Introduction à Psake.

Par Laurent Dardenne, le 03/04/2014.

Niveau

Débutant	Avancé	Confirmé

Conçu avec Powershell v2 & v3 sous Windows Seven 64 bits.

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

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

Chapitres

1 QU'EST-CE QUE PSAKE ? ... 3

2 POURQUOI L'UTILISER ? ... 3

3 PRINCIPE ... 3

4 COMMENT DECLARER UNE TACHE ? ... 4

4.1 LA VARIABLE PSAKE ... 6

5 UNE TACHE ET UNE SOUS-TACHE ... 6

5.1 DEPENDANCES DE TACHES ... 7
5.2 DEPENDANCE CIRCULAIRE ... 8
5.3 TACHE INCONNUE ... 8
5.4 DUPLICATION DE NOM DE TACHE ... 8

6 PARAMETRAGE DES TACHES ... 9

6.1 NOM DE TACHE ... 9
6.2 CONDITIONS ... 9
6.3 INITIALISATION ET FINALISATION DE TACHES ... 10
6.3.1 Préaction et post action d'une tâche ... 10
6.4 GESTION D'ERREUR ... 10
6.5 EXECUTION D'UNE SEULE TACHE D'UN SCRIPT ... 11
6.6 INCLUSION ... 11
6.7 PARTAGE DE VARIABLES ... 12
6.7.1 Propriétés d'une tâche ... 12
6.7.2 Paramétrer une propriété ... 12
6.8 CONSTRUCTION IMBRIQUEE ... 13
6.9 ASSERTION ... 13

7 LA VARIABLE $PSAKE ... 13

7.1 DETERMINER LA PRESENCE D'UNE TACHE DEFAULT ... 13
7.2 EXECUTION D'UN SCRIPT PAR DEFAUT ... 14

8 EXECUTION DE PROGRAMME EXTERNE ... 14

9 REGLES SYNTAXIQUES / BNF ... 14

10 CONCLUSION ... 15

1 Qu'est-ce que Psake ?

Psake est un **outil** d'automatisation de construction écrit en Powershell.

Son objectif est de simplifier la réalisation d'un traitement à l'aide de tâches.

Il propose des mots clés basés sur les possibilités du langage Powershell (un DSL, Domain Specific Languages). Cette approche permet de s'affranchir de la connaissance d'une syntaxe XML telle que celle de MSBuild (outil de construction de programme compilé).

Vous pouvez télécharger ce module ici : https://github.com/psake/psake

Sa documentation : https://github.com/psake/psake/wiki

2 Pourquoi l'utiliser ?

Lors de la conception d'un script ou d'un module on est amené à reproduire les mêmes actions, que celles-ci soient liées à la création de ressources nécessaires au fonctionnement du dit script ou à sa mise en place dans un environnement de tests.

Si le temps de conception est très court on peut se contenter d'actions manuelles, en revanche si elles se répètent sur plusieurs jours on se peut se poser la question de l'automatisation de ces actions récurrentes.

Si vous travaillez dans la production, vous pouvez également l'utiliser pour exécuter certaines opérations de validation avant la livraison d'un script corrigé dans l'urgence, ici il s'agit de déléguer à un outil une rigueur qui parfois nous fait défaut.

3 Principe

Psake étant un module il nous faut avant toute chose le charger :

```
Import-Module Psake
```

Son principe se base sur la construction de script contenant des déclarations de tâches, ce script ou ces tâches étant exécutées par une fonction du module Psake, à savoir ;

```
Invoke-Psake 'MesTaches.ps1'
```

L'élément primaire de ce script est la tâche :

```
Task Init {
  Write-Host "Exécute le code de la tâche Init"
}
```

Le mot clé nécessaire à la déclaration de la tâche est **Task** suivit de son nom ***Init*** et du code powershell à exécuter lorsque cette tâche sera appelée. Le code est déclaré dans un scriptblock.

Bien que le script 'MesTache.ps1' soit un script Powershell, son exécution ne peut se faire directement dans la console. La fonction *Invoke-Psake* construit dans un premier temps la liste des tâches à traiter, puis les exécute.

Cette approche permet également, comme nous le verrons plus avant, la déclaration de tâches dépendantes :

```
Task Clean -Depends Init {
  Write-Host "Exécute le code de la tâche Clean"
}
```

Ici, on déclare que la tâche nommée *Clean* dépend de la tâche *Init*, ce qui signifie que cette dernière sera exécutée avant la tâche nommée *Clean* :

```
Exécute le code de la tâche Init
Exécute le code de la tâche Clean
…
```

4 Comment déclarer une tâche ?

Voyons déjà le minimum à déclarer dans un script Psake ainsi que les cas d'erreur que l'on peut rencontrer lors de cette étape :

```
$Target=$env:Temp
$Encoding='UTF8'

'#Aucune informations, déclenche une exception'|
 Set-content "$Target\Script0.ps1" -encoding $Encoding
Invoke-PSake "$Target\Script0.ps1"
```

```
psake version 4.2.0
Copyright (c) 2010 James Kovacs
06/03/2014 09:38:11: An Error Occurred:
throw : 'default' task required.
```

Ce script ne contient pas de déclaration de tâche. Précisons le terme **Task** :

```
@'
 Task
'@ |Set-content "$Target\Script1.ps1" -encoding $Encoding
Invoke-PSake "$Target\Script1.ps1"
```

```
Applet de commande Task à la position 1 du pipeline de la commande
Fournissez des valeurs pour les paramètres suivants:
Name: Init
06/03/2014 09:38:11: An Error Occurred:
throw : 'default' task required.
```

Celui-ci déclenche une demande de saisie d'une valeur pour le paramètre *Name*.

```
@'
  Task Toto {Write-Host "Exécute du code"}
'@|Set-content "$Target\Script2.ps1" -encoding $Encoding
Invoke-PSake "$Target\Script2.ps1"
```

```
06/03/2014 09:38:11: An Error Occurred:
throw : 'default' task required.
```

Les messages d'erreur générés jusqu'à présent, nous indiquent que l'existence de la tâche ***default*** est requise. Déclarons là :

```
@'
  Task default {Write-Host "Exécute du code"}
'@|
Set-content "$Target\Script3.ps1" -encoding $Encoding
Invoke-PSake "$Target\Script3.ps1"
```

06/03/2014 09:38:11: An Error Occurred:
throw : Assert: 'default' task cannot specify an action.

Un contrôle interne de Psake nous indique cette fois que la tâche ***default*** ne peut spécifier une action. Essayons de lui associer non pas du code, mais une autre tâche :

```
@'
  Task default UneAutreTache
  Task UneAutreTache {Write-Host "UneAutreTache : exécute du code"}
'@|Set-content "$Target\Script4.ps1" -encoding $Encoding
Invoke-PSake "$Target\Script4.ps1"
```

06/03/2014 09:38:11: An Error Occurred:
Task : Impossible de traiter la transformation d'argument sur le paramètre « action ». Impossible de convertir la valeur « UneAutreTache » du type « System.String » en type « System.Management.Automation.ScriptBlock ».

Tout ceci pour vous indiquer que la construction de la tâche ***default*** est particulière, car elle construit la racine des tâches, elle doit préciser le nom d'une autre tâche à l'aide du paramètre **–Depend** et ne pas contenir de déclaration de code :

```
@'
  Task default -Depend UneAutreTache
  Task UneAutreTache {Write-Host "UneAutreTache : exécute du code"}
'@|Set-content "$Target\Script5.ps1" -encoding $Encoding
Invoke-PSake "$Target\Script5.ps1"
```

Le résultat étant :

```
psake version 4.3.1
Copyright (c) 2010 James Kovacs

Executing UneAutreTache
UneAutreTache : exécute du code

Build Succeeded!

----------------------------------------------------------------------
Build Time Report
----------------------------------------------------------------------
Name          Duration
----          --------
UneAutreTache 00:00:00.0060422
Total:        00:00:00.0202629
[STA] C:\temp>
```

Cet affichage ne fait pas mention de la tâche ***default***, mais nous indique que la tâche dépendante a bien été exécutée. Le compte rendu final nous précise le temps de traitement de chaque tâche.

Notez que la notion de tâche de PSake n'est qu'une notion logique, elle n'est pas liée à la notion de thread ou de PSJob. L'exécution d'un script Psake reste séquentielle.

4.1 La variable Psake

L'exécution du script précédent renseigne une variable automatique nommée *$psake* de type hashtable (ici sous Powershell v3) :

```
[STA] C:\temp> $psake

Name                           Value
----                           -----
version                        4.3.1
build_success                  True
build_script_dir               C:\Users\Laurent\AppData\Local\Temp
run_by_psake_build_tester      False
build_script_file              C:\Users\Laurent\AppData\Local\Temp\Script5.ps1
config_default                 @{framework=4.0; verboseError=False; coloredOut
context                        {}


[STA] C:\temp> $psake.config_default


framework      : 4.0
verboseError   : False
coloredOutput  : True
buildFileName  : default.ps1
moduleScope    :
taskNameFormat : Executing {0}
modules        :
```

5 Une tâche et une sous-tâche

Vous pouvez nommer vos tâches comme vous le voulez.

Le script suivant déclare une tâche nommée *'Build'* dépendante d'une tâche nommée *'Init'* :

```
 #Script6.ps1 du répertoire de démos
Task default -Depends Build


Task Build -Depends Init { Write-Host "Tâche Build : On exécute la
construction "}


Task Init {
   Write-Host "Tâche Init: dépendance imbriquée. On initialise la
construction."
}
```

Pour cet exemple, l'ordre d'exécution se lit facilement, la tâche par défaut est la tâche **Build**, qui dépend de la tâche **Init** et celle-ci n'a pas de dépendances, le parcours des dépendances s'arrête donc ici.

La tâche **Init** est exécutée en premier puis c'est au tour de la tâche **Build :**

```
Executing Init
Tâche Init: dépendance imbriquée. On initialise la construction.
Executing Build
Tâche Build : On exécute la construction

Build Succeeded!

----------------------------------------------------------------
Build Time Report
----------------------------------------------------------------
Name   Duration
----   --------
Init   00:00:00.0032877
Build  00:00:00.0058706
Total: 00:00:00.0290520
```

Notez que l'ordre de déclaration des tâches importe peu, elle facilite juste la relecture.

5.1 Dépendances de tâches

Voyons maintenant le script Script8.ps1 présent dans le répertoire de démos *:*

```
Task default -Depends Build

Task Build -Depends Compile {
   Write-Host "Tâche Build : On termine la construction."
}

Task Compile -Depends Clean, Init {
   Write-Host "Tâche Compile : On exécute la compilation"
}

Task Clean -Depends Init {
  $s="Tâche Clean : On nettoie les fichiers créés précédemment."
  Write-Host $s
}

Task Init {
   Write-Host "Tâche Init: On initialise la construction."
}
```

Il déclare une tâche nommée *'Build'* dépendante de la tâche nommée *'Compile'.*

La tâche *'Compile'* dépend des tâches '*Clean' et 'Init*', et la tâche '*Clean'* dépend de la tâche *'Init*'.

Le compte rendu reflète bien l'ordre d'exécution demandé :

```
Executing Init
Tâche Init: On initialise la construction.
Executing Clean
Tâche Clean : On nettoie les fichiers créés précédemment.
Executing Compile
Tâche Compile : On exécute la compilation
Executing Build
Tâche Build : On termine la construction.

Build Succeeded!
```

Cet ordre d'exécution peut être affiché à l'aide du paramètre *-Docs* de la fonction **Invoke-Psake** :

```
Invoke-psake Script8.ps1 -doc
```

```
Name     Description Depends On  Default
----     ----------- ----------  -------
Build                Compile        True
Clean                Init
Compile              Clean, Init
Init
```

Bien que la tâche '*Init'* soit déclarée comme dépendance des tâches *'Compile'* et '*Clean'*, celle-ci n'est exécutée qu'une seule fois, c'est une limite à connaitre. Pour la contourner il faut dupliquer le code en lui attribuant un autre nom de tâche unique.

5.2 Dépendance circulaire

Une tâche ne peut dépendre d'elle-même :

```
Task Clean -Depends Clean {
```

Cette situation est détectée lors de l'exécution :

```
throw : Assert: Circular reference found for task Clean.
```

La construction suivante est également détectée :

```
Task Build -Depends Init, Clean, Compile { 'code' }

Task Init -Depends Build { 'code' }
```

5.3 Tâche inconnue

Les tâches référencées, mais inconnue dans le script déclenche une exception lors de l'exécution :

```
throw : Assert: Task Init does not exist.
```

5.4 Duplication de nom de tâche

La tâche *default* comme tout autre nom de tâche est unique. Cette situation est détectée lors de l'exécution :

```
Assert: Task default has already been defined.
```

6 Paramétrage des tâches

Voyons maintenant l'étape suivante, à savoir le paramétrage autour de ces tâches.

6.1 Nom de tâche

Le plus simple, retrouver le nom de la tâche :

```
Task default -Depend UneAutreTache
Task UneAutreTache {Write-Host "$TaskName : exécute du code"}
```

Le mot clé *FormatTaskName* précise le format d'affichage du nom de chaque tâche :

```
FormatTaskName "-------- {0} --------"
```

6.2 Conditions

Psake permet de déclarer des conditions sur l'exécution d'une tâche, une condition est un scriptblock renvoyant un booléen.

Pour une pré condition, si la valeur de retour est ***$false***, alors la tâche n'est pas exécutée :

```
Task Init -PreCondition {"10/10/2018" -as [datetime] -gt [datetime]::now }
{ " code de la tâche" }
```

Dans ce cas le message est affiché avec la couleur Cyan et celle-ci n'est pas paramétrable :

```
Precondition was false, not executing task Init.
-------- Clean --------
Tâche Clean
--------------------------------------------------
```

Le résultat $*false* issu de l'exécution d'une condition n'est pas considéré comme une erreur.

Pour une post condition, si la valeur de retour est ***$false***, alors une exception est déclenchée :

```
Task Init -PostCondition {"10/10/2018" -as [datetime] -gt [datetime]::now
} { " code de la tâche" }
```

```
-------- Init --------
Tâche Init: On initialise la construction.
07/03/2014 15:06:34: An Error Occurred:
throw : Assert: Postcondition failed for task Init.
```

Son usage évite une gestion d'erreur élaborée dans le code de la tâche. Ce qui importe ici c'est la réussite du traitement.

6.3 Initialisation et finalisation de tâches

Les fonctions *TaskSetup* et *TaskTearDown* déclarent du code devant être appelé avant et après l'exécution du code de chaque tâche :

```
#Déclare un code exécuté à chaque début de tâche
TaskSetup {
  Write-host "`t`t`t`t *[$TaskName] Début $(get-date)" -fore Green
}

 #Déclare un code exécuté à chaque fin de tâche
TaskTearDown {
 Write-host "`t`t`t`t *[$TaskName] Fin $(get-date)" -fore DarkGreen
}
```

En cas d'erreur dans une tâche, la fonction *TaskTearDown* n'est pas exécutée.

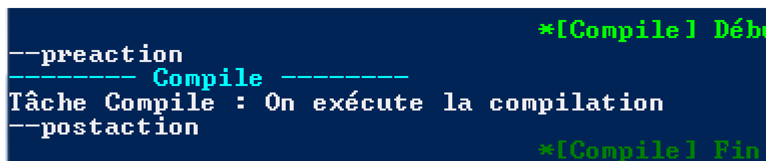
Si une précondition renvoie $false, les fonctions *TaskSetup* et *TaskTearDown* ne sont pas exécutées.

6.3.1 Préaction et post action d'une tâche

Un niveau supplémentaire peut être mis en place sur une tâche particulière.

Task propose les paramètres *–preaction* et -*postaction* :

```
Task Compile -Depends Clean,Init `
     -preaction {Write-host "--preaction" }`
     -postaction {Write-host "--postaction" }  {
       Write-Host "Tâche Compile : On exécute la compilation"
    }
```



6.4 Gestion d'erreur

L'exécution de scripts Psake peut déclencher des erreurs, toutes sont considérées comme bloquantes. Pour outrepasser cette gestion on doit configurer la tâche à l'aide du paramètre –***ContinueOnError*** :

```
Task Clean -ContinueOnError -Depends Init  {
   Write-Host "Tâche $TaskName"
   Write-Error "1 Erreur dans la tâche $Taskname"
   #Throw "2 Erreur dans la tâche $Taskname"
}
```

```
--------- Clean ---------
Tâche Clean
----------------------------------------------------------------------
Error in task Clean. 1 Erreur dans la tâche Clean
----------------------------------------------------------------------
--------- Compile ---------
Tâche Compile : On exécute la compilation
--------- Build ---------
Tâche Build : On termine la construction.

Build Succeeded!
```

Certains scripts de constructions nécessiterons de démarrer une session Powershell dédiée à chaque exécution. Par exemple, la compilation d'une dll suivie de son chargement ne peut réussir qu'une seule fois dans une même session. Le fonctionnement en *tout ou rien* est préférable.

6.5 Exécution d'une seule tâche d'un script

On peut vouloir exécuter une ou plusieurs tâches particulières contenues dans un script Psake, la fonction *Invoke-psake* propose le paramètre **–TaskList :**

```
Invoke-psake .\Script6-1.ps1 -taskList Init
```

```
Executing Init
Tâche Init: dépendance imbriquée. On initialise la construction.

Build Succeeded!
```

Ce mode d'exécution peut donc référencer un script Psake ne contenant pas de déclaration de la tâche nommée *default*.

6.6 Inclusion

La réutilisation de tâches est possible via le mot clé '*Include*' suivie du nom d'un script :

```
Include "$PsIonicTools\Common.ps1"
```

Ce script peut être un script Psake ou script powershell.

L'inclusion d'un ou plusieurs scripts Psake permet de découper les tâches et de les réassembler à volonté.

L'inclusion d'un ou plusieurs scripts powershell permet de publier des fonctions accessibles à toutes les tâches. Psake exécutant chaque tâche dans un contexte qui lui est propre, *Include* évite la duplication de code powershell :

```
 #Charge cette fonction dans la portée de PSake
include "$ProjectTools\Get-MyResources.ps1"
```

Include construit en interne une liste de scripts et chaque appel complètera cette liste.

Lors de son exécution*, Invoke-psake* injectera tous les scripts déclarées dans cette liste, dans la portée de la tâche en cours d'exécution.

Lors du démarrage d'un script Psake celui-ci est d'abord analysé puis exécuté, il est donc possible de placer des '*Include*' en fin de script.

6.7 Partage de variables

Psake exécutant chaque tâche dans un contexte qui lui est propre, on peut souhaiter partager des variables, celles-ci seront déclarées dans le code du script et pas dans le code d'une tâche.

6.7.1 Propriétés d'une tâche

Le mot clé *Properties* permet de déclarer de telles variables.

```
Properties {
   $Configuration=$Config
   $Cultures= "fr-FR","en-US"
   $ProjectName='PSProject'
}
```

Ici aussi *Properties* construit en interne une liste de déclaration de propriété et chaque appel complètera cette liste.

Lors de son exécution, *Invoke-psake* injectera toutes ces déclarations dans la portée de la tâche en cours d'exécution.

<u>Attention, aux modifications de variable via la portée **script**</u>.

Pour modifier le contenu d'une variable déclarée via *Properties* et rendre cette modification persistante pour les autres tâches, l'usage de la portée script: est nécessaire.

Ceci amène un effet de bord à connaitre. La portée script: est celle du module et pas celle du script Psake, la suppression de ces variables reste à votre charge à l'aide du cmdlet **Remove-Variable**.

Consultez le script nommé 'ScopeAndProperties.ps1' pour le détail.

Note :

Selon les cas les modules importés dans une tâche devront l'être en utilisant le paramètre –***Global***.

6.7.2 Paramétrer une propriété

Le paramètre **-parameters** de la fonction *Invoke-psake* sert d'une part à configurer le script Psake et d'autre part permet de paramétrer une déclaration *Properties* :

```
Invoke-Psake .\Release.ps1 –parameters @{"Config"="Debug"}
```

La propagation de paramètre pouvant également provenir d'un script principal appelant des tâches Psake :

```
[CmdletBinding(DefaultParameterSetName = "Debug")]
 Param(
     [Parameter(ParameterSetName="Release")]
   [switch] $Release
 )

 #Configure Debug ou Release
Invoke-Psake .\Release.ps1 –parameters @{"Config"="$($PsCmdlet.ParameterSetName)"}
```

6.8 Construction imbriquée

L'imbrication de script de construction au sein d'une tâche est possible :

```
Task RunNested {
   Invoke-psake .\nested\nested.ps1
}
```

6.9 Assertion

Le mot clé ***Assert*** précise une condition validant une condition que vous jugez nécessaire au bon déroulement du script. Dans le cas où la condition n'est pas remplie une exception est déclenchée :

```
$OS=(Get-WmiObject Win32_operatingsystem).Caption
Assert ($OS -match "Windows.2008.R2") "Invalid Operating system:'$OSName'"
Assert: Invalid Operating system : 'Microsoft Windows 7 Édition Intégrale N '
```

Voir aussi : http://smeric.developpez.com/java/astuces/assertions/

7 La variable $Psake

Cette variable porte diverses informations de configuration et d'exécution.

C'est une hashtable dont les clés peuvent référencer différentes structure de données. Par exemple *$Psake.config_default* est un PSObject et *$Psake.context* est une pile LIFO (Last-In-First-Out).

La clé nommée 'c*ontext*' contient à son tour de nombreuses informations dont la liste des tâches, celles exécutées, etc. Son contenu est mis à jour durant l'exécution du script *Psake* et est réinitialisé une fois le script terminé.

7.1 Déterminer la présence d'une tâche default

Lors de la construction d'un script Psake on peut vouloir utiliser une tâche dans deux scripts différents tout en évitant de dupliquer le code de cette tâche.

Par exemple on souhaite inclure cette tâche dans un script principal ou l'exécuter directement par *Invoke-Psake*. Dans le premier cas la déclaration de la tâche 'default' se fera dans le script principal l'inclusion ne posera pas de problème, en revanche en utilisation directe il lui manquera cette déclaration.

La solution est de tester le nombre de tâche porté par la variable *$Psake.Context* :

```
 #Fichier comportant une seule déclaration de tâche
if ($Psake.Context.tasks.Count -eq 0)
{ Task default -Depends FaitqqChose }

Task FaitqqChose  { 'Fait qq chose' }
```

On peut également tester si le nom du script courant, *$Psake.build_script_file*, correspond au nom du fichier courant.

Pour d'autres usages avancés, les informations suivantes pourraient être mises à profit :

$Psake.Context.executedTasks : liste des tâches exécutées,

$Psake.Context.callStack : liste des tâches restantes à exécuter.

7.2 Exécution d’un script par défaut

Si le paramètre ***–buildFile*** n’est pas renseigné Psake recherche dans le répertoire courant un script nommé *Default.ps1* et l’exécute.

8 Exécution de programme externe

Psake propose le mot clé ***exec***, celui-ci facilite la validation de l’exécution d’un programme externe au sein d‘une tâche :

```
$Target=$env:Temp ; $Encoding='UTF8'
@'
Task default -Depend PrgExterneNOK

Task PrgExterneNOK -Depend PrgExterneOK {
  exec { cmd /c exit 1 } -errorMessage "Echec du programme externe"

}
Task PrgExterneOK {
  "Appel d’un programme externe"
  exec { cmd /c exit }
}
'@|Set-content "$Target\PrgExterne.ps1" -encoding $Encoding
Invoke-PSake "$Target\PrgExterne.ps1"
```

9 Règles syntaxiques / BNF

```
<BuildScript> ::= <Includes>
                | <Properties>
                | <FormatTaskName>
                | <TaskSetup>
                | <TaskTearDown>
                | <Tasks>

<Includes> ::= Include <StringLiteral> | <Includes>

<Properties> ::= Properties <ScriptBlock> | <Properties>

<FormatTaskName> ::= FormatTaskName <Stringliteral>

<TaskSetup> ::= TaskSetup <ScriptBlock>
<TaskTearDown> ::= TaskTearDown <ScriptBlock>

<Tasks> ::= Task <TaskParameters> | <Tasks>
<TaskParameters> ::= -Name <StringLiteral>
                       | -Action <ScriptBlock>
                       | -PreAction <ScriptBlock>
                       | -PostAction <ScriptBlock>
                       | -PreCondition <ScriptBlock>
                       | -PostCondition <ScriptBlock>
                       | -ContinueOnError <Boolean>
                       | -Depends <TaskNames>
                       | -Description <StringLiteral>
<TaskNames> ::= <StringLiteral>, | <TaskNames>

<ScriptBlock> ::= { <PowerShellStatements> }

<Boolean> ::= $true | $false
```

10 Conclusion

L'outil Psake n'est pas en soi difficile à manipuler, il propose une mécanique simple et robuste, et comme tout le reste demande une phase d'apprentissage.

Les exemples sont basés sur des constructions de code, mais Psake peut servir pour organiser d'autres tâches, un exemple ici: http://thomasvm.github.io/blog/2012/10/02/introducing-unfold/.

Enfin n'hésitez pas à consulter les bugs sur le site de Psake, car il se peut qu'il faille quelques adaptations pour les dernières versions de Powershell.