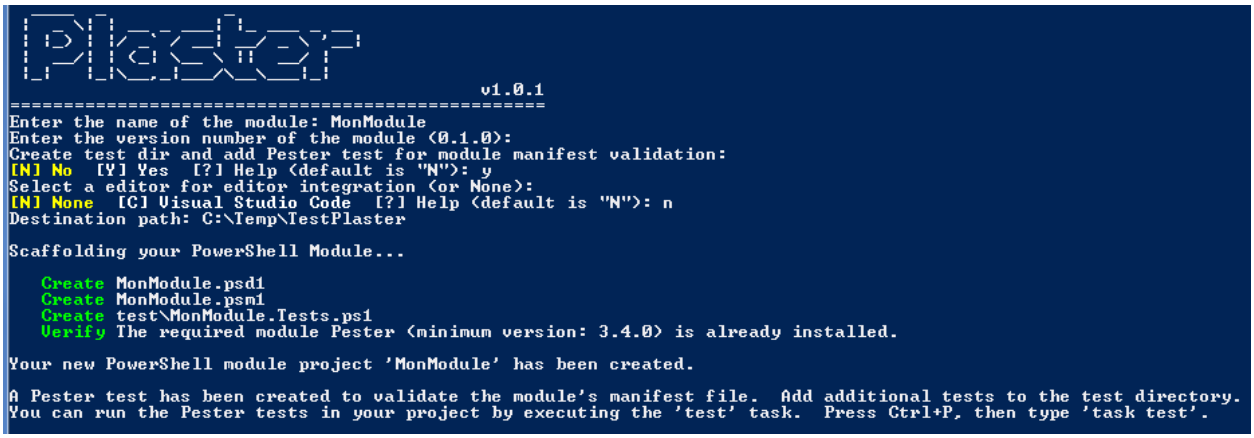
```
$TestPath='C:\Temp\TestPlaster'
Md $TestPath
```

Puis exécutons « l'assistant » avec le cmdlet **Invoke-Plaster** :

```
Invoke-Plaster -TemplatePath . -DestinationPath $TestPath
```

Il nous est demandé de saisir le nom du module, son numéro de version (par défaut 0.1.0), puis d'intégrer ou non un jeu de test ainsi que les fichiers de configuration de VSCode.

Le résultat à l'écran :



Une fois le traitement terminé, le répertoire de destination contient un répertoire de test, un module et son manifeste :

```
    Directory: C:\Temp\TestPlaster


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----       06/02/2017     13:43                test
-a----       06/02/2017     13:43           3876 MonModule.psd1
-a----       16/12/2016     11:33            323 MonModule.psm1


    Directory: C:\Temp\TestPlaster\test
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----       06/02/2017     13:43            267 MonModule.Tests.ps1
```

C'est tout, vous pouvez maintenant développer votre module.

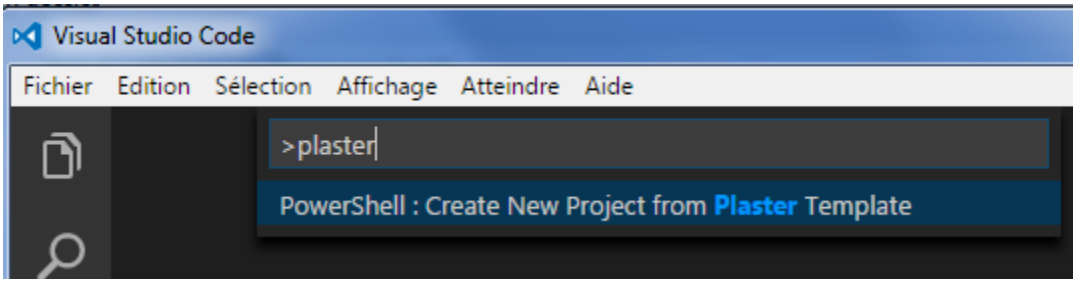
Vous pourriez vous dire « tout ça pour ça » ? Effectivement on réalise déjà ce type de construction avec des scripts Powershell.

Plaster permet d'uniformiser ces opérations récurrentes comme l'est Pester pour les tests.

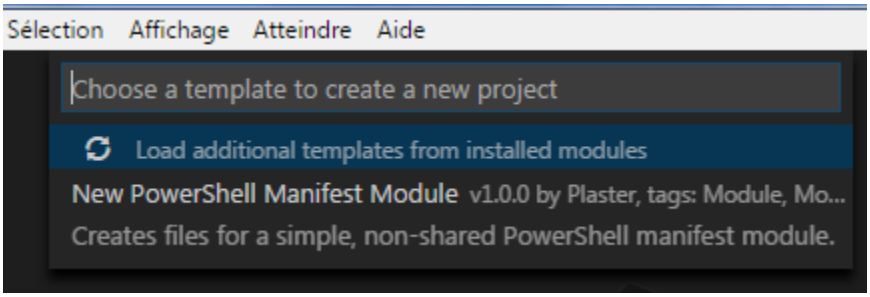
Plaster est constitué d'un moteur (le module) qui consomme des directives (le manifeste XML).

3.1 Exécution dans Vscode

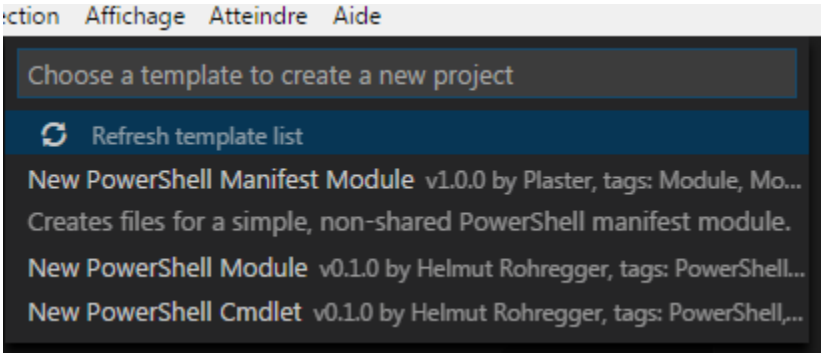
L'accès au menu se fait par *Ctrl+Shift+P* où l'on saisit *Plaster* :



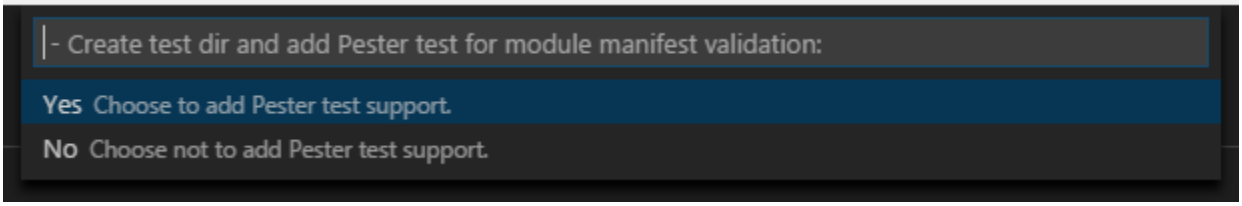
Une fois le menu sélectionné, on peut choisir un template par défaut :



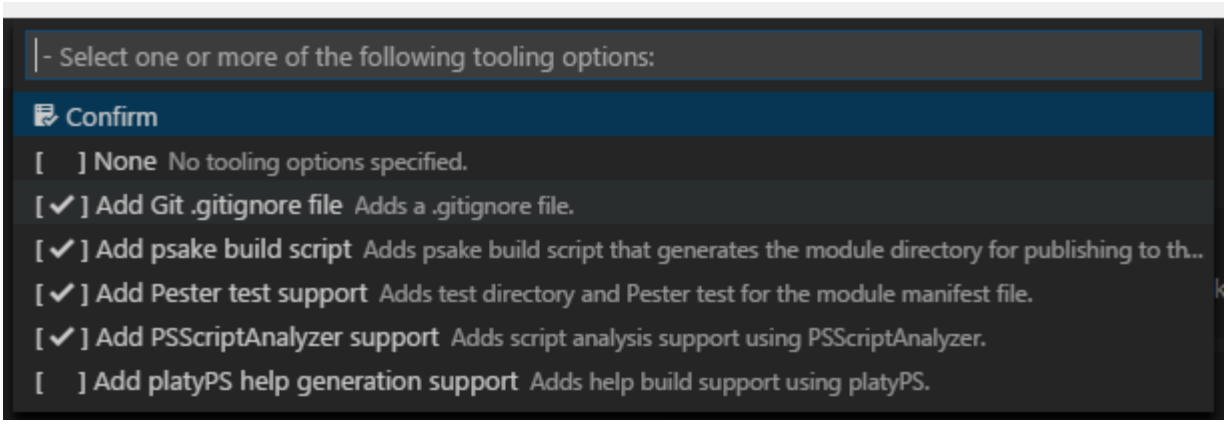
Ou charger des templates additionnels publiés sur la galerie Powershell :



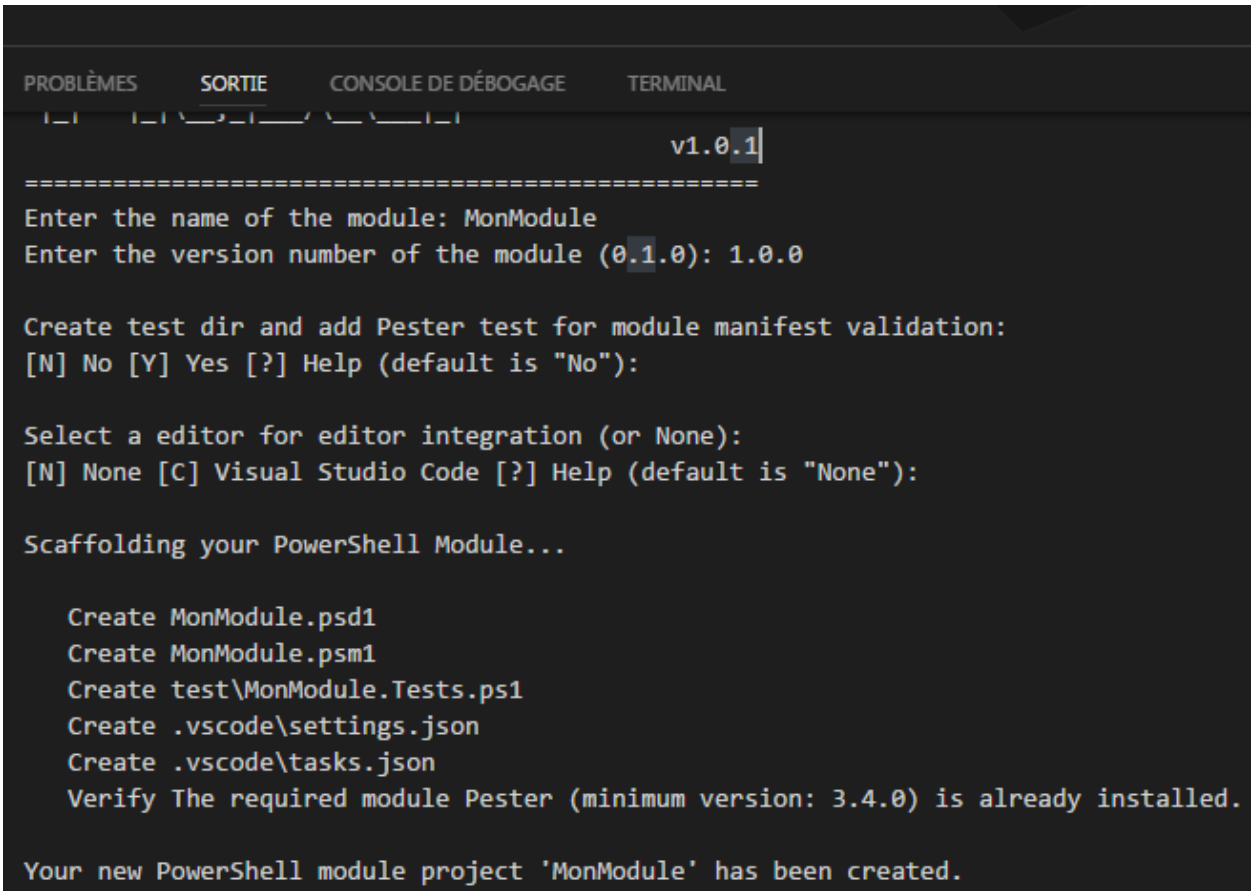
Un choix *Oui/Non* est affiché ainsi :



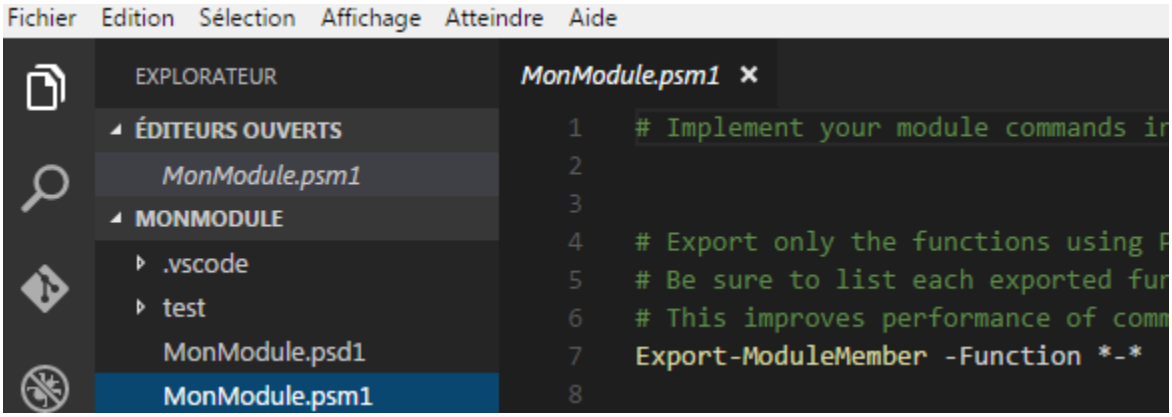
Un choix multiple de cette manière :



Le résultat de l'exécution du template est également affiché au fur et à mesure dans l'onglet *'Sortie'* de VsCode :



Une fois le traitement terminé, VsCode exécute une nouvelle instance avec le dossier du projet :



4 Contenu d'un manifeste

Les instructions de construction du projet sont regroupées dans un fichier ***plasterManifest.xml*** placé dans un répertoire dédié (*Templates\NewPowerShellManifestModule*), ce répertoire contient également les fichiers à recopier et/ou à recopier après modification.

Chaque manifeste d'échafaudage doit porter le nom ***plasterManifest.xml***.

Créons un manifeste avec le cmdlet **New-PlasterManifest** :

```
$TestPath='C:\Temp\TestManifest'
Md $TestPath
cd $TestPath
```

Attention New-PlasterManifest écrase le fichier *plasterManifest.xml* existant sans demander confirmation.

```
New-PlasterManifest -TemplateName Test
```

Celui-ci crée le fichier xml suivant :

```
Type .\plasterManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<plasterManifest
  schemaVersion="1.0"
xmlns="http://www.microsoft.com/schemas/PowerShell/Plaster/v1">
  <metadata>
    <name>Test</name>
    <id>576ed1c8-6f5a-4b5c-b0c4-84a6c0a570fa</id>
    <version>1.0.0</version>
    <title>Test</title>
    <description></description>
    <author></author>
    <tags></tags>
  </metadata>
  <parameters></parameters>
  <content></content>
</plasterManifest>
```

Notez que Plaster utilise son propre schéma XML :

```
..\Modules\Plaster\1.0.1\Schema> Type PlasterManifest-v1.xsd
```

Ce qui lui permet de valider le contenu de chaque manifeste.

La balise métadata contient en quelque sorte le versionning du manifeste. Les balises <ID> et <Version> concernent le fichier manifeste et pas le module ou script à créer.

Ensuite viennent les balises <parameters> et <content>.

4.1 Versionning de schéma

Chaque version du module Plaster est liée à un numéro de version du schéma XML et une version de schéma XML peut être commune à plusieurs versions du module Plaster.

On peut donc télécharger un manifeste qui nécessite une version supérieure du module Plaster installé sur la machine, dans ce cas l'exception suivante est déclenchée :

```
Plaster\Test-PlasterManifest : The template's manifest schema version
(1.1) in file
'C:\temp\TestManifest\plasterManifest.xml' requires a newer version of
Plaster. Update the Plaster module and try again.
```

La version 1.0.1 ne permet pas de retrouver cette association hormis en lisant le code source.

La prochaine version proposera un nouveau champ pour le type de template *Item* ou *Project* :

```
<plasterManifest schemaVersion="0.4"
                 templateType="Item"
                 xmlns="…">
```

Le type *Item* ajoutera un élément dans un projet existant.

Une balise *Content.File* indiquera quel fichier ouvrir dans l'éditeur une fois la génération du template terminé :

```
   <file source="PSScriptAnalyzerSettings.psd1"
         destination="$PLASTER_PARAM_FileName"
         openInEditor="true" />
```

4.2 Localisation

La localisation d'un manifeste est possible et s'appuie sur le mécanisme de Powershell, attention toutefois à l'encodage du fichier.

Le nom de fichier doit être plasterManifest_*Culture*.xml, par exemple 'plasterManifest_Fr.xml' situé dans le répertoire fr-FR.

Le code actuel ne gère pas l'accès à une clé de hashtable :

```
  <parameter name='ProjectName'
             type='text'
             prompt='$($ManifestLocalizedData.P_ProjectName)'/>
```

Cette restriction implique de gérer un fichier par culture :-/

4.3 Balise parameters

Elle contient les possibles options proposées à l'utilisateur/trice :

```
<parameters>
 <parameter name='ModuleName'
            type='text'
            prompt='Enter the name of the module'/>
```

Puisque l'on déclare des paramètres, le nom du paramètre *ModuleName* déclaré ci-dessus, peut être réutilisé dans la balise <*Content*> en tant que variable Powershell :

```
<file source='Module.psm1'
      destination='${PLASTER_PARAM_ModuleName}.psm1'/>
```

Ce nom de variable est constitué du préfixe ***PLASTER_PARAM_*** puis du nom du paramètre ***ModuleName***. Lors de l'exécution du manifeste Plaster injecte dans le contexte du module une variable nommée **$PLASTER_PARAM_ModuleName**.

Le nom de ce paramètre peut également être utilisé sur la ligne de commande lors d'exécutions sans interaction :

```
Invoke-Plaster -TemplatePath . -DestinationPath $TestPath -ModuleName
MonModule
```

Ceci est possible car Plaster lit le fichier manifeste puis crée un paramètre dynamique pour chaque balise <*parameter*>.

La majorité des balises propose un attribut *condition* qui permet d'exécuter une opération selon le résultat de l'évaluation de la condition.

Dans l'exemple suivant on réutilise le contenu du paramètre *Editor* :

```
 <templateFile condition="($PLASTER_PARAM_Editor -eq 'VSCode')"
               source=".vscode\settings.json"
               destination="" />
```

Le fichier indiqué sera copié si la condition est vraie, à savoir le choix de l'éditeur VSCode.

4.3.1 Type de paramètre

Un paramètre de manifeste peut être d'un des types suivant :

text, *user-fullname*, *user-email*, *choice* ou *multichoice*

Le type *text* contient une chaîne de caractères.

Les types *user-fullname* et *user-email* précise d'extraire le contenu des champs correspondant dans le fichier *.gitconfig* de l'utilisateur s'il en existe un.

Le type *choice* référence une seule valeur dans une liste de choix :

```
  <choice label='&amp;GitHub'
          help="Creates a GitHub project for you PowerShell module"
          value="GitHub"/>
```

Pour placer en surbrillance la lettre de sélection d'un menu préfixez la de *&*

Le type *multichoice* référence une ou plusieurs valeurs dans une liste de choix. Dans ce cas chaque choix nécessitera une saisie.

Un paramètre peut avoir une valeur par défaut :

```
<parameter name='Version'
           type='text'
           prompt='Enter the version number of the module'
           default='0.1.0'/>
```

Il est également possible de spécifier sa mise en cache à l'aide de l'attribut *store.*

4.3.2 Mise en cache

En cas d'exécution multiple d'un même manifeste, les paramètres peuvent être enregistrés et ainsi éviter de ressaisir des informations communes à plusieurs projets, par exemple le nom de l'auteur/organisation ou la licence.

Une fois enregistrées ces valeurs sont proposées comme valeur par défaut lors de la prochaine exécution du manifeste concerné.

La valeur de cet attribut peut être "text" ou "encrypted".

Les données stockées sont enregistrées dans le dossier du profil utilisateur, le nom du fichier est construit avec le nom du manifeste, son numéro de version, son numéro d'ID et l'extension *.clixml*.

L'emplacement du fichier cache diffère selon le système d'exploitation :

Windows

- `$env:LOCALAPPDATA\Plaster`

Linux

- `$XDG_DATA_HOME/plaster` (si `$XDG_DATA_HOME` a une valeur)
- `$Home/.local/share/plaster` (si `$XDG_DATA_HOME` n'a pas de valeur)

Autres

- `$Home/.plaster`

L'usage du paramètre *–Debug* avec **Invoke-Plaster** affichera le chemin du fichier de cache du manifeste :

```
DEBUG: Loading default value store from
'C:\Users\Laurent\AppData\Local\Plaster\NewModule-1.0.0-dcd95744-8abc-
4ecb-a439-bf2cd37821bb.clixml'.
```

Le guid du nom de fichier est celui indiqué dans le manifeste :

```
 <metadata>
    <version>1.0.0</version>
    <id>dcd95744-8abc-4ecb-a439-bf2cd37821bb</id>
```

4.4 Balise content

Cette balise contient les instructions de construction du projet, le paramètre *–DestinationPath* contient le répertoire d'installation.

Par exemple la balise suivante ;

```
<Content>
  <file source='Module.psm1'
        destination='${PLASTER_PARAM_ModuleName}.psm1'/>
```

référence un des fichiers présent dans le répertoire de démo *NewPowerShellManifestModule :*

```
-a----        16/12/2016     11:33            323 Module.psm1
```

C'est ce fichier qui est recopié dans le répertoire cible de notre nouveau projet et portera le nom que vous aurez saisi. On retrouve le nom de paramètre *'ModuleName'* déclaré précédemment, mais ici en tant que variable Powershell.

Les actions possibles avec la version 1.0.1 de Plaster :

file	Copie d'un ou plusieurs fichiers.
templateFile	Copie et transformation de fichiers de modèle. Utilise la substitution de variable. Vous pouvez utiliser la convention de nommage suivante : Module.**T**.ps1
Message	Affiche un message à l'écran.
Modify	Modifie un fichier existant (basée regex).
newModuleManifest	Crée un fichier de manifeste de module, utilise en interne le cmdlet *New-ModuleManifest*.
requireModule	Vérifie si le module spécifié est installé. Sinon, l'utilisateur est informé de la nécessité d'installer le module. **Ce n'est qu'un contrôle d'existence de module, aucune installation n'est ici possible.**

Pour le détail de ces balises consultez le fichier PlasterManifest-v1.xsd, chaque option y est documentée.

Note :

Les destinations indiquées dans ces balises *content* ne peuvent être que des chemins relatifs au répertoire de destination précisé dans le paramètre **-DestinationPath**.

Modifier ceci :

```
<file source='Module.psm1'
      destination='${PLASTER_PARAM_ModuleName}.psm1'/>
```

en :

```
<file source='Module.psm1'
      destination='C:\${PLASTER_PARAM_ModuleName}.psm1'/>
```

Déclenchera l'exception suivante :

```
The path 'C:\MonModule.psm1' specified in the file directive in the
template manifest cannot be an absolute path.
Change the path to a relative path.
```

Il n'est donc pas possible d'effectuer ces types de traitements en dehors du répertoire $**DestinationPath**.

Dans ce cas utilisez un script exécutant *Invoke-Plaster* puis copier les fichiers additionnels dans le ou les autres répertoires.

Pour ***requireModule*** on suppose que les outils référencés dans les fichiers du template existent sur le poste où il sera exécuté. Plaster n'est pas un outil de configuration de poste.

5 Valider un manifeste

Avant d'exécuter les instructions du manifeste *Invoke-Plaster* appelle **Test-PlasterManifest** afin de valider le fichier xml.

Pour ce test supprimons dans le fichier de manifeste la balise Name :

```
   <name>Test</name>
```

Puis exécutons **Test-PlasterManifest** dans le répertoire courant :

```
$TestPath='C:\Temp\TestManifest'
cd $TestPath
Test-PlasterManifest
```

Le fichier xml étant invalide une exception est déclenchée :

```
Test-PlasterManifest : The Plaster manifest
'C:\Temp\TestManifest\plasterManifest.xml' is not valid.
Specify –Verbose to see the specific schema errors.
```

L'usage du paramètre –*Verbose* affiche le détail de l'erreur :

```
Test-PlasterManifest -Verbose
VERBOSE: Plaster manifest schema error in file
'C:\Temp\TestManifest\plasterManifest.xml'. Error: Le contenu de
l'élément 'metadata' dans l'espace de noms
'http://www.microsoft.com/schemas/PowerShell/Plaster/v1' est incomplet.
Liste d'éléments possibles attendue : 'name' dans l'espace de noms
'http://www.microsoft.com/schemas/PowerShell/Plaster/v1'.
Test-PlasterManifest : The Plaster manifest
'C:\Temp\TestManifest\plasterManifest.xml' is not valid.
```

Notez que le cmdlet **Invoke-Plaster** utilise également –*Verbose* pour détailler les erreurs de validation XML car en interne il appelle la fonction **Test-PlasterManifest**.

6 Plaster et les variables

Outre les variable associées à chaque paramètre (${*PLASTER_PARAM_Name*}), Plaster déclare par défaut les variables suivantes :

- *PLASTER_TemplatePath* – Chemin absolu du répertoire de template.
- *PLASTER_DestinationPath* - Chemin absolu du répertoire de destination.
- *PLASTER_DestinationName* – Le nom du répertoire de destination.
- *PLASTER_FileContent* – Le contenu d'un fichier modifié par une directive `<modify>`.
- *PLASTER_DirSepChar* – Le caractère séparateur de chemin dépendant de la plateforme.
- *PLASTER_HostName* – Le nom du host PowerShell, par exemple $Host.Name
- *PLASTER_Version* – la version du module de Plaster exécutant le template.
- *PLASTER_Guid1* – Valeur de type GUID générée aléatoirement. 5 Variables de ce type sont déclarées (Guid1, Guid2, Guid3, Guid4 et Guid5).
- *PLASTER_Date* – Date courante dans le format court, par exemple 10/31/2016 .
- *PLASTER_Time* – Date et heure courante dans le format court, par exemple 5:11 PM .
- *PLASTER_Year* – L'année en cours sur quatre chiffres.

6.1 Substitution de variable

La construction à partir d'un fichier template utilise la substitution de variable.

Les variables déclarées dans le fichier template, telle que $*PSScriptRoot*, ne sont pas substituées seule celles inclues dans un pattern spécifique le sont.

Le pattern doit respecter cette forme :

<%=${*Nom_De_Variable*}%>

Le fichier ciblé par cette substitution doit être déclaré dans une balise *templateFile* :

```
 <templateFile condition="$PLASTER_PARAM_Options -contains 'Pester'"
               source='test\Module.T.ps1'
```

Pour les lignes suivantes Plaster substitue uniquement le pattern par le contenu du paramètre *ModuleName* :

```
 # <%=${PLASTER_PARAM_ModuleName}%>.psm1
. $PSScriptRoot\Shared.ps1
```

Dans ce cas la variable *$PSScriptRoot* n'est pas substituée :

```
# MonModule.psm1
. $PSScriptRoot\Shared.ps1
```

On peut également insérer du code dans un template, il doit être délimité par **<% et %> :**

```
# The Get-<%=$PLASTER_PARAM_TargetResourceName%> function fetches the
# status of the <%=$PLASTER_PARAM_TargetResourceName%> resource instance
# specified in the parameters for the target machine.

function Get-<%=$PLASTER_PARAM_TargetResourceName%> {
    [OutputType([Hashtable])]
    param (
<%
    if ($PLASTER_PARAM_Ensure -eq 'Yes') {
"        # Ensure the presence/absence of the resource."
"        [ValidateSet('Present','Absent')]"
"        [string]"
"        `$Ensure = 'Present'"
    }
%>
```

Si le paramètre *Ensure* à la valeur ' yes', on insère dans le fichier généré les lignes spécifiées.

Notez que dans ce cas les variables doivent être échappées pour éviter leur substitution.

6.2 Transformation de texte

Le manifeste peut déclarer des modifications de texte de ses fichiers :

```
<file source='Templates\_gitignore'
      destination='.gitignore'/>
   <modify path='.gitIgnore' encoding='UTF8'>
    <replace>
     <!-- Append  -->
     <original>(?s)^(.*)$</original>
      <substitute expand='true'>
       ``$0${PLASTER_PARAM_ProjectName}.tdl
      </substitute>
      </replace>
   </modify>
```

Ces transformations sont basées sur des expressions régulières, dans cet exemple on ajoute une exclusion dans le fichier *.gitignore*.

Le moteur de transformation étant interne à Plaster il n'est pas possible de le réutiliser pour d'autres traitements.

7 Runspace contraint

Le code du module Plaster exécute les instructions du manifeste dans un environnement contraint. Etant donné que l'assistant de projet peut provenir de la galerie Powershell, seules les commandes et instructions Powershell suivantes sont autorisées dans la version 1.0.1 :

- *Get-Content*
- *Get-Date*
- *Get-ChildItem*
- *Get-Item*
- *Get-ItemProperty*
- *Get-Module*
- *Get-Variable*
- *Test-Path*

L'usage d'un cmdlet qui n'est pas listé provoquera une erreur :

```
Invoke-Plaster : Exception calling "Replace" with "4" argument(s):
"PowerShell expression failed execution. Location:

templateFile 'Test.T.ps1'. Error: The expression '
  if ($true)
  {
    Set-Variable Nom 'Value'
  }
' generated error output - The term 'Set-Variable' is not recognized as
the name of a cmdlet, function, script file, or operable program.

Check the spelling of the name, or if a path was included, verify that the
path is correct and try again."
```

Ici le cmdlet *Set-Variable* n'est pas autorisé.

Il n'est donc pas possible d'utiliser des modules externes de génération de code dans un template Plaster.

Les variables $PLASTER_PARAM_ sont créées dans le scope de l'appelant.

Les variables Powershell existantes dans le scope courant, hormis les variables automatiques, ne peuvent être utilisées lors du traitement d'un manifeste puisqu'elles ne sont pas recopiées dans le runspace contraint.

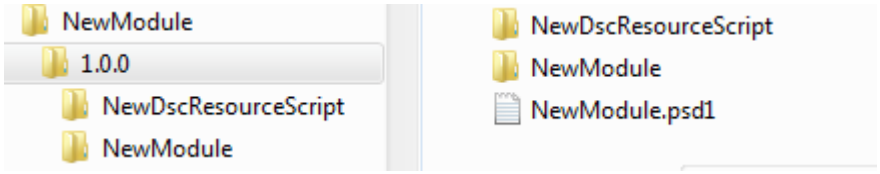
Le runspace contraint autorisant l'accès aux providers *Environment* et *FileSystem*, il est par contre possible d'utiliser les variables d'environnement déclarées avant l'appel à ***Invoke-Plaster***.

8 Publication de template

La publication de templates sur Powershell gallery nécessite un module (même vide) et un manifeste de module, celui-ci pourra référencer un ou plusieurs templates. Pour une installation locale VSCode n'utilise que les fichiers templates et le fichier manifeste.

Le manifeste de module doit contenir une section nommée *Extension* imbriquée dans la section ***PSData***. Sinon la fonction Get-PlasterTemplate ne retrouvera pas les templates.

Par exemple pour ces deux templates :



Le fichier manifeste contiendra :

```
PrivateData = @{
    PSData = @{
        Tags = @( 'plaster', 'template', 'new', 'module', 'DSC' )
        Extensions = @(
            @{
                Module = "Plaster"
                Details = @{
                    TemplatePaths = @(
                        'NewModule',
                        'NewDscResourceScript'
                    )                }
                MinimumVersion = "0.3.0"
                MaximumVersion = "1.0.1"
            }
        )
    } # End of PSData hashtable
} # End of PrivateData hashtable
```

La clé *Module* contient toujours '**Plaster**', c'est une extension gérée par le module Plaster.

Les clés *MinimumVersion* et *MaximumVersion* cible les versions minimum et maximum requises pour ces templates.

La clé *Details* ne contient pour le moment que la clé *TemplatePaths*. Elle référence des noms de chemin relatif au répertoire hébergeant le manifeste. Ici les 2 répertoires indiqués contiennent chacun un fichier plasterManifest.xml.