Les workflows PowerShell.

Par Laurent Dardenne, le 03/03/2015.

Conçu avec Powershell version 4 x64 sous Windows Seven FR.

Un grand merci à **Matthew Betton** d'avoir le pris le temps de me relire :-)

Merci également à 6ratgus pour ses tests.

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

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

Chapitres

1 Workflow ? Késako ?

« On appelle workflow la modélisation et la gestion informatique de l'ensemble des tâches à accomplir et des différents acteurs impliqués dans la réalisation d'un processus métier.

Il décrit le circuit de validation ou de traitement, les tâches à accomplir entre les différents acteurs d'un processus, les délais, les modes de validation, et fournit à chacun des acteurs les informations nécessaires pour la réalisation de sa tâche. Il permet ainsi d'automatiser les flux d'informations entre différents processus au sein de l'entreprise.

Le workflow est composé d'activités, correspondant à des tâches ou fonctions à réaliser pour compléter une étape du processus. »

Dixit : http://ics.utc.fr/c2m/DOCS/L2e/html/co/Workflow.html

Notez que dans un workflow des décisions doivent être prises et celles-ci peuvent être manuelles et à leur tour dépendre d'autres tâches.

1.1 Le principe sous Powershell

Pour faciliter l'écriture de workflows par des administrateurs, l'équipe Powershell a conçu une passerelle entre Powershell et le framework de développement Windows Workflow Foundation (**WF**) qui lui-même facilite l'incorporation de workflows dans les applications .NET.

Comme on se base sur les compétences Powershell acquises, l'apprentissage de l'écriture de workflows ne nécessite pas de connaissances d'un langage compilé tel que le C# ou VB .net, ni de parcourir le détail des classes dédiées sur MSDN. Ceci dit cela ne va pas se faire tout seul, comme tout le reste d'ailleurs ☺.

1.2 Qu'apporte Windows PowerShell workflow (PSWF) ?

WF permet le développement de workflows pilotés par des modèles (*model-driven workflow development*), en fournissant une conception visuelle (sous Visual Studio) tout en masquant les problèmes système, telles que les transactions, la gestion d'état et le contrôle d'accès concurrentiel.

Cette passerelle entre Powershell et WF offre de nouvelles fonctionnalités en nous déchargeant de leurs implémentations techniques. Souvenez-vous que Powershell se concentre sur le *quoi faire* et pas sur le *comment le faire*, ici aussi au travers des workflows il nous rend service.

Par exemple l'exécution d'un workflow peut continuer après un redémarrage de la machine, voir être interrompu puis relancé depuis son point d'interruption. Cette interruption pouvant être de plusieurs jours, l'état du workflow sera sauvegardé et rechargé depuis un disque.

Les workflows s'appuyant sur le remoting PowerShell, ils leur est possible de se reconnecter automatiquement à une machine distante (*managed node*). Voir de tenter de s'y reconnecter à intervalle régulier (retry) en cas de coupure réseau.

L'exécution d'activités en parallèle est également possible.

1.3 Code, script, WF ? WTF !

Si un script est constitué d'une suite d'instructions, un workflow est constitué par une suite d'activités.

Une activité est l'unité élémentaire d'un workflow, c'est une tâche à accomplir (simple ou complexe). Qu'elle le soit par une personne ou par un système (informatique ou autre), tous deux étant des acteurs ayant un rôle à jouer.

Vous trouverez dans ce glossaire en anglais de Windows Workflow Foundation la définition des termes utilisés dans les diverses documentations Microsoft.

Par exemple l'expression *long-running task* référence des actions implicites qu'une traduction sommaire ne porte pas. Le terme *Argument d'activité* est similaire à celui de paramètre dans Powershell. De plus un workflow est créé dans le provider de fonction, ce qui dans les premiers temps apporte quelques confusions quant à savoir ce qu'on manipule exactement.

Pour débuter sur le sujet, on peut ne pas s'appesantir sur ces points et se contenter de ceci :

Powershell **décrit** un workflow et le transforme, Windows Workflow Foundation l'**exécute**.

1.4 La documentation existante

Powershell V4 propose les fichiers d'aide offline suivants :

about_Workflows	*about_Suspend-Workflow*
about_WorkflowCommonParameters	*about_Checkpoint-Workflow*
about_ActivityCommonParameters	*about_Foreach-Parallel*
about_InlineScript	*about_Parallel*
about_Sequence	

Ces fichiers se trouvent dans le répertoire : "$psHome\Modules\PSWorkflow\en-US".

La documentation Powershell online, celle de Windows Workflow Foundation.

1.4.1 Mots-clés dédiés

Le premier est **Workflow**, une fois celui-ci précisé d'autres mots clés deviennent accessibles :

Parallel, Sequence, InlineScript, Checkpoint-Workflow, Suspend-Workflow

L'AST traduit le premier comme une définition de fonction et les trois derniers comme des commandes. La syntaxe de l'instruction ForEach est étendue, par l'ajout de l'option *–Parallel.*

Deux modules sont dédiés aux workflows : PSWorkflow et PSWorkflowUtility. Ceux-ci proposent les cmdlets suivants :

Invoke-AsWorkflow, New-PSSessionExecutionOption et New-PSWorkflowSession.

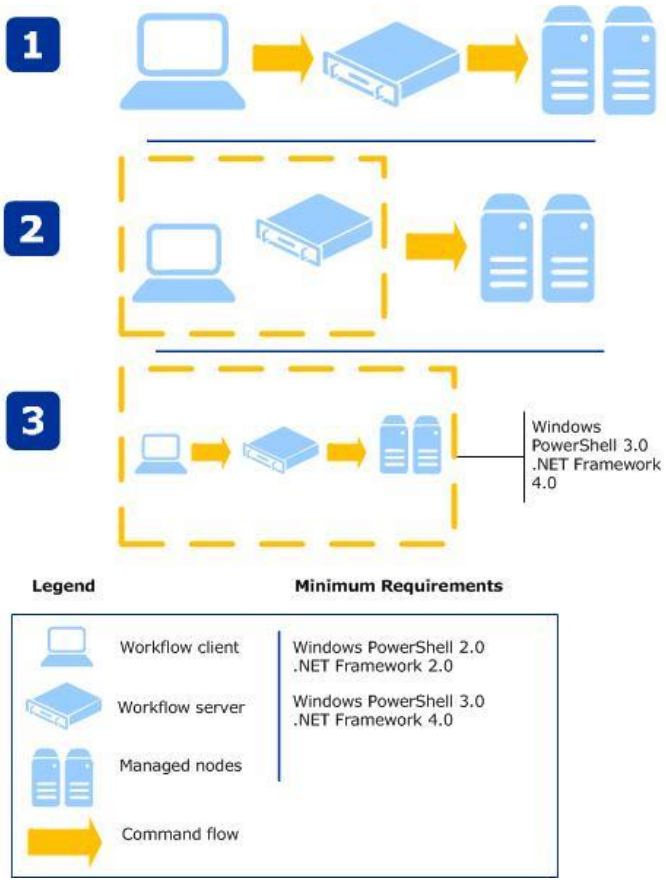
Il existe également de nouvelles variables de préférence dédiées aux Workflows :

$PSParentActivityId, $PSPersistPreference et *$PSRunInProcessPreference*

Les jobs de Workflow ajoutent deux cmdlets, il s'agit de ***Resume-Job*** et ***Suspend-Job***.

1.5 Configuration d'installation

Le diagramme suivant présente les configurations d'installation possibles pour les workflows, ce tutoriel se base sur la configuration 3 :



- **Configuration 1** : le serveur de workflow et le client de workflow sont en cours d'exécution sur des ordinateurs distincts. Les nœuds gérés ne se trouvent pas sur les mêmes ordinateurs ou périphériques que ceux exécutant le serveur et le client de workflow. Notez que le flux de commandes va du client de workflow aux nœuds gérés, en passant par le serveur de workflow. Dans ce scénario, il n'est pas nécessaire d'installer Windows PowerShell 3.0 sur l'ordinateur client.
- **Configuration 2** : le serveur de workflow et le client de workflow sont tous les deux en cours d'exécution sur le même ordinateur. Les versions de Windows PowerShell et du .NET Framework qui sont en cours d'exécution sur cet ordinateur satisfont la configuration requise la plus exigeante, celles du serveur de workflow. Les nœuds gérés sont en cours d'exécution sur un ou plusieurs ordinateurs distants.
- **Configuration 3** : le serveur de workflow, le client de workflow et les nœuds gérés sont en cours d'exécution sur le même ordinateur ou périphérique. Windows PowerShell 3.0 est requis dans ce scénario.

Consultez sur Technet le détail de ces configurations.

2 Créer un workflow

En prérequis nous devons charger le module PSWorkflow :

```
Import-Module PSWorkflow
```

La session Powershell doit être une session 64 bits sinon on obtient l'erreur suivante :

```
Import-Module PSWorkflow
```

Import-Module : Le workflow Windows PowerShell n'est pas pris en charge dans une console Windows PowerShell x86.

Un workflow est écrit à l'aide du langage PowerShell, comme ceci par exemple :

```
Workflow Test {
  Param($Parametre)
 Write-Output "Hello"
}
```

Ce code a un air de déjà vue, car hormis le mot clé ***Workflow***, il est similaire à une déclaration de fonction, comme son exécution :

```
Test
```

Hello

C'est principalement en cela qu'il est aisé de déclarer des workflows avec Powershell.

A la différence d'une déclaration de fonction, vous remarquerez que lors de la validation de la déclaration d'un workflow, Powershell prend plus de temps avant de retourner au prompt. Essayons d'en trouver la raison.

Avant d'aller plus loin déclarons une fonction contenant le même code que le workflow :

```
Function Test2 {
  Param($Parametre)
 Write-Output "Hello"
}
```

On utilise le nom 'Test2', car le provider de fonction ne peut contenir qu'une seule entrée nommée 'Test'.

Explorons la commande créée lors de la déclaration de notre workflow :

```
$Wcmd=Get-Command Test
$Wcmd.CommandType
```

Workflow

```
$Wcmd.GetType().Fullname
```

System.Management.Automation.WorkflowInfo

Puis celle de la fonction :

```
$Fcmd=Get-Command Test2
$Fcmd.CommandType
```

Function

```
$Fcmd.GetType().Fullname
```

System.Management.Automation. FunctionInfo

La commande de type *Workflow* est bien une instance d'une nouvelle classe de métadonnées.

Maintenant comparons le code de chaque commande :

```
$Fcmd.ScriptBlock
```

```
 Param($Parametre)
  Write-Output "Hello"
```

```
$Wcmd.ScriptBlock
```

```
 [CmdletBinding()]
 param (
    [System.Object] $Parametre,
    [hashtable[]] $PSParameterCollection,
    [string[]] $PSComputerName,
    [ValidateNotNullOrEmpty()] $PSCredential,
    [uint32] $PSConnectionRetryCount,
…
```

On constate une modification du code. Comparons les propriétés des deux types :

```
$Workflow=$Wcmd.psobject.Properties.name
$Function=$Fcmd.psobject.Properties.name
Compare-Object $Workflow $Function
```

```
InputObject                 SideIndicator
-----------                 -------------
XamlDefinition              <=
NestedXamlDefinition        <=
WorkflowsCalled             <=
```

Puis affichons le contenu de la propriété *XamlDefinition :*

```
$Wcmd.XamlDefinition
```

```
<Activity
   x:Class="Microsoft.PowerShell.DynamicActivities.Activity_958253109"
   xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
   xmlns:sad="clr-namespace:System.Activities.Debugger;assembly=System.Activities"
   xmlns:local="clr-namespace:Microsoft.PowerShell.DynamicActivities"
..
```

C'est une déclaration XML, plus particulièrement du XAML. Notez le nom de balise *Activity*.

On y retrouve notre code Powershell appelant ***Write-Output*** transformé en ceci :

```
    <ns3:WriteOutput>
      <ns3:WriteOutput.InputObject>
        <ns2:InArgument x:TypeArguments="ns4:PSObject[]">
          <ns1:PowerShellValue x:TypeArguments="ns4:PSObject[]" Expression="&amp;quot;Hello&amp;quot;" />
        </ns2:InArgument>
      </ns3:WriteOutput.InputObject>
    </ns3:WriteOutput>
```

Donc, la raison pour laquelle Powershell ne retourne pas rapidement au prompt est que la transformation du code Powershell vers WF (Windows **W**orkflow **F**oundation) nécessite un nombre plus important de traitements et de validations de règles que pour une fonction.

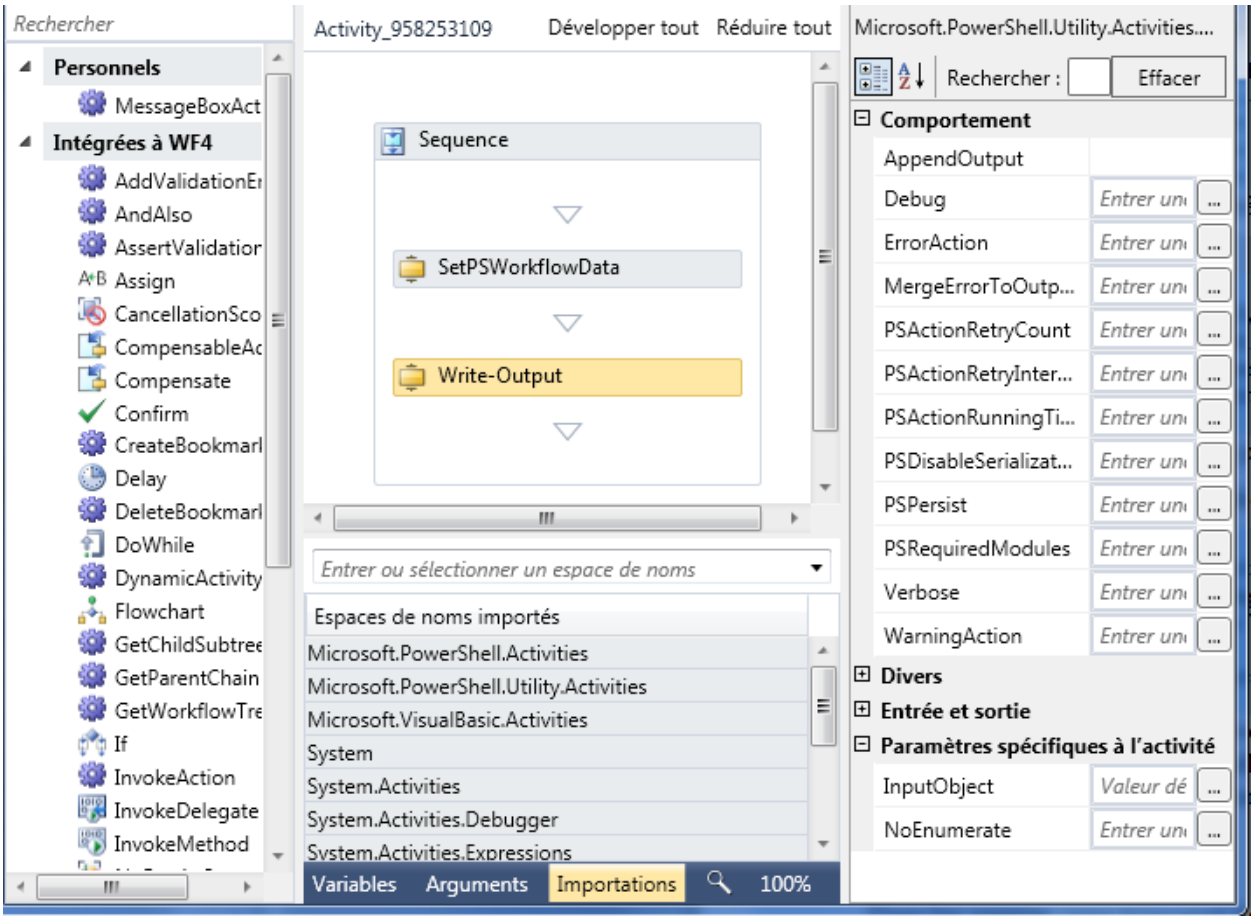
2.1 FlowPad

Jérémy Jeanson a créé un outil de visualisation de workflow à partir d'un fichier .xaml nommée Flowpad. Vous le trouverez dans le répertoire *Tools\FlowPad*.

Auparavant, enregistrons dans un fichier texte la définition XAML de notre Workflow 'Test' :

```
$Wcmd.XamlDefinition > C:\Temp\Test.xaml
FlowPad.exe
```

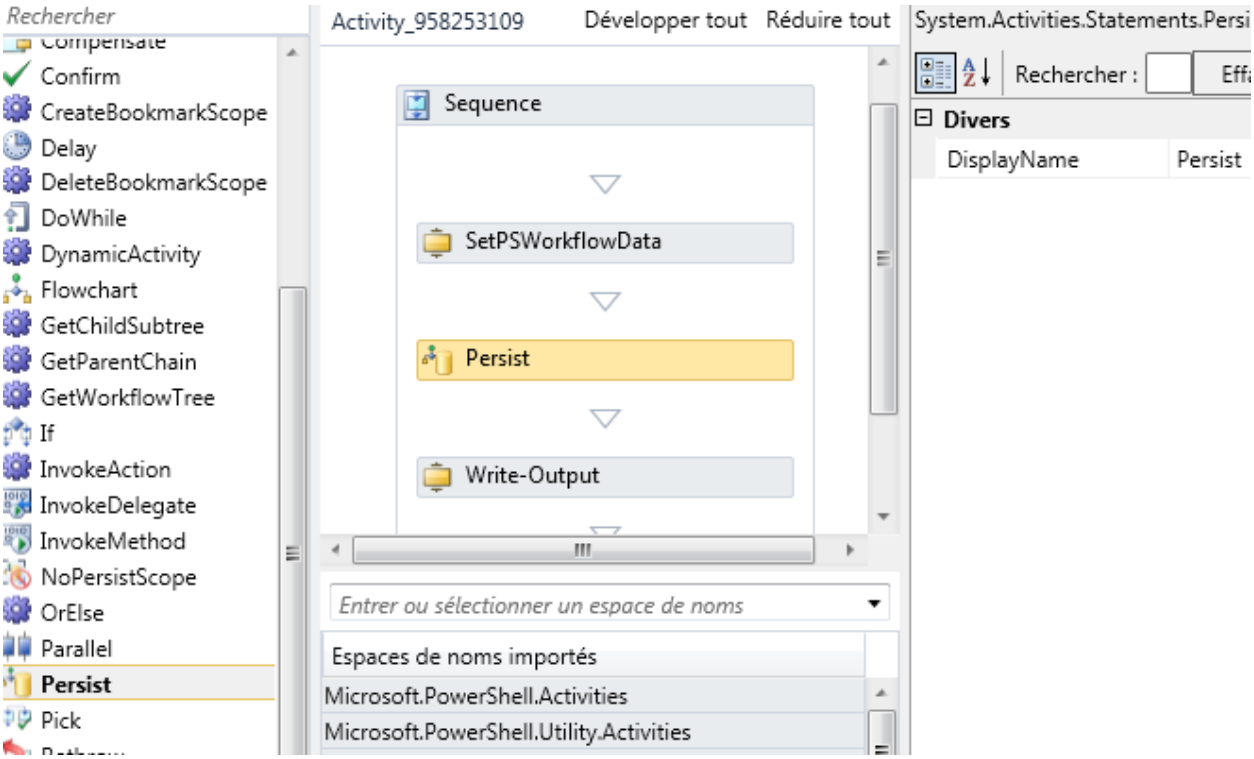
Une fois l'outil exécuté, utilisez le menu **'Ouvrir**' puis sélectionnez le fichier 'C:\Temp\Test.xaml'. Ce qui affichera ceci :



La partie centrale contient le visuel de notre Workflow, celui-ci est composé d'activités.

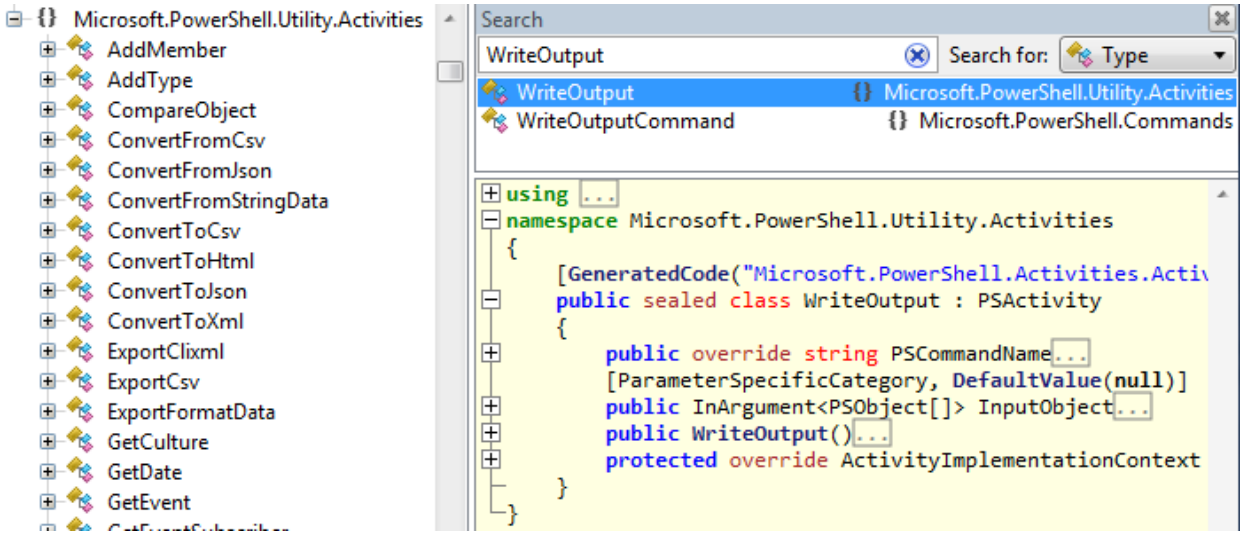
La partie droite contient les propriétés de l'activité *WriteOutput*, ceux indiqués 'spécifique' concernent les paramètres du cmdlet ***Write-Output***.

La partie gauche contient les composants graphiques représentant des activités utilisables dans notre workflow, ici j'ai ajouté l'activité nommée 'Persist' :



La dernière partie contient la liste des assemblies chargé via le fichier .xaml.

Affichons, à l'aide l'outil [Ilspy](), le détail de l'assembly contenant l'activité **WriteOutput** :



Voilà donc ce qu'est à l'origine une activité, il s'agit d'une classe spécialisée.

2.2 Passerelle

Dans la recopie d'écran précédente on constate qu'il existe une commande nommée *WriteOutputCommand* et une activité nommée *WriteOutput.*

C'est le traitement de transformation qui se charge de la correspondance, si elle existe, entre un cmdlet Powershell et son activité WF. Dans un workflow Powershell la notion d'activité est sous-jacente, on n'utilise que du code Powershell.

L'usage de la notion de passerelle est personnel, car le terme de gateway n'existe pas dans la documentation Powershell. Il s'agit plutôt d'une transformation basée sur des correspondances et validation de règles que nous aborderons dans un prochain chapitre.

2.3 Workflow ou fonction ?

Comme indiqué précédemment la déclaration d'un workflow crée une entrée dans le provider de fonction, et bien qu'il s'agisse d'un autre type de code, cette approche est nécessaire.

Dans les premières versions de Powershell v3, l'information de workflow était portée par un attribut :

```
 #code PS v3 ctp
function Test {
 [Workflow()]
 param()
 begin {
   "Hello World"
 }
}
```

On avait donc un seul mot-clé pour deux types d'objets. La version finale propose deux mots-clés et deux types d'objets, mais toujours un seul provider.

L'usage du provider de fonction permet de stocker en mémoire les différentes définitions d'un workflow tout en limitant le nombre de cycle de transformation. Ce type d'optimisation à également lieu lors de l'import d'un module alors que pour un script .ps1 le parsing a toujours lieu lors de son exécution.

Le principal usage du provider de fonction est que chaque instance de workflow doit être exécutée dans un hôte de type *WorkflowApplication* qui charge le runtime WF. Powershell masque l'implémentation de cet hôte en exécutant chaque workflow dans un job.

C'est ce qui fait qu'on puisse démarrer simplement un workflow en ligne de commande.

Le dernier point que nous n'avions pas traité lors de notre exploration initiale est justement la création de cet hôte :

```
Workflow Test {
  Param($MonParametre)
 Write-Output "Hello"
}
```

```
$Wcmd=Get-Command Test
```

Chargeons le code de notre workflow dans un éditeur :

```
$Wcmd.ScriptBlock > c:\temp\sbWFTest.ps1
#ii c:\temp\sbWFTest.ps1
Notepad c:\temp\sbWFTest.ps1
```

2.3.1 L'invocation de la Trinité

La déclaration d'un workflow crée bien une fonction, mais le code de cette fonction n'a rien à voir avec celui du workflow. Le runtime Powershell génère ce code comme il génère la définition du code XAML, une seule déclaration de workflow, mais deux définitions de code :

```
$Cmd=Get-Command Test
$Cmd.Definition
```

```
 Param($MonParametre)
Write-Output "Hello"
```

```
$Cmd.ScriptBlock
```

```
 [CmdletBinding()]
  param (
      [System.Object] $MonParametre,
       [hashtable[]] $PSParameterCollection,
        [string[]] $PSComputerName, …
```

La première partie de ce scriptblock est une signature de fonction avancée contenant les noms des paramètres précisés dans la clause *Param* **et** des paramètres communs à chaque fonction exécutant un workflow. Ceux-ci sont ajoutés automatiquement. Leur description se trouve dans le fichier d'aide *about_WorkflowCommonParameters*.

La seconde partie de ce code est constitué des blocs **begin**, **process** et **end**.

Le bloc ***begin*** contient la définition d'une fonction portant le même nom que le workflow, ici *'Test'*. Sa liste de paramètres est identique à celle contenue dans la première partie, excepté que le paramètre nommé '$*PSParameterCollection'* n'existe plus et que le paramètre '*$PSInputCollection*' a été ajouté. Son rôle est de renvoyer la liste des paramètres liés.

Le bloc ***process*** insère les objets reçus du pipeline dans la variable *$PSInputCollection*.

Le bloc ***end*** effectue des validations fonctionnelles sur les paramètres, puis exécute un job.

Notez que le workflow peut recevoir des données du pipeline avant son exécution et celle-ci l'étant dans un job implique qu'on reçoive des objets désérialisés.

2.3.2 Job synchrone

Si on ne précise pas le paramètre *–AsJob* lors de l'invocation du workflow celui-ci est tout de même exécuté dans un job, mais Powershell attend la fin de son exécution avant de passer à l'instruction suivante.

Chaque instance de workflow est exécutée dans un job de type ***PSWorkflowJob***, qui propose en interne, à la différence des autres types de job, une propriété *SynchronousExecution*.

Ce job est créé via la méthode *NewJob()* de l'objet ***JobManager*** :

```
$ExecutionContext.host.Runspace.JobManager
```

Dans le scriptblock du workflow la méthode statique suivante se charge de cet appel :

```
[Microsoft.PowerShell.Commands.ImportWorkflowCommand]::StartWorkflowApplication
```

```
OverloadDefinitions
-------------------
static System.Management.Automation.ContainerParentJob StartWorkflowApplication(
                        System.Management.Automation.PSCmdlet command,
                        string jobName,
                        string workflowGuid,
                        bool startAsync,
                        bool parameterCollectionProcessed,
                        hashtable[] parameters)
…
```

L'exemple suivant exécute le workflow dans job synchrone, car dans ce cas le paramètre *startAsync* à la valeur $False :

```
Workflow Test { Param($MonParametre)  Write-Output "Fin" }
Test
```

```
Fin
```

Si dans le workflow on suspend son exécution, alors son code renvoi un job dans l'état *'Suspended'* :

```
Workflow Test {
  Param($MonParametre)
  Suspend-Workflow
  Write-Output "Fin"
}
Test
```

```
Id   Name     PSJobTypeName    State       HasMoreData   Location      Command
--   ----     -------------    -----       -----------   --------      -------
4    Job4     PSWorkflowJob    Suspended   True          localhost     Test
```

On se retrouve désormais avec un job asynchrone suspendu que l'on doit relancer :

```
Resume-Job -id 4
```

```
Id   Name     PSJobTypeName    State       HasMoreData   Location      Command
--   ----     -------------    -----       -----------   --------      -------
4    Job4     PSWorkflowJob    Running     True          localhost     Test
```

Et puisque désormais on manipule un job asynchrone, on doit récupérer le résultat explicitement :

```
Receive-Job -id 4
```

```
Fin
```

```
Get-Job
```

```
Id     Name      PSJobTypeName     State         HasMoreData     Location        Command
--     ----      -------------     -----         -----------     --------        -------
4      Job4      PSWorkflowJob     Completed     True            localhost       Test
```

Une fois ceci fait il reste à le supprimer :

```
Remove-Job -id 4
```

Si on ne précise pas le paramètre *–AsJob*, la fonction associée au workflow se charge de ces opérations.

Côté utilisateur, l'implémentation d'un workflow Powershell est donc une fonction démarrant un job qui exécute une instance d'un workflow WF.

2.4 Workflow appelant un workflow

Si la définition d'un workflow se trouve dans le provider de fonction et qu'il est exécuté par le runtime de WF, la résolution de ses dépendances doit se faire avant son exécution :

```
Workflow Test {
  Param($Parametre)
 Write-output "Version 1"
}
$W=Get-Command Test
$W.NestedXamlDefinition
```

```
#vide
```

```
Workflow Autre_WF {
  Param($Parametre)
 Test
}

$W.NestedXamlDefinition
```

```
< Activity
  x:Class="Microsoft.PowerShell.DynamicActivities.Activity_1488400146_Nested"
…
```

```
$W.XamlDefinition
```

```
< Activity
  x:Class="Microsoft.PowerShell.DynamicActivities.Activity_1488400146"
…
```

```
Test
```

```
Version 1
```

```
Autre_WF
```

```
Version 1
```

Notez que dans les métadonnées du workflow 'Test' la propriété *NestedXamlDefinition* est mise à jour une fois que le second workflow le référence et que seul le nom de classe diffère dans le code XAML.

Voyons ce qui change dans ce cas pour ‘Autre_WF’ :

```
$Wcmd=Get-Command Autre_WF
$Wcmd.WorkflowsCalled
```

```
CommandType     Name           ModuleName
-----------     ----           ----------
Workflow        Test
```

La définition XAML du workflow nommé ‘*Autre_WF’* contient un bloc dédié :

```
$Wcmd.XamlDefinition > c:\temp\Autre_wf.xaml
notepad c:\temp\Autre_wf.xaml
```

```
 <ns1:InlineScript Command="trap { break }&#xD;&#xA;function Test
```

Ce bloc contient une référence au nom du workflow enfant, nous pouvons nous arrêtez là.

Il reste toutefois un détail d’implémentation à connaitre. Le workflow référencé, c’est-à-dire ’*Test*’, est compilé en une dll dans le répertoire suivant :

```
"$env:LOCALAPPDATA\Temp\PSWorkflowCompilation"
   #ou
[System.IO.Path]::GetTempPath() +"PSWorkflowCompilation"
```

Cette dll est chargée automatiquement et reste verrouillée pendant la durée de vie du host Powershell (la console, ISE,…) l’ayant généré. Notez que le nom de fichier .dll ainsi que le nom du répertoire l’hébergeant contiennent le même GUID.

Chaque déclaration de ce workflow dans une nouvelle session Powershell créera un nouveau répertoire, dont le nom sera un autre GUID contenant une nouvelle dll. Le contenu de ce répertoire doit donc être nettoyé régulièrement. Nous verrons que les jobs persistants ne sont pas impactés par cette suppression.

Exécutons une seconde fois la déclaration de workflow ‘Autre_WF’ :

```
Workflow Autre_WF {
  Param($Parametre)
 Test
}
```

Je suppose que le parseur constate qu’il existe déjà un workflow de même nom et que son code et ses références n’ont pas changées et n’opère donc aucun changement.

Modifions le workflow ‘Test’ référencé dans le workflow ‘Autre_WF’ :

```
Workflow Test {
  Param($Parametre)
 Write-output "Version 2"
}
```

Puis exécutons le workflow ‘Autre_WF’ :

```
Autre_WF
```

```
Version 1
```

```
Test
```

```
Version 2
```

Le workflow ‘Test’ est bien modifié, mais le workflow ‘Autre_WF’ référence toujours l’ancienne version, celle de la dll compilée. En effet celle-ci contient la définition XAML du workflow ‘Test’ en tant que ressources. **Nous avons affaire ici à une référence statique.**

Si on exécute maintenant une troisième fois la déclaration de ‘Autre_WF’ :

```
Workflow Autre_WF {
  Param($Parametre)
  Test
}
Autre_WF
```

```
Version 2
```

Dans ce cas Powershell crée une seconde DLL et ce workflow référence bien la dernière version déclarée. Ceci permet de s’assurer que la première version du workflow ‘Autre_WF’ continu de fonctionner, par exemple si elle est cours d’utilisation dans un job.

2.4.1 Autre exemple

```
workflow One { #From Technet
    Write-output "One"
    Two  #-AsJob -> Error
    workflow Two {
        Write-output "Two"
        Three
        Function Three  {
            Write-output "Three"
            function Five {Write-output  "Five"}
            Workflow Four {Write-output "Four" ; Five}
            Four #-AsJob OK -> implicite inlineScript
        }
}  }
```

Bien que les deux workflows ‘Two’ et ‘Four’ soient imbriqués, et à la différence des fonctions imbriquées, ils sont créés dans le provider de fonction et dans la portée courante :

```
Get-Command -CommandType Workflow
```

```
CommandType     Name        ModuleName
-----------     ----        ----------
Workflow        Four
Workflow        One
Workflow        Two
```

Note : l’usage du paramètre *–AsJob* n’est pas possible sur un appel à un workflow imbriqué.

2.4.2 Erreur de compilation

En cas d'erreur de compilation de la dll, Powershell affichera un message du type :

```
Erreur de compilation lors de la création des workflows dépendants. Pour plus d'informations, consultez :

C:\Users\AccountName\AppData\Local\Temp\PSWorkflowCompilation\Workflow_ac367129c0dd4cacad9ca69f4316ce82\Project\Build.Log.
```

Ce fichier de log contient des informations résultant de l'exécution d'un projet de compilation MSBuild construit en interne par Powershell.

2.4.3 Paramètres communs et niveau d'imbrication

Les workflows imbriquées ayant trois niveaux de profondeur ne supportent pas les paramètres communs :

```
workflow One {
    Write-Output "One"
}

workflow Two {
    One -ErrorAction Stop
}

workflow Three {
    Two -ErrorAction Stop
    # Two -Debug
}
```

```
Paramètre nommé «ErrorAction» introuvable. Les paramètres pris en charge sont: Debug, ErrorAction, Input,…
```

Dans l'exemple suivant on utilise un paramètre commun de workflow, dans ce cas le message d'erreur est plus explicite :

```
workflow Three {
    two -PSDebug
}
```

```
Le paramètre nommé «PSDebug» est introuvable. Les paramètres communs de workflows, tels que PSComputerName, ne sont pas pris en charge dans les workflows imbriqués qui possèdent déjà des workflows imbriqués.
```

Ces messages d'erreur concernent bien la même règle, ils seront peut être identique dans une prochaine version.

3 Activité

Après cette dizaine de pages nous n'avons toujours pas abordé le cœur du sujet qui est l'écriture d'un workflow, il reste à détailler ce qu'est une activité Windows **W**orkflow **F**oundation.

Nous avons vu qu'au cmdlet **Write-Output** correspond une activité nommée *WriteOutput*. Celle-ci se trouve dans l'assembly *Microsoft.PowerShell.Utility.Activities.*

Cette activité dérive de la classe PSActivity comme la majorité des activités **WF** spécifique à Powershell. Voici le code portant la correspondance :

```
public sealed class WriteOutput : PSActivity
  {
    public override string PSCommandName
    {
      get
      {
        return "Microsoft.PowerShell.Utility\\Write-Output";
      }
    }
```

Le nom affiché dans le designer :

```
    public WriteOutput()
    {
      base.DisplayName = "Write-Output";
    }
```

3.1 Contexte d'activité

L'utilisation d'un cmdlet doit respecter les contraintes de WF, la plus importante étant que chaque activité doit avoir son propre contexte d'activité qui représente son environnement d'exécution, et qu'elle ne doit modifier que le sien. On parle aussi d'activité sans état.

Une activité Powershell couplée à un cmdlet est implémentée de la manière suivante :

```
protected override ActivityImplementationContext GetPowerShell(NativeActivityContext context)
{
  PowerShell powerShell = PowerShell.Create();
  PowerShell powerShell2 = powerShell.AddCommand(this.PSCommandName);
  if (this.InputObject.Expression != null)
  {
    powerShell2.AddParameter("InputObject", this.InputObject.Get(context));
  }
  return new ActivityImplementationContext
  {
    PowerShellInstance = powerShell
  };
}
```

La classe Powershell crée un runspace, ajoute un cmdlet, le paramètre, puis l'exécute et enfin renvoi un résultat si besoin. Cette classe répond parfaitement à ce besoin de cloisonnement.

Une activité ne partage pas de données avec une autre activité, mais en reçoit et en renvoi.

De suivre ce principe offre la possibilité en cas d'erreur de relancer le workflow là où il s'est arrêté sans avoir à réexécuter toutes les étapes précédentes. Un workflow est un enchainement d'activités et pas un enchainement d'instructions Powershell.

Prenons l'exemple d'un workflow créant un utilisateur puis envoyant un mail.

Supposons que la première activité réussisse, le workflow contient les informations de compte et de messagerie. Ceci sous réserve de prévoir des points de contrôle que nous aborderons dans un prochain chapitre.

Si la seconde activité échoue, par exemple le serveur mail est injoignable, le workflow peut être relancé immédiatement pour tenter un nouvel envoi de mail. Il n'est pas nécessaire de vérifier si le compte a été créé ou de relire les informations du compte nécessaires à la génération du contenu du mail.

Le workflow possède les données nécessaires à la réexécution de l'activité 'envoi de mail', toutefois vous devez vous assurer que cette reprise est fonctionnellement cohérente.

Ici on suppose implicitement que le compte utilisateur n'a pas été supprimé entre temps.

3.1.1 Exemples d'activités modifiant le contexte de l'appelant

Le cmdlet **Set-Variable** n'a pas d'activité correspondante, car il modifie l'état de l'appelant :

```
Workflow State {Set-Variable –Name Chemin –Value 'c:\temp' }
```

```
Impossible d'appeler la commande «Set-Variable».
D'autres commandes issues de ce module ont été empaquetées en tant qu'activités de workflow, mais cette commande a été exclue de manière spécifique …
```

Le parseur déclenche donc une exception.

Dans l'exemple suivant le cmdlet **Get-Process**, qui a une activité correspondante, crée en cas d'erreur une variable dans le contexte de l'appelant, mais ceci n'est pas supporté dans un workflow :

```
Workflow GP {
  Get-Process -ErrorVariable gpErreurs
}
```

Ce code déclenche une exception dans les versions 3 et 4. Ce bug est corrigé dans la version 5.

L'opérateur '**&**' et '**.**' (dot sourcing), ne sont pas supportés pour la même raison.

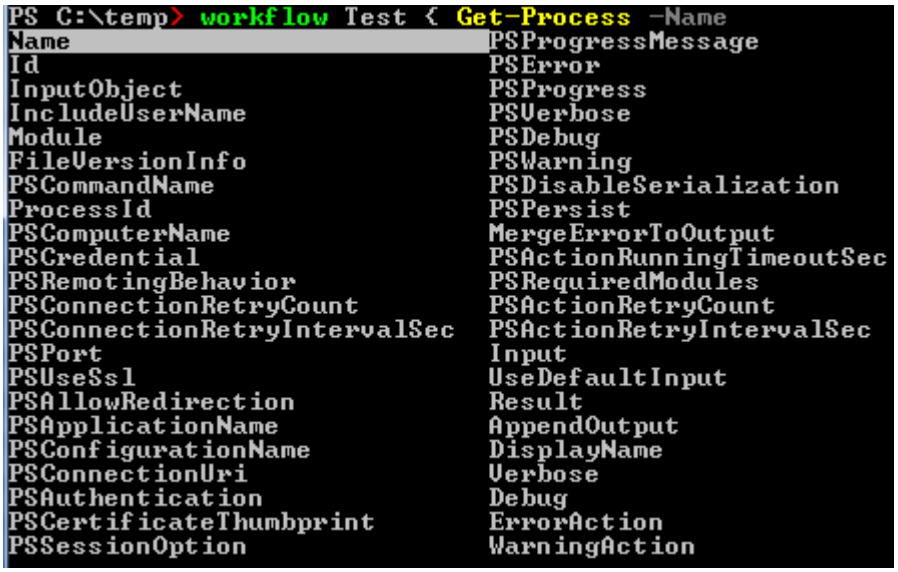
3.2 Paramètres commun d'activité

Powershell ajoute des paramètres communs aux activités, comme il ajoute des paramètres communs aux cmdlets. Seules les activités dérivées de la classe **PSActivity**, excepté les activités *Suspend-workflow* et *Checkpoint-workflow*, sont concernées par cet ajout.

Le fichier d'aide about_ActivityCommonParameters détaille la liste de ces paramètres.

Dans la recopie d'écran suivante, le module PSReadline nous aide (*Ctrl-Space*) à lister les paramètres du cmdlet ***Get-Process*** ainsi que ces paramètres communs encadrés de rouge :

Dans un workflow l'activité correspondante au cmdlet ***Get-Process*** proposera les paramètres suivants :



On peut utiliser *PSComputerName* (WinRM) au lieu de *Computername* (WMI/DCOM) ou bénéficier d'un nombre de reprise en cas d'échec de connexion en paramétrant *PSActionRetryCount*. Pour respecter le principe d'une activité énoncé précédemment, certains paramètre commun du cmdlet ne sont plus accessibles, par exemple *WarningVariable*.

Certaines de ces fonctionnalités sont paramétrable lors de l'exécution du workflow ou directement sur une activité, cette dernière possibilité primant le paramétrage du workflow.

3.3 Cmdlet sans activité correspondante

Si certains cmdlets ou instructions n'ont pas d'activité WF correspondante on peut utiliser l'activité dédiée nommée **InlineScript.**

Powershell encapsule automatiquement ces commandes dans une activité de ce type :

```
Workflow Test {
 (Get-ChildItem C:\Temp\*.txt).Name| out-String
}
$c=Get-Command Test
$c.XamlDefinition -split "`r`n"|select-string 'InlineScript'
```

```
#ras
```

Dans l'exemple précédent les deux cmdlets ont une activité correspondante, la transformation de code n'utilise pas l'activité **InlineScript**.

L'exemple suivant nécessite le module PSCX, mais vous pouvez utiliser le cmdlet *'\Source\Cmdlet\TouchFile.dll'*.

Dans le code suivant, l'appel du cmdlet ***Join-String*** issu du module PSCX, est inséré implicitement dans un appel à l'activité **InlineScript** :

```
Workflow Test {
  (Get-ChildItem C:\Temp\*.txt).Name | PSCX\Join-String -Separator '-*-'
}
$c=Get-Command Test
$c.XamlDefinition -split "`r`n"|select-string 'InlineScript'
```

```
        <ns1:InlineScript Command="PSCX\Join-String -Separator '-*-'">
…
```

Le code exécuté dans une activité **InlineScript** n'est pas soumis aux contraintes des workflows, c'est du code Powershell natif. Vous remarquez qu'il n'est pas possible de sélectionner les paramètres d'activités sur ***Join-String***.

Ce code utilise le chargement automatique de module, si le cmdlet est dans un module dont le répertoire n'est pas dans le chemin de recherche *$env:PsModulePath*, on peut utiliser explicitement l'activité **InlineScript** et disposer ainsi des paramètres d'activités :

```
Workflow Test {
  (Get-ChildItem C:\Temp\*.txt).Name |
 InlineScript {
   PSCX\Join-String -Separator '-*-'
 } -PSRequiredModules "C:\Program Files (x86)\PowerShell Community
Extensions\Pscx3\Pscx\Pscx.psd1"
}
```

L'activité chargera alors le module indiqué par le paramètre ***-PSRequiredModules***.

Précisez le nom du manifeste s'il en existe un.

3.3.1 Cmdlet transformé en activité

Powershell propose une API générant le code C# d'un wrapper d'activité basé sur la classe Powershell :

```
Import-Module PSWorkflow
$csFile='C:\Temp\JoinString.cs'
$cmd=Get-Command PSCX\Join-String
[Microsoft.PowerShell.Activities.ActivityGenerator]::GenerateFromCommandInfo($Cmd,'PSCX.JoinString') > $csFile
Type $csFile
```

```
namespace PSCX.JoinString
{
   /// <summary>
   /// Activity to invoke the Pscx\Join-String command in a Workflow.
   /// </summary>
   [System.CodeDom.Compiler.GeneratedCode("Microsoft.PowerShell.Activities.ActivityGenerator.Genera
]
   public sealed class JoinString : PSRemotingActivity
   {
      /// <summary>
      /// Gets the display name of the command invoked by this activity.
      /// </summary>
      public JoinString()
      {
         this.DisplayName = "Join-String";
      }
      /// <summary>
      /// Gets the fully qualified name of the command invoked by this activity.
      /// </summary>
      public override string PSCommandName { get { return "Pscx\\Join-String"; } }
…
```

La classe ActivityGenerator propose d'autres méthodes statiques, un bug sur MSConnect indique que celle nommée *GenerateFromModuleInfo* pose problème.

Il reste à compiler ce code C# dans une dll :

```
Import-Module PSWorkflow
  #Construit la liste des références nécessaires à la compilation
  #Il s'agit du chemin des DLL.
$References=@(
 ([Microsoft.PowerShell.Activities.PSActivity].Assembly.Location),
 ([System.Management.Automation.PSObject].Assembly.Location),
 ([System.Activities.Activity].Assembly.Location),
 ([System.Collections.Generic.List[string]].Assembly.Location), #Option
 ([System.ComponentModel.Component].Assembly.Location) #Option
)
$csFile='C:\Temp\JoinString.cs'
$code=Get-Content $csFile -Raw
Add-Type -TypeDefinition $Code -OutputAssembly 'C:\Temp\JoinString.dll' -OutputType Library -ReferencedAssemblies $References
```

Si on charge cette dll dans la session Powershell, le traitement de transformation n'en tiendra pas compte et continuera d'utiliser l'activité **InlineScript.**

Pour modifier ce comportement on doit spécifier la clause ***#Requires Assembly***, celle-ci n'est valide qu'au sein d'une déclaration de Workflow :

```
workflow Test{
  #Requires -Assembly "c:\temp\JoinString.dll"
  (Get-ChildItem *.txt).Name|Join-String -Separator '-*-'
}
```

Puisqu'il existe désormais une correspondance, ceci force le traitement de transformation à ajouter dans le code XAML une référence à l'assembly puis à utiliser l'activité au lieu du cmdlet encapsulé dans une activité **InlineScript** :

```
xmlns:ns5="clr-namespace:PSCX.JoinString;assembly=JoinString"…
```

Cette activité étant lié au module PSCX, la présence de celui reste nécessaire.

Pour les applications C# utilisant ce type d'activité, consultez ce bug.

3.3.2 Autre syntaxe d'appel d'activité

L'exemple suivant, proposé par un membre de l'équipe Powershell, utilise directement l'activité If , les paramètres correspondant aux noms de propriétés de la classe :

```
#requires –version 4.0
Workflow ContainerSimpleActivityOne
{
    #requires -Assembly "System.Activities, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35"

    System.Activities.Statements.If -Condition $true `
     -Then { "Got True" } -Else { "Got False" }
    System.Activities.Statements.If -Condition $false `
     -Then { "Got True" } -Else { "Got False" }
}
```

La version Powershell native étant :

```
Workflow ContainerSimpleActivityOne
{
  If ($true) { "Got True" } Else { "Got False" }
  If ($false) { "Got True" } Else { "Got False" }
}
```

3.3.3 Classes d'activités

Vous trouverez ici la hiérarchie des activités Powershell (PSActivity).

Dans les sources, le script *Find-ActivityNamespace.ps1* permet de les retrouver dynamiquement.

4 Portage de code existant

Si l'écriture de workflow sous Powershell est simplifié, le traitement de transformation, c'est à dire la traduction ou le portage du code PS en un code WF, ne supporte pas toutes les possibilités de Powershell, qui fidèle à son habitude nécessite de connaitre des comportements et pas seulement de la syntaxe.

Le remplacement du mot clé **Function** par **Workflow** est insuffisant, un code Powershell natif fonctionnel peut être invalidé lors de la transformation s'il ne respecte pas certaines règles spécifiques au Workflow, d'autres sont validées lors de l'exécution. Notez que la gestion d'événement n'est pas possible dans un Workflow Powershell.

4.1 Règles de validation

Dans le répertoire '\Documentation' les documents '*Known differences between PS Scripts and PS-Workflows'* et '*WMF3 CTP2 Windows PowerShell Workflow.pdf*' recensent ces incompatibilités entre le code PowerShell et WF.

Sachez que certains cmdlets n'ont pas d'activité correspondante et que d'autres sont limités à un usage local (consulter le détail).

Les variables automatiques suivantes ne sont pas supportées dans un workflow :

$Args	*$Error*	*$MyInvocation*	*$PID*
$PSBoundParameters	*$PsCmdlet*	*$PSCommandPath*	*$PSScriptRoot*
$StackTrace			

Par contre les suivantes y sont ajoutées :

```
Import-Module PSWorkflow
$Type=[Microsoft.PowerShell.Activities.PSWorkflowRuntimeVariable]
$Names=[System.Enum]::GetNames($Type)|
         Where { $_ -notmatch ('^All$|^Other$')}|
         Sort-Object
$code=@"
workflow VariableReport {
 $(  $Names|% { "Write-Output `"$('{0}:${0}' -F $_)`"`r`n" }  )
}
"@
iex $code
VariableReport
```

Les noms de variables identiques aux mots clés du VB dotNet posent problème comme indiqué dans ce bug. Enfin le nom de paramètre '***ID***' est à éviter dans la clause *Param* d'un workflow.

La fonction *\Source\Test-VariableNamedWithVBKeywords.ps1* recherche ces noms de variable.

4.1.1 Activités différentes de leur cmdlet

Les activités suivantes ont un comportement différent de leur cmdlet respectif :

Invoke-Expression :

Dans un workflow ce cmdlet supporte le paramètre *–Language*, qui permet d'insérer des balises XAML lors de la transformation du workflow :

```
Workflow Test
{
  Invoke-Expression -Language XAML `
   -Command '<WriteLine>["Ajout de la balise XAML WriteLine"]</WriteLine>'
}
```

La syntaxe du paramètre *–Command* est très restrictive, elle ne gère pas la substitution, ni l'usage de variable, mais uniquement une chaine de caractères.

New-Object :

Un workflow ne supporte que le paramètre *–Typename*, l'usage d'objet COM n'est donc pas supporté dans PSWF.

Restart-Computer :

Le paramètre *-Wait* est autorisé en local afin de redémarrer la machine exécutant le workflow.

Note :

Les cmdlets **Checkpoint-Workflow**, **Suspend-Workflow** sont en fait des pseudos commandes interprétées lors de la transformation :

```
Workflow Test {
    #--On peut utiliser ce nom de fonction
 function Checkpoint-Workflow {param ($path) dir -path $path}

   #--Mais ici PS ne référencera pas la fonction
 Checkpoint-Workflow -path c:\temp
}
```

```
La commande Checkpoint-Workflow n'accepte aucun paramètre.
```

Le mot clé *Persist* est également reconnu et transformé en un appel à la classe PSPersist.

4.2 Mots clés spécifiques

4.2.1 InlineScript

Comme nous l'avons vu ce mot clé est une activité dédiée à l'exécution de code Powershell natif. Par défaut celui-ci est exécuté dans un processus séparé ce qui implique que les données sont sérialisées.

Comme indiqué sur ce blog, le nombre maximum de processus simultanés est de 5. Cette limite apparaît dans l'exemple suivant :

```
ipmo PSWorkflow
  #Configuration par défaut.
New-PSWorkflowExecutionOption


Workflow Test {
  ForEach -Parallel ($I in 1..10) {
    InlineScript {Write-Warning "$Using:I - Start process"; Start-process
-FilePath Notepad.exe -Wait}
    InlineScript {Write-Warning "$Using:I - Dir C:\ " ; Dir -path
C:\Temp\*.txt|select -First 1; Start-Sleep -Seconds 60}
 }
}
```

Vous pouvez tester en modifiant la durée d'attente ou le nombre d'itération ou encore inverser les blocs **InlineScript**. La fin d'un processus *Notepad.exe* débloquera l'exécution du workflow.

Si on souhaite modifier localement ce comportement sans créer de configuration personnelle (ce qui est recommandé), on utilisera la variable de préférence ***$PSRunInProcessPreference***

```
Workflow Test {
  $PSRunInProcessPreference="True"
   ForEach -Parallel ($I in 1..10) {
     InlineScript {Write-Warning "$Using:I - Start process"; Start-process
-FilePath Notepad.exe -Wait}
  }
  $PSRunInProcessPreference=$null
   ForEach -Parallel ($I in 1..10) {
     InlineScript {Write-Warning "$Using:I - Start process"; Start-process
-FilePath Notepad.exe -Wait}
  }
}
```

A mon avis, cette solution est à utiliser en dernier recours.

4.2.2 Parallel

Chaque instruction contenue dans le bloc de code associé est exécuté en *Parallèle*, c'est-à-dire simultanément et dans un ordre quelconque :

```
Workflow Test {
 Write-Warning "Début"
 Parallel {
    Get-Service
    Get-Command
    Get-Process
 }
 Write-Warning "Fin"
}Test|% {$_.PStypenames[0]}
```

```
Deserialized.System.ServiceProcess.ServiceController      #Get-Service
Deserialized.System.Diagnostics.Process                   #Get-Process
Deserialized.System.Management.Automation.AliasInfo       #Get-Command
Deserialized.System.ServiceProcess.ServiceController      #Get-Service
Deserialized.System.Management.Automation.AliasInfo       #Get-Command
Deserialized.System.Diagnostics.Process…                  #Get-Process
```

On constate ici que les trois cmdlets sont bien exécutés en parallèle et pas les uns à la suite des autres. Les données reçues seront donc entrelacées.

Dans un bloc *Parallel* l'ordre d'exécution des instructions est imprévisible.

Voir aussi.

4.2.1 Sequence

Ce mot clé force l'exécution séquentielle des activités qu'il englobe. On peut ainsi ordonner des groupes d'activités dans un bloc parallèle :

```
Workflow Test {
 Parallel {
    Get-Service
    Sequence {
      Get-Command
      Get-Process
   }
 }
}
Test|% {$_.PStypenames[0]}
```

```
…
Deserialized.System.ServiceProcess.ServiceController
Deserialized.System.ServiceProcess.ServiceController …
Deserialized.System.Management.Automation.CmdletInfo …
Deserialized.System.Diagnostics.Process
Deserialized.System.Diagnostics.Process
…
```

Exemple avec deux blocs *Sequence* :

```
Workflow Test {
 Parallel {
   Sequence {
     Get-Service
     Get-Process
   }
   Sequence {
     Get-Date
     Get-Command Get*
   }
 }
}
Test|% {$_.PStypenames[0]}
```

```
…
Deserialized.System.Diagnostics.Process
Deserialized.System.Management.Automation.FunctionInfo
Deserialized.System.Management.Automation.FunctionInfo
Deserialized.System.Diagnostics.Process
…
```

Voir aussi.

4.2.1 Foreach –Parallel

Dans un workflow l'instruction *Foreach* supporte les paramètres ***–ThrottleLimit*** et ***–Parallel***.

La version 3 autorise jusqu'à 5 threads maximum en parallèle, la version 4 propose le paramètre ***–ThrottleLimit*** pour contrôler le nombre de threads maximum.

L'itération des éléments de la collection se fait en parallèle, ici c'est l'intégralité du scriptblock qui est concerné par le parallélisme et toutes ces instructions sont exécutées séquentiellement.

Dans l'exemple suivant on exécute 10 threads en parallèle :

```
Workflow Test {
Param ([Int] $ThrottleLimit=10)
  ForEach -Parallel –ThrottleLimit $ThrottleLimit ($P in 1..10) {
    write-Warning "p=$p : $(get-date -Format 'mm:ss.ffff')"
    Start-Sleep –Seconds 5
  }
}
Test
```

```
AVERTISSEMENT : [localhost]:p=10 : 35:34.4004
…
AVERTISSEMENT : [localhost]:p=1 : 35:34.4472
```

Dans celui-ci on en exécute 3 en parallèle :

```
Test –ThrottleLimit 3
```

```
AVERTISSEMENT : [localhost]:p=3 : 36:32.7913
AVERTISSEMENT : [localhost]:p=2 : 36:32.7913
AVERTISSEMENT : [localhost]:p=1 : 36:32.7913

AVERTISSEMENT : [localhost]:p=4 : 36:37.8145
AVERTISSEMENT : [localhost]:p=5 : 36:37.8145
AVERTISSEMENT : [localhost]:p=6 : 36:37.8301
…
```

Note : le document de spécification de Powershell version 3 indique :

> Windows PowerShell: The *switch-parameter* `-parallel` is only allowed in a workflow (§8.10.2).

Mais ce paramètre n'est pas implémenté dans la v3, ni dans la v4.

4.3 Programme console interactive

Les programmes console suivants ne sont pas supportés explicitement :

"cmd","cmd.exe", "diskpart", "diskpart.exe", "edit.com", "netsh", "netsh.exe", "nslookup", "nslookup.exe", "powershell","powershell.exe"

Exemple :

```
Workflow Test { netsh.exe }
Test
```

```
Impossible de démarrer «netsh.exe». Les applications de console interactive ne sont pas prises en charge dans un workflow Windows PowerShell. Pour exécuter l'application, utilisez l'applet de commande Start-Process.
```

Restriction similaire à celle sous ISE, mais ici la variable *$psUnsupportedConsoleApplications* n'est pas implémentée.

Les programmes console qui ne sont pas interactif ne pose pas de problème :

```
Workflow Test { TaskList.exe /Fo CSV|convertfrom-csv }
```

La redirection de la sortie d'un programme console dans un fichier n'est pas prise en charge :

```
Workflow Test { TaskList.exe /Fo CSV > C:\Temp\ObjetsTask.csv }
test
```

```
Seule la redirection de fusion à partir du flux d'erreurs vers le flux de sortie est prise en charge.
```

```
Workflow Test {
 TaskList.exe /Fo CSV|Set-Content -Path c:\Temp\ ObjetsTask.csv
}
```

4.4 Usage du pipeline

Un workflow peut utiliser le pipeline, mais étant exécuté dans un job, il reçoit et émets des objets désérialisés. De plus les paramètres définis pour les fonctions de workflow ne prennent pas en charge l'attribut '*ValueFromPipeline*'.

La fonction associée au workflow propose bien le paramètre *–InputObject*, mais celui-ci n'existe plus dans le workflow :

```
Workflow Test {
   if ($InputObject -eq $null)
   {Write-warning 'InputObject est $Null' }
   Else
   {Write-warning "InputObject = $InputObject" }
}
Test –InputObject "MaDonnée"
```

```
AVERTISSEMENT : [localhost]:InputObject est $Null
```

```
"MaDonnée"|Test
```

```
AVERTISSEMENT : [localhost]:InputObject est $Null
```

Ce paramètre est supprimé de la liste des paramètres dans le bloc *End* de la fonction d'exécution du workflow. Seule la variable ***$Input*** permet de récupérer les données du pipeline :

```
Workflow Test {
  Write-warning "Input = $Input"
  Write-Output $Input
}
Test –InputObject "MaDonnée"
```

```
AVERTISSEMENT : [localhost]:Input = MaDonnée
```

```
"MaDonnée",1,2|Test
```

```
AVERTISSEMENT : [localhost]:Input = MaDonnée 1 2
```

Il reste dans ce cas un dernier point à régler. La variable automatique ***$Input*** est d'un type de collection particulier que l'on doit réinitialiser après une itération.

Si l'on souhaite réutiliser cette collection on doit mémoriser ces données, car l'appel à une méthode (*.Reset*) n'est pas supporté dans un workflow :

```
Workflow Test {
  $InputDatas=$Input|% {$_}
  Write-warning "Input=$InputDatas"
  Write-Output $InputDatas
}
Test –InputObject "MaDonnée"
```

```
AVERTISSEMENT : [localhost]:Input=MaDonnée
MaDonnée
```

```
"MaDonnée",1,2|Test
```

```
AVERTISSEMENT : [localhost]:Input = MaDonnée 1 2
MaDonnée
1
```

```
2
```

Il existe une autre possibilité pour connecter le pipeline à une activité qui est d'utiliser le paramètre commun d'activité nommé *-UseDefaultInput* :

```
Workflow Test{
  Write-warning "Step un"
  Get-Service -UseDefaultInput $True
  Write-warning "Step deux. `$Input=$input"
  Get-Process -UseDefaultInput $True
  Write-warning "Fin"
}
Test -InputObject "WinRM","Powershell"
```

On constate que seule la première activité lit les données du pipeline :

```
AVERTISSEMENT : [localhost]:Step un
Microsoft.PowerShell.Management\Get-Service : Impossible de trouver un service assorti du nom
« Powershell ».
Status      Name            DisplayName                                PSComputerName
------      ----            -----------                                --------------
Running     WinRM           Gestion à distance de Windows (Gest...     localhost
AVERTISSEMENT : [localhost]:Step deux. $Input=
AVERTISSEMENT : [localhost]:Fin
```

Je suppose qu'en interne la variable *$Input* et le pipeline pointe sur la même collection...

Voir également ce post.

4.5 Scope

Avant de poursuivre, revenons sur l'opération de transformation d'un code Powershell en un workflow.

4.5.1 Fonction

La génération du code XAML, et à la différence de Powershell, oblige à déclarer chaque fonction 'externe' avant de l'utiliser dans le code d'un workflow.

Ce qui fait que la construction suivante échoue :

```
Workflow Test {
  Fonction_Définie_Ultérieurement
}
Function Fonction_Définie_Ultérieurement {
 "Fait qq chose"}
```

```
La commande «Function_Définie_Ultérieurement» est introuvable.
Si cette commande est définie en tant que workflow, assurez-vous qu'elle est définie avant d'être appelée par le
workflow. …
```

On doit d'abord déclarer la fonction, puis le workflow l'utilisant :

```
Function Fonction_Définie_Antérieurement{
 "Fait qq chose"
```

```
}
Workflow Test {
  Fonction_Définie_Antérieurement
}
```

Dans ce cas le code de la fonction est inséré implicitement dans un appel à une activité **InlineScript** :

```
$c=Get-Command Test
$c.XamlDefinition -split "`r`n"|select-string 'InlineScript'
```

```
<ns1:InlineScript Command="trap { break }&#xD;&#xA;function
    Fonction_Définie_Antérieurement&#xD;&#xA;{&#xD;&#xA;&#xD;&#xA; &quot;Fait qq chose  …
```

Ce mécanisme n'inclue pas le code des fonctions imbriquées dépendantes :

```
Function Fonction_Imbriquée {
 "Fonction imbriquée"
}

Function Fonction_Définie_Antérieurement{
 "Fait qq chose"
  Fonction_Imbriquée
}
Workflow Test {
  Fonction_Définie_Antérieurement
}
```

Le workflow est déclaré sans erreur, mais son exécution échouera.

```
Test
```

```
Fait qq chose
Le terme «Fonction_Imbriquée» n'est pas reconnu comme nom d'applet de commande, fonction, fichier de
script ou programme exécutable. …
```

Une autre limite est qu'un workflow ne supporte pas les appels récursifs :

```
Workflow Test {
  Test
}
```

```
Un workflow ne peut pas utiliser la récursivité.
```

Une déclaration avancée de fonction est autorisée dans le code d'un workflow :

```
Workflow Test {
  Fonction_Définie_Ultérieurement
  Function Fonction_Définie_Ultérieurement {"Fait qq chose"}
}
```

Le code suivant ne référence pas la fonction, mais bien le workflow imbriqué :

```
Function Test-Imbrique {"[Fonction] "}
workflow Test
{
    Test-Imbrique
    Workflow Test-Imbrique {"[Workflow]"}
    Test-Imbrique
}
Test
```

```
[Workflow]
[Workflow]
```

4.5.2 Variable

PSWF ne supporte pas la portée dynamique de variables. C'est-à-dire qu'une variable définie dans une portée ne peut pas être redéfinie dans une portée enfant. De plus l'usage d'une variable déclarée dans la portée de l'appelant ne fonctionne pas :

```
$Parent='Externe'
Workflow Test { Write-Warning "Scope parent:$Parent"}
Test
```

```
AVERTISSEMENT : [localhost]:Scope parent:
```

Bien que son nom soit référencé dans le code XAML, elle n'est pas pour autant déclarée dans le code du workflow.

Nous avons vu qu'un workflow est exécuté dans un job, essayons donc le spécificateur de portée *Using* :

```
Workflow Test { Write-Warning "Scope parent: $Using:Parent"}
Test
```

```
Impossible de récupérer une variable Using. Une variable Using ne peut être utilisée qu'avec Invoke-Command, Start-Job ou InlineScript dans le workflow de script. …
```

Pour recevoir des données de l'appelant on déclarera un paramètre dans la section *Param().*

Bien que supporté dans un workflow, le spécificateur *$Using*: ne concerne pas un job, mais l'accès à une variable de workflow dans le code d'une activité **InlineScript** :

```
Workflow Test
{
    $a = 3
    Write-warning "Scope du workflow `$a=$a"

      # Sans $Using la variable de workflow $a n'est pas visible
    InlineScript {"Inline sans 'Using' = $a"}

    Write-warning "Scope du workflow `$a=$a"

     # $Using importe la variable et son contenu
```

```
    InlineScript {"Inline avec 'Using' = $Using:a"}
    $a = 3

    # Ici on utilise une copie de la variable
    InlineScript {$a = $using:a+1; "Inline modification de A = $a"}
    Write-warning "Scope du workflow `$a=$a"

    #Pour modifier la valeur de a dans un bloc inlinescript
    #on doit retourner la nouvelle valeur
    $a = InlineScript {$a = $Using:a+1; $a}
    Write-warning "Modification de la variable de workflow `$a=$a"

    $a = InlineScript {
     $Using:a+1
     "Retourne une nouvelle valeur, mais affiche l'ancienne A = $Using:a"
    }
    Write-warning "Fin `$a=$a"
}
```

```
AVERTISSEMENT : [localhost]:Scope du workflow $a=3
Inline sans 'Using' =
AVERTISSEMENT : [localhost]:Scope du workflow $a=3
Inline avec 'Using' = 3
Inline modification de A = 4
AVERTISSEMENT : [localhost]:Scope du workflow $a=3
AVERTISSEMENT : [localhost]:Modification de la variable de workflow $a=4
AVERTISSEMENT : [localhost]:Fin $a=5 Retourne une nouvelle valeur, mais affiche l'ancienne A = 4
```

4.5.3 Instructions créant une nouvelle portée

Nous avons vu qu'un workflow ne partage pas les variables de la session Powershell appelante. Les variables de workflow, c'est à dire celles déclarées dans le code du workflow, sont ajoutées à chaque nouvelle portée crée par une des instructions suivantes :

Boucles / Instructions de contrôle, Sequence, InlineScript, Parallel, Foreach -parallel

Par contre la modification d'une variable de workflow dans une nouvelle portée nécessitera l'usage du spécificateur *$Workflow :*

```
Workflow Test
{
  $Variable= "Initialisation"
  $Variable= "Modification"

  Sequence  #Nouvelle portée
  {
      #$Using:Variable #Impossible de récupérer une variable Using

      #$Variable= "Modification Impossible. Duplication de nom"
```

```
      $Workflow:Variable= "Modification possible."
  }
  Write-Warning "Fin Variable=$Variable"
}
```

Consultez le détail sur cette page : http://technet.microsoft.com/fr-fr/library/jj574187.aspx

4.5.4 Race condition

Dans la troisième édition du livre 'Windows PowerShell Cookbook', l'auteur Lee Holmes signale la possibilité de situation de compétition lors de l'accès aux variables de workflow dans un bloc *Parallel*.

Prenons l'exemple suivant :

```
Workflow Test {
 $result=0
 Parallel {
    $WORKFLOW:result=1
    $WORKFLOW:result=2
    $WORKFLOW:result=3
 }
 Write-Warning "Result=$Result"
}
```

```
AVERTISSEMENT : [localhost]:Result=3
```

Le résultat affiché est bien celui attendu, dans l'exemple suivant également :

```
Workflow Test {
 $Result=0
 Parallel {
    sequence { $WORKFLOW:result=$result+1;Write-Warning "sequence A
Result=$Result" }
    sequence { $WORKFLOW:result=$result+1;Write-Warning "sequence B
Result=$Result" }
    sequence { $WORKFLOW:result=$result+1;Write-Warning "sequence C
Result=$Result" }
 }
 Write-Warning "Result=$Result"
}
```

```
AVERTISSEMENT : [localhost]:sequence A Result=1
AVERTISSEMENT : [localhost]:sequence B Result=2
AVERTISSEMENT : [localhost]:sequence C Result=3
AVERTISSEMENT : [localhost]:Result=3
```

Mais si on implémente le second exemple de la page Wikipédia citée, le problème apparaît :

```
Workflow Test {
 $TotalPieces=0
```

```
 Parallel {
    sequence {
      $Accumulateur=$WORKFLOW:TotalPieces
      Write-Warning "sequence A Accumulateur=$Accumulateur"
      $Accumulateur=$Accumulateur+1
      $WORKFLOW:TotalPieces=$Accumulateur
      Write-Warning "--- sequence A TotalPieces=$TotalPieces"
     }
    sequence {
     $Accumulateur=$WORKFLOW:TotalPieces
     Write-Warning "sequence B Accumulateur=$Accumulateur"
     $Accumulateur=$Accumulateur+1
     $WORKFLOW:TotalPieces=$Accumulateur
     Write-Warning "--- sequence B TotalPieces=$TotalPieces"
    }
    #sequence { $Accumulateur=$WORKFLOW:TotalPieces..
 }
 Write-Warning "$TotalPieces=$TotalPieces"
}
```

```
AVERTISSEMENT : [localhost]:sequence A Accumulateur=0
AVERTISSEMENT : [localhost]:sequence B Accumulateur=0
AVERTISSEMENT : [localhost]:--- sequence A TotalPieces=1
AVERTISSEMENT : [localhost]:--- sequence B TotalPieces=1
AVERTISSEMENT : [localhost]:1=1
```

Chaque séquence lit la valeur initiale de la variable de Workflow, l'incrémente et lui réaffecte une valeur. Le résultat final est égal à 1 et pas à 2 comme on pourrait s'y attendre.

Essayons le même code avec une boucle Foreach –Parallel :

```
Workflow Test {
 $TotalPieces=0
 Foreach -Parallel ($I in 1..200) {
   $Accumulateur=$WORKFLOW:TotalPieces
   Write-Warning "$I Accumulateur=$Accumulateur"
   $Accumulateur=$Accumulateur+1
   $WORKFLOW:TotalPieces=$Accumulateur
   Write-Warning "--- $I TotalPieces=$TotalPieces"
 }
 Write-Warning "FIN TotalPieces=$TotalPieces"
}
```

```
…
AVERTISSEMENT : [localhost]:FIN TotalPieces=1
```

Dans ce cas on peut supposer que le nombre d'exécution en parallèle fausserait le résultat, il n'en est rien. Par contre si on limite le nombre d'exécution en parallèle avec -***ThrottleLimit*** :

```
Workflow Test {
 $TotalPieces=0
 Foreach -Parallel -ThrottleLimit 20 ($I in 1..200) {
…
```

```
…
AVERTISSEMENT : [localhost]:FIN TotalPieces=10
```

Le résultat final est différent, mais ne change rien au problème. On constate surtout que la valeur de la variable de workflow est mise à jour dix fois, c'est-à-dire à chaque exécution d'un lot *via* le paramètre ***ThrottleLimit***.

4.1 Logs

Les flux standards sont utilisables

```
Workflow TestFlux
{
   $PSParentActivityID='Tests des flux PS'
   Get-PSWorkflowData[Hashtable] -VariableToRetrieve All
   Write-Output "Output Logging"
   Write-Warning "Warning Logging"
   Write-Error "Error Logging"
   Write-Verbose "Verbose Logging"
   Write-Debug "Debug Logging"
}
$VerbosePreference='continue'
$DebugPreference='continue'
TestFlux
```

```
Output Logging
AVERTISSEMENT : [localhost]:Warning Logging
Microsoft.PowerShell.Utility\Write-Error : Error Logging
COMMENTAIRES : [localhost]:Verbose Logging
DÉBOGUER : [localhost]:Debug Logging
```

La redirection fonctionne :

```
$Filename='C:\Temp\Flux.txt'
TestFlux -PSComputerName $env:COMPUTERNAME,LocalHost *>$Filename
```

Dans ce cas chaque ligne du fichier contient un guid, identique à la variable *$JobInstanceId*, et le nom du host :

```
b566a1fa-3166-49f5-8636-b9195207a62a:[ComputerName]:Verbose Logging
796ecc32-c209-4ccc-946e-76e81c7a0a19:[LocalHost]:Verbose Logging
```

On peut utiliser la variable de préférence nommée *$PSParentActivityID*, celle-ci permet d'identifier de manière unique chaque instance du flux de travail.

Le paramètre commun d'activité *–DisplayName* permet de préciser un libellé lors de l'affichage de la progression d'exécution :

```
Workflow Test {
  start-sleep -seconds 5 -DisplayName "Recherche le process Powershell"
  $p=Get-process  -DisplayName "Recherche le process Powershell"
  $f=dir -path c:\temp\t*.txt -DisplayName "Recherche les fichier texte"
  start-sleep -seconds 5 -DisplayName "Recherche les JOB "
  get-Job -DisplayName "Récupére les job " -name win*
 }
```

La redirection des flux ne concerne pas l'affichage de ***Write-Progress***.

Note : Le cmdlet Get-Service possédant déjà un paramètre de même nom, il n'est pas possible d'utiliser le paramètre commun de même nom.

4.1.1 Traces de debug ETW

On peut utiliser le mécanisme de trace de Powershell (ETW) à l'aide de la méthode statique *GetTraceSource()* de la classe suivante **PowerShellTraceSourceFactory**.

Celle-ci renvoi une instance permettant d'écrire des traces via le provider *'Microsoft-Windows-PowerShell'* :

```
Logman query providers|? { $_ -match 'powershell'}
```

```
Microsoft-Windows-PowerShell                {A0C1853B-5C40-4B15-8766-3CF1C58F985A}
Microsoft-Windows-PowerShell-DesiredStateConfiguration-FileDownloadManager {AAF67066-0BF8 …
```

L'instance récupérée doit être explicitement libérée une fois l'appel effectué :

```
Workflow Test {
  InlineScript {
   Try {

$Tracer=[System.Management.Automation.Tracing.PowerShellTraceSourceFactory
]::GetTraceSource()
     [void]$Tracer.WriteMessage("[WF_Test] BEGIN Message de Thrace")
   }
   Finally {
    If ($Tracer -ne $Null)
    { $Tracer.Dispose() }
   }
  }
  $p=Get-process  -DisplayName "Recherche le process Powershell"
}
```

On doit au préalable charger le module PSDiagnostics puis activer les traces à l'aide du cmdlet **Enable-PSWSManCombinedTrace** :

```
  Import-Module PSDiagnostics
  Enable-PSWSManCombinedTrace
```

Ensuite on exécute le workflow, puis on désactive les traces et enfin on récupère le résultat :

```
  Test
  Disable-PSWSManCombinedTrace
  $Global:Result = Get-WinEvent -Path $pshome\Traces\PSTrace.etl -Oldest
  $Global:Result|Where-Object {($_.id -eq 45061) -and ($_.Message -match '
\[WF_Test]')}|Fl Message
}
```

```
Message : Informations de trace :
        [WF_Test] BEGIN Message de Thrace
```

On peut également utiliser une fonction externe au Workflow :

```
Function Trace {
 Param([String] $Message)
 Try {

$Tracer=[System.Management.Automation.Tracing.PowerShellTraceSourceFactory
]::GetTraceSource()
   [void]$Tracer.WriteMessage($Message)
 }
 Finally {
  If ($Tracer -ne $Null)
  { $Tracer.Dispose() }
 }
}


Workflow Test {
  Trace "[WF_Test] BEGIN Message de Thrace"
  $p=Get-process -DisplayName "Recherche le process Powershell"
  Trace "[WF_Test] END Message de Thrace"
}
```

Ceci a un coût, car chaque appel à la fonction *Trace* génère un appel à l'activité **InlineScript** :

```
(gcm Test).XamlDefinition -split "`r`n"|select-string 'InlineScript'
```

4.2 Gestion d'erreur

Sous Powershell version 4, le paramètre commun ***-ErrorAction*** accepte la nouvelle valeur "*Suspend*". Celle-ci doit être précisée uniquement lors de l'exécution d'un workflow :

```
Un_Workflow –ErrorAction Suspend
```

Tout autre usage provoquera une exception. Le code du workflow doit également paramétrer la variable de préférence *$ErrorActionPreference* à '*Stop*'. Ainsi lors du déclenchement d'une

erreur le job du workflow sera suspendu, laissant la possibilité d'analyser la cause de cette erreur :

```
workflow Test {
 $ErrorActionPreference='Stop'

 Get-ChildItem –Path A:\

 Get-ChildItem –Path NotExist:\

 Throw "Exception WF Test"    #--- Commentez cette ligne
 Write-Error "Erreur WF Test"
 "Suite"
}
$Job=Test –ErrorAction Suspend -JobName TestSuspend
```

Le premier appel à *Get-ChildItem* sur un lecteur inexistant génère une erreur :

```
L'exécution de la commande s'est arrêtée, car la variable de préférence «ErrorActionPreference» ou le paramètre  courant a la valeur Stop: Lecteur introuvable. Il n'existe aucun lecteur nommé «A».
```

On reçoit donc une instance de job qui est dans l'état suspendu :

```
$Job

Id     Name      PSJobTypeName   State         HasMoreData     Location        Command
--     ----      -------------   -----         -----------     --------        -------
20     Job20     PSWorkflowJob   Suspended     True            localhost       Test
```

Ce job suspendu est enregistré localement dans un répertoire dédié, il est donc possible de rebooter la machine puis de relancer le job de workflow en appelant *Resume-Job*.

Son paramétrage initial restant identique, le job peut de nouveau être suspendu lors la génération d'une nouvelle erreur :

```
Get-Job -Name TestSuspend |Resume-Job

Id     Name      PSJobTypeName   State         HasMoreData     Location        Command
--     ----      -------------   -----         -----------     --------        -------
20     Job20     PSWorkflowJob   Running       True            localhost       Test
```

Désormais, on gère un job de workflow :

```
$Job

Id     Name      PSJobTypeName   State         HasMoreData     Location        Command
--     ----      -------------   -----         -----------     --------        -------
20     Job20     PSWorkflowJob   Suspended     True            localhost       Test
$job|Receive-Job –Keep

Receive-Job : L'exécution de la commande s'est arrêtée, car la variable de préférence «ErrorActionPreference» ou le paramètre courant a la valeur Stop: Lecteur introuvable. Il n'existe aucun lecteur nommé «NotExist».
$Job|Resume-Job

Id     Name      PSJobTypeName   State         HasMoreData     Location        Command
--     ----      -------------   -----         -----------     --------        -------
20     Job20     PSWorkflowJob   Running       True            localhost       Test
```

Le déclenchement d'une exception placera le job de workflow dans l'état '*Failed' :*

```
$Job
```

```
Id     Name      PSJobTypeName   State         HasMoreData     Location        Command
--     ----      -------------   -----         -----------     --------        -------
20     Job20     PSWorkflowJob   Failed        True            localhost       Test
```

```
$Job|Receive-Job
```

```
Receive-Job : L'exécution de la commande s'est arrêtée, car la variable de préférence «ErrorActionPreference»
ou le paramètre courant a la valeur Stop: Lecteur introuvable. Il n'existe aucun lecteur nommé «NotExist».

Exception WF Test
```

```
$Job|Remove-Job
```

Notez que la première erreur déclenchant la suspension du workflow est consommée et placée dans la collection $Error.

Pour des workflows imbriqués seul le premier appel peut préciser *–ErrorAction Suspend* et chaque workflow doit affecter la valeur *Stop* à la variable ***$ErrorActionPreference*** :

```
workflow Inner {
  $ErrorActionPreference='Stop'
  Write-error "Erreur dans inner."
  "Reprise dans inner"
}

workflow Test {
Try {
  $ErrorActionPreference='Stop'
  Write-error "Erreur dans Test."
  Inner  #-ErrorAction Suspend

  "Suite dans Test"
 } Catch {
   $errorMessage = "Try:Test $_"
 }
 $errorMessage
}
Test -ErrorAction Suspend
```

Dans ce dernier exemple le bloc ***Catch*** n'est jamais exécuté, car le mécanisme de suspension du job intercepte l'exception.

Vous pouvez également consulter les usages des paramètres communs de workflow *PSError* et *MergeErrorToOutput.*

4.1 Divers

4.1.1 Modification de paramètres communs

Les paramètres communs d'un workflow sont en lecture seule :

```
Workflow Test {
  Write-Warning "PSComputerName=$PSComputerName"
  $PSComputerName =$PSComputerName+'LocalHost'
  dir -path $test
}
```

La valeur de la variable « PSComputerName » ne peut être modifiée qu'en utilisant l'activité Set-PSWorkflowData.

Pour les modifier on utilisera l'activité Set-PSWorkflowData qui n'est pas documentée dans Powershell :

```
Workflow Test {
  Set-PSWorkflowData –OtherVariableName 'ErrorAction' –Value "Stop"

  $before = Get-PSWorkflowData[Hashtable] -VariableToRetrieve All
  Write-Warning "Before Setting: $($before.PSComputerName)"
    #------ Excécute l'activité sur $env:COMPUTERNAME
   Dir –Path C:\Temp|Select-Object –First 1
    #------Variable temporaire
  $ComputersName=$before.PSComputerName
  $ComputersName +="LocalHost"
  Set-PSWorkflowData -PSComputerName $ComputersName
  $after = Get-PSWorkflowData[string[]] -VariableToRetrieve PSComputerName
$ofs=' ; '
  Write-Warning "After Setting: $after"
   #------ Excécute l'activité sur $env:COMPUTERNAME et LocalHost
   Dir –Path C:\Temp|Select-Object –First 1
}
Test -PSComputerName $env:COMPUTERNAME
```

L'activité générique Get-PSWorkflowData récupère les paramètres communs, le type indiqué correspondant au type du résultat renvoyé.

Une fois le paramètre commun ***PSComputerName*** modifié, chaque activité suivante utilisera son nouveau contenu.

4.1.1.1 PSParameterCollection

Le paramètre commun de workflow *PSParameterCollection* autorise une configuration spécifique pour chaque machine ciblée par le workflow :

```
$ArrayOfHashtable=@(
```

```
 @{
    PSComputerName="Server01";
    PSElapsedTimeoutSec=10;
    PSConnectionRetryCount=3
 },
 @{
   PSComputerName="$Env:ComputerName"
   PSElapsedTimeoutSec=20;
 },
 @{
   PSComputerName="*"
   PSElapsedTimeoutSec=30;
 }
) #$ArrayOfHashtable

MyWorkflow -PSParameterCollection $ArrayOfHashtable …
```

4.1.2 #Requires

Les clauses ***#Requires*** sont analysées par le parseur, mais inappliqués lors de l'exécution, hormis la syntaxe avec *–Assembly*.

```
Workflow Test {
    #--> Aucune erreur
  #requires -Version 6.0
     #--> erreur : -1 est invalide
  ##requires -Version -1.0
 #Requires -PSSnapin NotExistSnapin -Version 1.2
  Get-ChildItem –Path C:\
}
Test
```

4.1.3 Chemin

La documentation indique que tous les disques ajoutés par les providers de base de Windows PowerShell le sont également dans les workflows. Les providers de base sont : *Alias, Certificate, Environment, FileSystem, Function, Registry, Variable* et *WS-Management*.

Si l'import d'un module imbrique le chargement d'un provider spécifique suivi de la création de lecteurs, on peut utiliser le paramètre d'activité *PSRequiredModules* ou utiliser une commande Import-Module dans une activité InlineScript. Les exemples suivant utilisent ces techniques pour accéder au provider ***IIS*** :

```
workflow Get-IIS {
 dir -Path IIS: -PSRequiredModules WebAdministration
```

```
}

workflow Get-IIS {
 InlineScript { Import-Module WebAdministration; dir IIS: }
}
```

Ceci fonctionne avec PSCX :

```
Workflow Test { Dir Feed:\Powershell\win* -PSRequiredModules PSCX }
```

La localisation courante peut influencer l'exécution d'un workflow :

```
Workflow Test {  Get-ChildItem FileSystem::\\Localhost\C$ }
cd Feed:\Powershell
Test
```

```
Lecteur introuvable. Il n'existe aucun lecteur nommé « Feed ».
```

Ici ce n'est pas l'appel à l'activité **Get-ChildItem** qui pose problème :

```
cd Cert:
Test
```

```
#ok…
```

C'est le répertoire courant qui pointe sur un drive d'un provider qui n'est pas un des providers de base de Windows PowerShell.

4.1.1 Help

Un workflow ne supporte pas les balises d'aide dans le corps du script. Vous devez utiliser un fichier XML, L'outil PowerShell Cmdlet Help Editor vous aidera à le construire.

4.2 Valider la transformation d'une fonction en un workflow

On peut à l'aide d'API valider la transformation d'une fonction en un workflow. Prenons une fonction simple, mais sans intérêt si ce n'est pour cette démonstration :

```
Function Test {
  param (
         [Parameter(Mandatory=$true,ValueFromPipeline = $true)]
        [String] $Name
  )
  $Name.Substring(1,5)
  Write-warning "Name=$Name"
  . .\DotSource.ps1
}#Test
```

Cette validation nécessite deux étapes la première étant de rechercher le code incompatible :

```
ipmo PSWorkflow
$Converter=New-Object Microsoft.PowerShell.Workflow.AstToWorkflowConverter
$F=Get-Command Test
$Converter.ValidateAst($F.ScriptBlock.ast)
```

```
Extent                 ErrorId                          Message                      IncompleteInput
------                 -------                          -------                      ---------------
$Name.Substring(1,5)   MethodInvocationNotSupported     L'appel des méthodes n'est...  False
. .\DotSource.ps1      AlternateInvocationNotSupp...    L'appel de source de type ...  False
```

Une fois le code adapté :

```
Function Test {
  param (
         [Parameter(Mandatory=$true,ValueFromPipeline = $true)]
        [String] $Name
  )
  Write-warning "Name=$Name"
}#Test
```

Une seconde vérification s'impose :

```
$F=Get-Command Test
$Converter.ValidateAst($F.ScriptBlock.ast)
```

```
#ras
```

Le code ne posant à priori plus de problème, on passe à la seconde étape qui est la compilation de ce code en un workflow :

```
$wfName='Test'
$FunctionDefinition=$f.Definition
$initialsessionstate=
[System.Management.Automation.Runspaces.Runspace]::DefaultRunspace.InitialSessionState
$WF=$o.CompileWorkflow($wfName,$FunctionDefinition,$initialsessionstate)
```

```
Exception lors de l'appel de «CompileWorkflow» avec «3» argument(s): «Les paramètres définis pour les fonctions de workflow ne prennent pas en charge ValueFromPipeline. Supprimez cet attribut et réimportez ou redéfinissez le workflow.
```

Une fois la déclaration de l'attribut adaptée, une nouvelle tentative de compilation réussie. La méthode ***CompileWorkflow()*** renvoie un objet de type *WorkflowInfo* :

```
$F=Get-Command Test
$Converter.ValidateAst($F.ScriptBlock.ast)
$FunctionDefinition=$f.Definition
$WF=$o.CompileWorkflow($wfName,$FunctionDefinition,$initialsessionstate)
$WF
```

```
CommandType     Name                                ModuleName
-----------     ----                                ----------
Workflow        Test
```

Notez que la fonction d'origine existe toujours dans le provider de fonction.

5 Créer un module de workflow

Un workflow peut être déclaré dans un module comme une fonction ou *via* un fichier .XAML.

5.1 Basé Powershell

Reprenons l'exemple de workflow appelant un autre workflow :

```
@'
Workflow Nested {
  Param($Parametre)
 Write-output "Workflow imbriqué"
}
Workflow Test {
  Param($Parametre)
  Nested
}
'@ > C:\temp\TestWF.psm1
Import-module –Name C:\temp\TestWF.psm1
Test
```

L'implémentation est identique, c'est-à-dire qu'une DLL est créée et verrouillée. L'appel à **Remove-Module** ne changera rien à ce verrouillage.

5.2 Basé .xaml

Créons un workflow simple basé sur un fichier XAML :

```
Workflow Test {
  Dir -path 'C:\Temp'
}
$Cmd=Get-Command Test
$Cmd.XamlDefinition > C:\Temp\TestWF.xaml
Del Function:Test
Ipmo -Name C:\Temp\TestWF.xaml -PassThru
```

```
ModuleType Version    Name        ExportedCommands
---------- -------    ----        ----------------
Workflow   0.0        TestWF      TestWF
```

Notez que le nom de la commande exportée est identique au nom de fichier .xaml.

Le code suivant provient de cette page d'exemples. Ici on doit d'abord charger un assembly référencé dans le code .xaml créé sous Visual Studio :

```
Cd 'Votre répertoire\Source\ModuleXAML'
Add-Type -path "$pwd\InvokeMethodUsage.dll"
Import-Module "$pwd\Sequence1.xaml"
Sequence1
```

Le nom de la dll doit correspondre au nom indiqué dans la balise suivante :

```
<Activity mc:Ignorable="sap"
   x:Class="Microsoft.Samples.InvokeMethodUsage.Sequence1"
```

```
    xmlns:msi="clr-namespace:Microsoft.Samples.InvokeMethodUsage;assembly=InvokeMethodUsage"
…
```

Sinon l'exécution du workflow échouera :

```
Impossible de démarrer le workflow «Sequence1» : Impossible de créer le type inconnu
'{http://schemas.microsoft.com/netfx/2009/xaml/activities}Variable(
{clr-namespace:Microsoft.Samples.InvokeMethodUsage;assembly=InvokeMethodUsage}TestClass)'.
```

Le programme console *InvokeMethodUsage.exe*, présent dans le même répertoire, correspond au projet CSharp compilé.

5.3 Invoke-AsWorkflow

Le module **PSWorkflowUtility** héberge une fonction nommée *Invoke-AsWorkflow*, celle-ci exécute, dans un bloc InlineScript au sein d'un workflow, soit une commande soit du code Powershell :

```
ipmo PSWorkflowUtility
  #Exécute une expression
Invoke-AsWorkflow -Expression "(Get-ChildItem C:\Temp\*.txt).Name|
PSCX\Join-String -Separator '-*-'"


  #Exécute un cmdlet. Affiche les drives de base
Invoke-AsWorkflow -Command Get-PSDrive


  #La modification du path ne se fait pas dans le sessio ncourante
  # mais dans une autre session.
Invoke-AsWorkflow -Command Set-location -Parameter @{Path='C:\windows'}
```

La commande doit impérativement exister, par exemple une fonction ne peut être utilisée :

```
Function TestFonction { Get-ChildItem C:\Temp\*.txt}
  #Exécute une fonction.
Invoke-AsWorkflow -Command TestFonction -Parameter @{Name='Invoke As
Workflow'}
```

```
Le terme «TestFonction» n'est pas reconnu comme nom d'applet de commande, fonction, fichier de script ou
programme exécutable.
```

En revanche l'exécution d'un script ne pose pas de problème :

```
$Fichier='C:\Temp\TestFonction.ps1'
@"
 $((Dir Function:TestFonction).Definition)
"@ > $Fichier
Invoke-AsWorkflow -Command $Fichier -Parameter @{Name='Invoke As
Workflow'}
```

```
   Répertoire : C:\Temp
Mode    LastWriteTime         Length   Name        PSComputerName
----    -------------         ------   ----        --------------
-a---   23/10/2014   15:46    15       datas.txt   localhost …
```

6 Persistance

Un des points fort de Windows Workflow Foundation est de rendre persistant un workflow en sauvegardant l'ensemble de son état et de ses données. Cette sauvegarde est réalisée sur la machine exécutant le workflow.

Son objectif est de reprendre le traitement d'un workflow là où il s'est arrêté. Cet arrêt pouvant être ou non volontaire.

En contrepartie cette fonctionnalité à un coût et nécessite d'étudier quand et où déclencher cette persistance, car on ne l'implémente pas 'au petit bonheur la chance' !

6.1 Répertoire de persistance

Par défaut le répertoire de sauvegarde est le suivant :

$env:LocalAppData\Microsoft\Windows\PowerShell\WF\PS

La taille maximum de ce chemin d'accès est de 120 caractères :

```
$Path="$env:LocalAppData\Microsoft\Windows\PowerShell\WF\PS"
New-PSWorkflowExecutionOption –PersistencePath "$Path\$('1'*100)"
```

```
New-PSWorkflowExecutionOption : Impossible de lier le paramètre «PersistencePath» à la cible.
Exception lors de la définition de «PersistencePath» : «La valeur du paramètre PersistencePath C:\Users\....
dépasse la longueur maximale autorisée du chemin d'accès de persistance ne devant pas être supérieure à 120
caractères.»
```

Le nommage du répertoire pour un compte administrateur est le suivant :

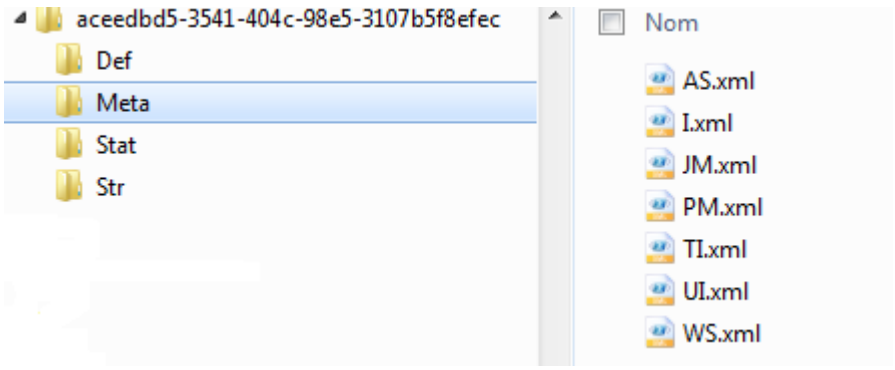
```
$User=gwmi Win32_UserAccount -Filter "Name='$Env:UserName'"
$PSWFOptions=New-PSWorkflowExecutionOption
Dir "$($PSWFOptions.PersistencePath)\Default\$($User.SID)_EL"
```

L'extension '_EL' pouvant être, dans d'autres contextes, remplacé/complété par '_CP ou '_NI'.

Je suppose que **'_EL'** est pour *Elevated*, **'_NI'** pour *non-interactive* et **'_CP'** semble indiquer un contexte de délégation CredSSP…

6.1.1 Contenu sommaire du répertoire de sauvegarde

Ce répertoire contient un sous-répertoire par workflow et celui-ci héberge les données sauvegardées :



Voici, issues du code source *via* ILSpy, quelques informations sur les préfixes utilisés :

```
// Microsoft.PowerShell.Workflow.PSWorkflowFileInstanceStore
private readonly string Streams = "Str";
private readonly string Error = "Err";
private readonly string Metadatas = "Meta";
private readonly string Definition = "Def";
private readonly string WorkflowState = "Stat";
private readonly string Version_xml = "V.xml";
private readonly string InputStream_xml

…
```

6.2 Point de contrôle (Checkpoint-Workflow)

Nous avons vu que la persistance peut être mise en œuvre lors du déclenchement d'une erreur. Cette fonctionnalité crée implicitement un point de contrôle puis suspend le job du workflow.

Il est possible d'implémenter cette fonctionnalité à l'aide des mots-clés **Checkpoint-Workflow** et **Suspend-Workflow.** Le premier enregistre un point de contrôle (la notion de bookmark sous WF) en écrasant le dernier sauvegardé s'il existe.

Un workflow Powershell ne peut donc être relancé qu'à partir du dernier point de contrôle enregistré. Ce mécanisme est similaire à un snapshot de VM ou à une sauvegarde complète d'un disque dur.

L'exemple suivant crée un point de contrôle :

```
Workflow Test
{
    Write-Warning "Recherche des process"
    $Name='Po*'
    Get-Process -Name $Name

 #---- Enregistre l'état du workflow
    Checkpoint-Workflow
    Write-Warning "Suite du Workflow. Name='$Name'"
    Start-Sleep -Seconds 40
    Get-service -Name W*
}
Test –JobName TestReprise
```

Comme aucune erreur interne (exception) ou externe (coupure réseau) n'interrompt son exécution, le point de contrôle est automatiquement supprimé une fois l'exécution du workflow terminé correctement. Vous pouvez le constater en visualisant le répertoire de sauvegarde dans l'explorateur de fichiers.

Un point de contrôle persiste sur le disque uniquement en cas d'erreur ou de suspension de son exécution. Si on tue le processus Powershell lorsque le workflow est en attente :

```
AVERTISSEMENT : [localhost]:Recherche des process
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  ProcessName PSComputerName
-------  ------    -----      ----- -----   ------     --  ----------- --------------
    198      27    17024      31792   186    24,79   5416 PowerShell  localhost
AVERTISSEMENT : [localhost]:Suite du Workflow. Name='Po*'
```

Il suffira d'ouvrir une nouvelle session et de relancer le job, celui-ci reprend son exécution à partir du dernier point de contrôle enregistré :

```
Get-Job –Name TestReprise
Id     Name            PSJobTypeName   State         HasMoreData     Location        Command
--     ----            -------------   -----         -----------     --------        -------
3      TestReprise     PSWorkflowJob   Suspended     True            localhost       Test
```

```
Get-Job –Name TestReprise|Resume-Job|Receive-Job -Wait
AVERTISSEMENT : [localhost]:Suite du Workflow. Name='Po*'
Status      Name            DisplayName            PSComputerName
------      ----            -----------            --------------
Stopped     W32Time         Temps Windows          localhost
…
```

Notez que le contenu de la variable *$Name* est restaurée. On retrouve donc le workflow et ses données dans l'état où il était lors de l'exécution du dernier checkpoint avant son arrêt inopiné.

Le déclenchement de l'interruption de l'exécution du workflow se fait ainsi :

```
    # Enregistre l'état du workflow puis suspend l'exécution
    Suspend-Workflow
    Write-Warning "Suite du Workflow. Name='$Name'"
```

Le script complet se trouve dans le fichier *\Source\TestSuspend.ps1*.

On peut utiliser le mot clé *Persist* ou *PSPersist* en lieu et place de **Checkpoint-Workflow**. Chacun de ces mots-clés est transformé en un appel à l'activité PSPersist.

Consultez également ce chapitre Where to place checkpoints.

6.1 Interruption temporaire (Suspend-Workflow)

L'appel à **Suspend-Workflow** crée un point de contrôle (*Checkpoint)* avant de suspendre l'exécution du workflow**.** Cette instruction renvoi une instance d'un job (PSWorkflowJob), même si le paramètre *–AsJob* n'est pas précisé.

Le cycle arrêt/reprise permet de combiner des activités automatiques et manuelles.

6.1.1 Suspendre un job de workflow (Suspend-Job)

Il est possible à l'appelant de suspendre un workflow à partir du moment où il est exécuté en tant que job :

```
Workflow Test
{
    Get-Process
    Get-ChildItem c:\windows\System32
    Get-service
}
$Job=Test –JobName TestReprise –AsJob
$Job |Suspend-Job
$Job |receive-job -Wait
$Job
Get-Job|Remove-Job;Cls
```

Pour cet exemple, comme le workflow ne déclare pas de point de contrôle, son comportement est erratique. On peut continuer à appeler **Receive-Job** jusqu'à ce que le job passe dans l'état *'Completed'*, sans pour autant exécuter **Resume-Job** !

La documentation de **Suspend-Job** précise : *"If the workflow job does not have checkpoints, it cannot be suspended properly."*

Une solution dans ce cas est d'utiliser le paramètre *–PsPersist* :

```
$Job=Test -JobName TestReprise -AsJob -PSPersist $True
$Job|Suspend-Job

 # 1- Process
$Job |Receive-Job -Wait
Resume-Job $job
$Job |Suspend-Job

 # 2- Files
$Job |Receive-Job -Wait
Resume-Job $job
```

```
 # 3- Service
$Job |Receive-Job -Wait
$Job
#Completed
```

Ainsi l'état est sauvegardé après l'exécution de chaque activité, même si le code ne déclare pas d'appel à **Checkpoint-Workflow**. «*Cette dernière option a une incidence sur les performances ; elle doit donc être utilisée avec parcimonie et en cas de besoin uniquement*».

Le paramètre *–PsPersist* laisse à l'utilisateur du workflow le choix de la persistance. Il ne modifie pas le comportement des points de contrôle existant dans le code.

Il est préférable de prévoir un point de contrôle explicite.

Pour un résultat similaire, une autre possibilité est d'ajouter le paramètre **–PSPersist:$true** à une activité ou encore d'utiliser dans le code du workflow la variable de préférence ***$PSPersistPreference***. Si vous lui affectez la valeur *$True*, chaque activité suivant cette affectation sera persistante, ainsi on peut choisir de rendre persistant un groupe d'activités et pas l'ensemble des activités du workflow. L'affectation de la valeur *$False* arrêtera ce comportement.

Enfin si le code du workflow contient un ou plusieurs checkpoints, alors **Suspend-Job** suspend le job après l'exécution du premier point de contrôle atteint :

```
Workflow Test
{
    Get-Process
   Checkpoint-Workflow
    Get-service
}
$Job=Test –JobName TestReprise –AsJob
$Job |Suspend-Job
$Job |Receive-Job -Wait
$Job
$Job |Resume-Job
$Job |Receive-Job -Wait
$Job
```

6.1.2 SerializationDepth

En passant, le problème de profondeur de sérialisation des données, abordé dans ce tutoriel, demeure. Le cmdlet **Update-TypeData** n'est pas supporté dans un workflow, en revanche le paramètre d'activité *–PSDisableSerialization* règle, en local, ce problème :

```
Workflow Depth {
 #Update-TypeData -TypeName System.Diagnostics.Process -SerializationDepth 2 -Force

 $Objet1=New-Object PSobject -property @{
            Level1=1..5
            Level2=@((6..10),(11..15))
            Level3=@(@((16..20),(21..25)),@((26..30)))
         }
 $Objet2=New-Object PSobject -property @{
            Nested=$Objet1
            Process=(Get-Process -id $pid)
         }

 Write-output $Objet2 -PSDisableSerialization $True
}
$Result=Depth
$Result.Nested.Level3[0][0][0]
$Result.Process.StartInfo
```

Dans ce cas l'objet *$Result.Process* n'est plus sérialisé, on manipule une instance de processus.

Autre possibilité *via* la variable de préférence nommée ***$PSDisableSerializationPreference*** de type booléen qui n'est pas encore documentée :

```
Workflow Depth {
 $PSDisableSerializationPreference=$True

 $Objet1=New-Object PSobject -property @{
            Level1=1..5
            Level2=@((6..10),(11..15))
            Level3=@(@((16..20),(21..25)),@((26..30)))
         }
 $Objet2=New-Object PSobject -property @{
            Nested=$Objet1
            Process=(Get-Process -id $pid)
         }
Write-output $Objet2
}
```

```
$Result=Depth
$Result.Nested.Level3[0][0][0]
$Result.Process.StartInfo
```

6.2 Implémenter un compteur de reprise

On peut utiliser la persistance pour gérer un compteur de reprise, le script d'exemple '*\Source\CompteurDeReprise.ps1*' teste trois fois l'existence d'un répertoire avant de terminer le job.

Etant donné que les instructions **Break** et **Continue** ne sont pas supportées dans une boucle ***Foreach*** on utilise une boucle ***While***.

Exécutons ce workflow :

```
$VerbosePreference='Continue'
CompteurDeReprise -JobName Reprises
```

```
COMMENTAIRES  : [localhost]:Tentative numéro : 1/3
Microsoft.PowerShell.Utility\Write-Error : Le répertoire 'C:\Temp\WF1' n'existe pas. Corrigez ce point.
Id     Name            PSJobTypeName   State         HasMoreData     Location             Command
--     ----            -------------   -----         -----------     --------             -------
56     Reprises        PSWorkflowJob   Suspended     False           localhost            CompteurDeReprise
```

Première relance :

```
Get-Job -Name Reprises|Resume-Job|Receive-Job -wait
```

```
COMMENTAIRES  : [localhost]:Tentative numéro : 2/3
Microsoft.PowerShell.Utility\Write-Error : Le répertoire 'C:\Temp\WF1' n'existe pas. Corrigez ce point.
Id     Name            PSJobTypeName   State         HasMoreData     Location             Command
--     ----            -------------   -----         -----------     --------             -------
56     Reprises        PSWorkflowJob   Suspended     False           localhost            CompteurDeReprise
```

Seconde relance (elle débute sur la même ligne, mais pas dans le même état) :

```
Get-Job -Name Reprises|Resume-Job|Receive-Job –wait
```

La troisième tentative termine le job de workflow, celui-ci passe dans l'état '*Completed*' :

```
COMMENTAIRES  : [localhost]:Tentative numéro : 3/3
Microsoft.PowerShell.Utility\Write-Error : Le répertoire 'C:\Temp\WF1' n'existe pas. Corrigez ce point
AVERTISSEMENT : [localhost]:Echec. Nombre de reprise maximum atteint.
Get-Job -Name Reprises
Id     Name            PSJobTypeName   State         HasMoreData     Location             Command
--     ----            -------------   -----         -----------     --------             -------
56     Reprises        PSWorkflowJob   Completed     False           localhost            CompteurDeReprise
```

Pour terminer le workflow, au lieu d'un appel à **Write-Warning** suivi de l'instruction *Exit*, on peut également envisager de déclencher une exception, à vous de voir.

Comme indiqué dans ce tutoriel, si vous utilisez plusieurs cibles vous n'aurez qu'un seul job contenant plusieurs jobs enfants :

```
$Computers='LocalHost',"$Env:Computername"
$Job=CompteurDeReprise -JobName Reprises -PSComputerName $Computers
$Job.ChildJobs
```

On peut reprendre un seul job enfant suspendu :

```
Resume-Job $Job.ChildJobs[0]
$Job|Receive-Job
```

Le job parent reste dans l'état '*Suspended*' tant qu'un des jobs enfant est suspendu.

6.3 Workflow imbriqué

La suppression à intervalle régulier du contenu du répertoire *PSWorkflowCompilation*, n'impacte pas les jobs en attente de reprise. Ceux en cours d'exécution ou ceux exécutés verrouillent leur dll.

Prenons l'exemple suivant :

```
Workflow Test {
 Write-Warning "Suspend"
  Suspend-Workflow;
 Write-Warning "Reprise"
}
Workflow Imbrication {
 Write-Warning "Debut "
 Test
 Write-Warning "Fin"
}
Imbrication
```

Si on termine la session une fois ce workflow exécuté, le job associé reste en attente de reprise, entre temps il est possible de supprimer le répertoire contenant la dll du workflow imbriqué :

```
Del "$env:LOCALAPPDATA\Temp\PSWorkflowCompilation"
```

Ce scénario est prévu, car la dll du workflow imbriqué liée au workflow 'parent' est recopiée dans le répertoire de sauvegarde du parent. Si dans ce répertoire on renomme ou supprime cette dll, alors **Resume-Job** déclenchera l'erreur suivante :

```
Get-Job|Resume-Job
```

```
Resume-Job : Impossible de reprendre la tâche de workflow, soit parce que les données de persistance n'ont pas pu être enregistrées en totalité, soit parce que les données de persistance enregistrées ont été endommagées. Vous devez redémarrer le workflow.
```

Le job est désormais dans l'état 'Failed'. Il ne reste plus qu'à le supprimer :

```
Get-Job|Remove-Job
```

Lorsque vous exécutez un workflow en tant que job, les points de contrôle du job sont conservés jusqu'à ce que vous supprimiez le job.

6.4 Reboot et relance automatique d'un workflow

La persistance de workflow autorise la reprise d'un traitement après le reboot d'une machine.

C'est à l'utilisateur de redémarrer le job ou d'implémenter une reprise automatique.

L'exemple cité sur cette page détaille les étapes d'une relance automatique. Comme il est précisé, le job suspendu est enregistré dans un répertoire associé au profile utilisateur ayant exécuté le workflow. La contrainte est donc d'ouvrir une session avec le même compte.

Pour cet exemple réutilisons le script *\Source\TestSuspend.ps1*. Auparavant modifiez le nom de son chemin d'accès :

```
Get-Job|Remove-Job
$ScriptPath=' *A-MODIFIER* \Source\TestSuspend.ps1'
Invoke-Command -ComputerName LocalHost –Filepath $ScriptPath
Get-Job
#Aucun job
```

Invoke-Command exécute le job du workflow dans une nouvelle session locale, ce job est donc inaccessible pour la session courante, si on inverse le scénario ce problème d'accès demeure.

Si on exécute le script dans la session courante puisqu'on liste le contenu du répertoire de sauvegarde du compte utilisé pour cet exemple :

```
.$ScriptPath
$Path_CP="$env:LocalAppData\Microsoft\Windows\PowerShell\WF\PS\Default"
Dir $Path_CP  -Name -Directory
```

On constate désormais la présence de deux répertoires de sauvegarde :

```
S-1-5-21-3393592527-2738484632-2348561448-1001_EL
S-1-5-21-3393592527-2738484632-2348561448-1001_EL_NI
```

Pour obtenir la liste des jobs exécutés par **Invoke-Command,** on doit de nouveau exécuter la recherche dans une nouvelle session locale :

```
Invoke-Command -ComputerName localhost {Get-Job}|Select *
```

La propriété *StatusMessage* contient la chaîne suivante :

```
StatusMessage  : Ce PSWorkflowJob a été récupéré à partir d'une autre session dans un état suspendu. …
```

6.4.1 Restart-Computer

Le paramètre -*Wait* de ce cmdlet attend que l'ordinateur ciblé redémarre, puis s'y reconnecte avant d'exécuter la commande suivante.

Le paramètre *–Protocole* spécifie le protocole à utiliser pour redémarrer les ordinateurs. Les valeurs valides sont WSMan et DCOM. La valeur par défaut est DCOM. Les deux approches utilisent en interne la méthode '*Win32shutdown*' de la classe WMI **Win32_OperatingSystem**.

Comme indiqué précédemment, le paramètre -*Wait* est autorisé dans un workflow afin de redémarrer la machine locale exécutant le workflow, un point de contrôle est alors automatique créé.

7 Liens

La documentation d'une nouvelle version détaille différent points dont les nouveautés et implicitement les limites de la version précédente.

Windows PowerShell Script Workflow Debugging.

Document d'architecture de PSWF.

Le fichier " \Documentation\MS Connect-bugs Workflow.htm" contient la liste des bugs, concernant les workflows, référencés au 03 mars 2015.

8 Conclusion

Powershell facilite effectivement la transformation de code vers Windows **W**orkflow **F**oundation, mais PSWF impose une autre manière de coder, car tout ce qu'on connait doit être revu et adapté.

Que faire également avec les traitements futurs et existants ? A moins de n'utiliser que des blocs ***InlineScript***, on se doute que l'écriture de code ciblant les deux environnements ne sera pas une mince affaire.

Faute de temps et d'infrastructure de test, je n'ai pas abordé la création de workflow sous Visual Studio, ni les sessions de workflows, un prochain chapitre peut être.

Enfin, si certains se posent la question de savoir pourquoi il y a des trous dans l'emmental, moi j'aimerais bien savoir pourquoi il y en a dans la documentation de Powershell ;-)