PowerShell version 2 :
La gestion des événements.

Par Laurent Dardenne, le 3 juin 2010.

Niveau

Débutant Avancé Confimé

Une des limites de PowerShell version 1 est son incapacité à gérer nativement des événements au sein de la console.

La version 2 comble cette lacune, ce tutoriel vous propose d'aborder les différentes possibilités offertes.

Les fichiers sources :

ftp://ftp-developpez.com/laurent-dardenne/articles/Windows/PowerShell/La-gestion-des-evenements-sous-PowerShell-version-2/fichiers/La-gestion-des-evenements-sous-PowerShell-version-2.zip

Testé avec PowerShell V2 sous Windows XP sp3 et Windows Seven.

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

Chapitres

1 Mais que se passe-t-il ?

Ça, nous aimerions bien le savoir ! Cela tombe bien puisque le système Windows passe son temps à émettre, c'est-à-dire à *produire en envoyant hors de soi*, des messages.

S'il fonctionne ainsi c'est que ses concepteurs ont jugé que quelque chose passerait une partie de son temps à écouter, c'est-à-dire à prêter attention à sa production.

Le système Windows ne prêche pas dans le désert, il fait connaître ce qui se passe en lui à qui veut l'entendre. On peut donc prêter attention à un fait que l'on jugera, selon les situations, important. Ce fait peut être provoqué par l'utilisateur (clic souris) ou par le système (timer).

Mais comment être averti ?

2 Il s'est passé quelque chose !

La notion d'événement, un fait, est très fugace et ce qui persiste dans le système prend la forme d'un message. De notre côté, un événement se produit à la réception du message.

Une fois un message émis, on doit pouvoir déterminer :

- ✓ quel est l'émetteur, quelle partie ou objet du système, le QUI,
- ✓ quel est le type du message afin d'en extraire les informations rattachées, presque tous les messages contiennent des informations structurées, qui nous informent sur le QUOI.

Le système Windows nous notifie un événement en utilisant une file d'attente de messages, étant donné que nous sommes dans un environnement multitâche, un traitement associé à un événement particulier en cours d'exécution peut être momentanément suspendu, et pendant cette mise en attente le système peut déclencher une autre occurrence de l'événement qui nous intéresse. Ainsi, le système Windows s'assure de nous livrer tous les messages qui nous concernent.

3 Je sais ce qu'il s'est passé.

L'apport d'une gestion d'événement permet donc d'être informé que quelque chose c'est passé afin d'effectuer une action en retour.

Suite à un événement, le mécanisme de Windows dépose un message dans la file d'attente de notre application PowerShell, chaque application (plus exactement chaque thread) possède une file d'attente dédiée. Une fois ceci fait, pour accéder à ce message on doit passer par un intermédiaire, appelé gestionnaire d'événement, c'est lui qui va nous permettre d'associer un traitement à chaque réception d'un message, et c'est également lui qui déclenche ce traitement, puisque lui seul connaît le ou les destinataires du message.

Étant sous Dotnet, et à la différence du C, sous PowerShell on ne manipule pas la file d'attente de message du thread, c'est-à-dire qu'on ne code pas «l'écouteur», on est informé des événements au travers d'un mécanisme d'abonnement.

Ce qui fait qu'on ne gère que le ou les événements qui nous intéressent, tous les messages sont bien déposés dans la file d'attente du thread, mais ceux qui ne nous intéressent pas sont pris en charge et supprimés par le système.

Par exemple, à la fermeture d'une session PowerShell émet toujours l'événement *PowerShell.Closing*, mais si on ne s'y abonne pas, c'est-à-dire si on ne lui associe pas de traitement, cet événement ne déclenche rien.

Notez qu'un événement peut signifier une opération ayant lieu avant, pendant ou après un traitement particulier.

Par exemple le déclenchement de l'événement *PowerShell.Closing* signifie que la session est en cours d'arrêt, celui de Word *Workbook.BeforeSave* se produit avant l'enregistrement du classeur.

Note

La documentation de PowerShell parle d'événement au lieu de message, étant donné qu'on référence le mécanisme de gestion d'événements sous dotnet, et pas la notion de message de Windows Win32.

Les définitions que l'on peut trouver sur internet sont souvent différentes et peuvent créer une confusion, car tout le monde parle de la même chose, mais pas de la même manière ou donne un sens particulier à un même mot. Cela dit j'en ajoute certainement une de plus ;-).

Ainsi, on peut confondre l'eventlog de Windows avec un événement système ou encore avec la programmation événementielle.

4 Mise en œuvre

Sous PowerShell la gestion des événements ne peut se faire que sur des objets issus du Framework .NET. Les messages Windows natifs, tel que *WM_DEVICECHANGED*, ne peuvent être traités directement sous PowerShell.exe, car c'est une application console qui ne dispose pas de méthode de répartition de message, appelée aussi boucle de message (celle-ci gère l'attente et la réception).

Une Winform quant à elle met en place une telle boucle de message, nous verrons qu'il reste possible d'envoyer un message Windows.

Avant toute chose on doit connaître les messages mis à notre disposition par une classe dotnet.

Les classes déclarent des événements et on s'abonne aux instances de ces classes, c'est-à-dire à des objets. Les concepteurs de PowerShell ont donc modifié le cmdlet **Get-Member** afin de pouvoir lister les membres de type ***Event*** :

```
$O="Test"
$O|Get-Member -Type Event
#ras
[System.String].GetEvents()
#ras
```

La classe *System.String* ne propose pas de gestionnaire d'événement, essayons avec la classe *Process* :

```
 #Possible conflit avec les cmdlets du projet opensource PSCX
$Process=Microsoft.PowerShell.Management\start-process Notepad.exe -pass `
 -WindowStyle Minimized
#[System.Diagnostics.Process].GetEvents()
$Process|Get-Member -Type Event
```

```
   TypeName: System.Diagnostics.Process
Name                MemberType Definition
----                ---------- ----------
Disposed            Event      System.EventHandler Disposed(System.Object, System.EventArgs)
ErrorDataReceived   Event      System.Diagnostics.DataReceivedEventHandler  ErrorDataReceived(
Exited              Event      System.EventHandler Exited(System.Object, System.EventArgs)
OutputDataReceived  Event      System.Diagnostics.DataReceivedEventHandler OutputDataReceived…
```

Celle-ci propose quatre membres de type événement, étudions celui nommé *Exited.*

Il est du type ***System.EventHandler*** et sa signature est ***(System.Object, System.EventArgs)***, nous verrons plus avant la signification de ses définitions. Vous pouvez consulter sa documentation sur MSDN : http://msdn.microsoft.com/fr-fr/library/system.diagnostics.process.exited(v=VS.80).aspx

Si on essaie d'utiliser ce membre soit comme une propriété :

```
$Process.Exited
```

```
#Ras
```

Soit comme une méthode :

```
$Process.Exited()
```

```
L'appel de la méthode a échoué parce que [System.Diagnostics.Process] ne contient pas de méthode nommée « Exited ».
```

On constate qu'il ne nous est pas d'une grande utilité. On doit d'abord s'abonner à cet événement, à l'aide du cmdlet **Register-ObjectEvent,** et lui associer un traitement, un bloc de code :

```
Register-ObjectEvent $Process Exited -SourceIdentifier MonEvenement `
 -Action {Write-host "Fin de Notepad." -Fore green}|Out-Null
```

Nous reviendrons également sur la valeur renvoyée par cmdlet **Register-ObjectEvent**, pour le moment terminons notre process :

```
Stop-Process -id $Process.Id
```

```
Fin de Notepad.
```

L'événement ***Exited***, qui selon sa documentation *se produit lorsqu'un processus se termine*, déclenche bien l'exécution du code qu'on lui a associé, ainsi cet événement est consommé et n'existe plus dans la file d'attente des messages.

Affichons la liste des abonnements existants à l'aide du cmdlet **Get-EventSubscriber** :

```
Get-EventSubscriber
```

```
SubscriptionId   : 2
SourceObject     : System.Diagnostics.Process (notepad)
EventName        : Exited
SourceIdentifier : monEvenement
Action           : System.Management.Automation.PSEventJob
HandlerDelegate  :
SupportEvent     : False
```

```
ForwardEvent     : False
```

On peut récupérer notre abonnement en précisant son identifiant :

```
$Abonnement=Get-EventSubscriber MonEvenement
$Abonnement
```

```
SubscriptionId   : 2
…
```

Désormais, puisque l'objet source, c'est-à-dire le producteur de l'événement, n'existe plus :

```
$Abonnement.SourceObject.HasExited
```

```
$True
```

Cet abonnement n'est plus utile, on peut donc l'annuler grâce au cmdlet **UnRegister-Event** :

```
UnRegister-Event MonEvenement
```

Note :

La liste des abonnés n'étant pas ordonnée, les éléments sont juste numérotés, on doit la trier avant d'effectuer une recherche, par exemple pour retrouver l'id du dernier abonnement inséré :

```
Get-EventSubscriber "MonEvenement"|
 Sort SubscriptionId |
 Select-Object SubScriptionID -Last 1|
 Foreach-Object {$_.SubScriptionID}
```

4.1 Approche synchrone

La gestion d'événement que l'on vient de voir est une approche asynchrone et n'est pas bloquante.Un mécanisme d'écoute interne à PowerShell se charge de déclencher le traitement.

Si, lors de la déclaration de l'abonnement, on ne lui associe pas de traitement, l'événement produit n'est pas consommé, mais seulement placé dans la file d'attente des messages propre à PowerShell :

```
$Process=Microsoft.PowerShell.Management\start-process Notepad.exe –pass -
WindowStyle Minimized
$null=Register-ObjectEvent $Process Exited -SourceIdentifier monEvenement
Stop-Process –id $Process.Id
#RAS
```

Le cmdlet **Get-Event** lit les événements à partir de cette file d'attente sans pour autant les consommer, ceux-ci y persistent tant que l'on ne les supprime pas:

```
$Event=Get-Event MonEvenement
$Event
```

```
ComputerName    :
RunspaceId      : 1e55667e-856a-4033-9da2-b9a727843324
EventIdentifier : 2
Sender          :
SourceEventArgs : System.EventArgs
SourceArgs      :
SourceIdentifier : monEvent
```

```
TimeGenerated      : 18/04/2010 13:59:51
MessageData
$Event=Get-Event MonEvenement
```

Le message émis par le système Windows lors de la fin du process, car c'est lui qui gère les process, est donc transformé par le Framework dotnet en un événement. Le résultat de son déclenchement est dupliqué par chaque session PowerShell possédant un abonnement sur cet événement provenant d'un même objet. PowerShell ne duplique que les événements pour lesquels il existe un abonnement (voir la liste renvoyée par **Get-EventSubscriber)**.

L'exemple suivant propose de surveiller un même événement sur un process identique, dans deux instances distinctes de PowerShell.exe :

```
$Process=Microsoft.PowerShell.Management\start-process Notepad.exe –pass -
WindowStyle Minimized
sleep 1
1..2|Foreach {
  Microsoft.PowerShell.Management\start-process PowerShell.exe -
WindowStyle Minimized -Argumentlist @'
   -NoExit  -Command "& {
     $P=Get-Process Notepad
     Register-ObjectEvent $P Exited -Source Evt –A {Write-host 'Fin de
Notepad.'-fore Green}
    }"
'@
}
 #chargement de profile
sleep 10
Stop-Process -id $Process.Id
```

Si vous le souhaitez, vous pouvez bloquer l'exécution de votre script sur une attente d'événement, en synchrone donc, en utilisant le cmdlet **Wait-Event** :

```
UnRegister-Event "MonEvenement"
$Process=Microsoft.PowerShell.Management\start-process Notepad.exe –pass -
WindowStyle Minimized
Register-ObjectEvent $Process Exited -SourceIdentifier monEvenement
$Event=Wait-Event MonEvenement
```

Dans notre cas l'appel à **Wait-Event** n'est pas bloquant, car notre dernier événement, du même type que celui précisé, est toujours dans la file d'attente. On doit donc supprimer toutes les entrées ayant pour identifiant *monEvenement* avec le cmdlet **Remove-Event :**

```
Remove-Event monEvenement
Get-EventSubscriber MonEvenement
$Event=Wait-Event MonEvenement ; Get-EventSubscriber MonEvenement
```

Notez que l'annulation de l'abonnement "MonEvenement" n'est pas couplée à une suppression automatique des événements de type "MonEvenement" existant dans la file d'attente des messages.

Désormais l'appel est bloquant, mais ici il le restera puisque notre déclaration d'abonnement référence, non pas toutes les instances de Notepad, comme on peut le faire avec WMI, mais une instance particulière d'un process qui n'existe plus au moment de l'appel. Vous pouvez arrêter l'attente d'événement de **Wait-Event** par la saisie des touches *Control-C*.

Il reste possible de placer un timeout, en seconde, sur l'attente d'un événement :

```
wait-event MonEvenement -timeout 180
```

Ou d'attendre n'importe quel type d'événement inséré dans la file d'attente d'événements :

```
wait-event
```

Note :

Pour une attente bloquante sur une fin de process on peut également utiliser la méthode *WaitForExit()* de la classe **Process.**

4.2 Récupérer les informations sur l'événement

Dans le traitement déclenché, suite à une déclaration d'abonnement asynchrone, nous pouvons récupérer les informations rattachées à l'événement en accédant aux variables automatiques créées dans la portée du scriptblock *Action.*

La documentation référence par erreur deux variables automatiques nommées *$SourceEventArgs* et *$SourceArgs.* Celles-ci n'existent pas, seules les quatre suivantes sont déclarées :

$EventSubscriber	Contient un objet ***PSEventSubscribe****r* représentant l'abonné de l'événement qui est en cours de traitement. La valeur de cette variable est le même objet que celui retourné par l'applet de commande **Get-EventSubscriber**.
$Event	Contient un objet ***PSEventArgs*** représentant l'événement en cours de traitement. La valeur de cette variable est le même objet que celui retourné par l'applet de commande **Get-Event**. Vous pouvez par conséquent utiliser les propriétés de la variable *$Event***,** par exemple *$Event.TimeGenerated*, dans un bloc de script du paramètre Action.
$EventArgs	Contient des objets représentant les arguments de l'événement en cours de traitement. La valeur de cette variable peut aussi se trouver dans la propriété *SourceArgs* de l'objet retourné par **Get-Event**.
$Sender	Contient l'objet qui a généré cet événement. La valeur de cette variable peut aussi se trouver dans la propriété *Sender* de l'objet retourné par **Get-Event**.

Affichons leurs contenus à l'aide de la fonction ***Write-Properties***, disponible dans les sources de ce tutoriel.

Notez que ces variables automatiques sont propres au contexte de l'événement, c'est-à-dire le job, leurs référencements dans une fonction déclarée dans un autre contexte n'est pas possible :

```
 #Contexte de la session PowerShell
Function Write-EventVariables {
  dir variable:E*, variable:S*|sort name |% {write-host $_.name}
```

```
 Write-Warning "EventSubscriber"; wp $EventSubscriber
 Write-Warning "Event"; wp $Event
 Write-Warning "Eventargs"; wp $EventArgs
 Write-Warning "Sender"; wp $Sender
}
Unregister-event *
$Process=Microsoft.PowerShell.Management\start-process Notepad.exe –pass -
WindowStyle Minimized
$null=Register-ObjectEvent $Process Exited -SourceIdentifier monEvent –
Action {
  #Contexte différent de celui de la session PowerShell
 Write-EventVariables
}
Stop-Process -id $Process.Id
```

```
Error
ErrorActionPreference
ErrorView
ExecutionContext
ShellId
StackTrace
AVERTISSEMENT : EventSubscriber
AVERTISSEMENT : Event
AVERTISSEMENT : Eventargs
AVERTISSEMENT : Sender
```

Ceci étant connu déplaçons leurs affichages dans le scriptblock du paramètre *–Action* :

```
Unregister-event *
$Process=Microsoft.PowerShell.Management\start-process Notepad.exe –pass -
WindowStyle Minimized
$null=Register-ObjectEvent $Process Exited -SourceIdentifier monEvent –
Action {
$Event # N'est pas affichée
 Write-Warning "EventSubscriber"; Write-Properties $EventSubscriber
 Write-Warning "Event"; wp $Event
 Write-Warning "Eventargs"
  wp $EventArgs; Write-host $EventArgs.ToString()
 Write-Warning "Sender"; wp $Sender
}
Stop-Process -id $Process.Id
```

On peut constater que dans le scriptblock de notre traitement d'événement, l'émission d'objet dans le pipeline, ici la variable ***$Event***, ne provoque pas d'affichage dans la console, on doit donc utiliser un cmdlet d'affichage.

Dans notre exemple l'émetteur de l'événement *($Sender*) contient le process Notepad, mais certains membres de l'instance ne sont plus valides. L'appel, direct ou indirect, à sa méthode ***toString()*** provoquera une erreur d'affichage :

```
$Event.Sender
$Event.Sender.ToString()
```

```
Exception lors de l'appel de « ToString » avec « 0 » argument(s) : « Les informations demandées ne sont pas disponibles, car le processus n'est plus exécuté. »
```

C'est pourquoi la fonction Write-Propertie renseigne le champ *$Event.Sender* avec"Errror - not applicable" :

```
#Le résultat de l'affichage a été tronqué
AVERTISSEMENT : EventSubscriber
…
EventName : Exited
SourceIdentifier : monEvent …

AVERTISSEMENT : Event
Sender : Errror - not applicable.
SourceEventArgs : System.EventArgs
SourceIdentifier : monEvent …

AVERTISSEMENT : Eventargs
System.EventArgs

AVERTISSEMENT : Sender
__NounName : Process
ExitCode : -1 …
```

La variable *$EventArgs* contient bien une instance, mais celle-ci ne possède pas de propriété. Pour en comprendre la raison il faut consulter la signature de l'événement *Exited* qui est ***(System.Object, System.EventArgs)***. Les arguments d'un événement qui sont de type ***System.EventArgs*** ne portent pas d'information complémentaire sur l'événement, voici ce que nous dit la documentation MSDN à son sujet :

«*Cette classe ne contient pas de données d'événement ; elle est utilisée par des événements qui ne passent pas d'informations d'état à un gestionnaire d'événements lorsqu'un événement est déclenché. Si le gestionnaire d'événements nécessite des informations d'état, l'application doit dériver une classe à partir de cette classe pour contenir les données. »*

Voyons le cas où un événement utilisant dans sa signature une classe dérivée de ***System.EventArgs***.

Nous utiliserons l'événement *EntryWritten* la classe **EventLog**, celui-ci se déclenche lors de l'ajout d'une entrée dans un eventlog. La classe de ses arguments d'événement est ***System.Diagnostics.EntryWrittenEventArgs*** :

```
function New-EventLogEntryType([string]$EventLogName="Windows PowerShell")
{ #crée un événement de test dans l'eventlog $EventLogName
 $Event=new-object System.Diagnostics.EventLog($EventLogName)
 $Event.Source="TestEventing"
 $Event.WriteEntry("Test événement",
 [System.Diagnostics.EventLogEntryType]::Information)
```

```
}

$EL=Get-EventLog -list|? {$_.Log -match "Windows PowerShell"}
Unregister-event *
$null=Register-ObjectEvent $EL EntryWritten -SourceIdentifier AjoutLog `
-Action {
 Write-Warning "EventSubscriber"; Write-Properties $EventSubscriber
 Write-Warning "Event"; wp $Event
 Write-Warning "Eventargs"; wp $EventArgs;
 Write-Warning "Sender"; wp $Sender;
 #Write-Warning "Eventargs.Entry"; wp $EventArgs.Entry
}
New-EventLogEntryType
```

Cette fois ci la variable *$Eventargs* contient des informations supplémentaires :

```
AVERTISSEMENT : Event
Sender : System.Diagnostics.EventLog
SourceArgs : System.Diagnostics.EventLog System.Diagnostics.EntryWrittenEventArgs
SourceEventArgs : System.Diagnostics.EntryWrittenEventArgs
…
AVERTISSEMENT : Eventargs
Entry : System.Diagnostics.EventLogEntry
System.Diagnostics.EntryWrittenEventArgs
…
```

Si on complète l'affichage avec son membre nommé *Entry*, de la classe ***EntryWrittenEventArgs***, on voit qu'il contient les informations de l'entrée que l'on vient d'ajouter dans l'eventlog :

```
Write-Warning "Eventargs.Entry"; wp $EventArgs.Entry
```

```
AVERTISSEMENT : Eventargs.Entry
Category : (0)
CategoryNumber : 0
Container :
Data :
EntryType : Information
EventID : 0
Index : 1794
InstanceId : 0
MachineName : xxxxx
Message : Test événement
ReplacementStrings : Test événement
Site :
Source : TestEventing
…
```

Note :

Il ne semble pas possible d'utiliser des événements retournant leurs données modifiées tels que AddingNewEventArgs ou CancelEventargs (ce dernier fournit des données pour un événement annulable).

4.3 Déclarer plusieurs abonnements pour un événement d'instance

Il est possible de s'abonner plusieurs fois à un même événement d'une instance :

```
Unregister-event *
$EL=Get-EventLog -list|Where {$_.Log -match "Windows PowerShell"}

Register-ObjectEvent $EL EntryWritten AjoutLog  –Action {
  Write-Host "1-Réception nouvel événement" –fore White }|Out-Null

Register-ObjectEvent $EL EntryWritten SecondAjoutLog  –Action {
  Write-Host "2-Réception nouvel événement" –fore Green}|Out-Null

New-EventLogEntryType
```

```
1-Réception nouvel événement
2-Réception nouvel événement
```

Il suffit de préciser un nom d'identifiant unique lors de la déclaration de l'abonnement, ici *AjoutLog* et *SecondAjoutLog*.

Bien évidemment, l'annulation d'un abonnement d'un événement ne concerne que celui précisé :

```
Unregister-event AjoutLog
New-EventLogEntryType
```

```
2-Réception nouvel événement
```

Cette possibilité permet de moduler les traitements si besoin où de traiter différemment le même message, par exemple l'écrire sur la console et l'écrire dans un fichier.

4.4 Création d'événement personnel

PowerShell version 2 permet également de déclencher nos propres événements à l'aide du cmdlet **New-Event** :

```
Remove-Event *
$MonEvnt=New-Event -SourceIdentifier Nommé
Get-Event
```

Dès lors, la file d'attente des événements contient un message qui est identique à celui contenu dans la variable ***$MonEvnt***. C'est le minimum pour générer un message personnel, pas besoin d'abonnement, ni de manipulation de classe, ni d'instance de classe.

Allons plus loin, renseignons l'émetteur et les arguments d'événement :

```
$UnObjet=New-object psObject -Property @{Nom="MonObjet";Nombre=4}
$Data=@{CS=Get-CallStack;Path=$Pwd}
$MonEvnt=New-Event "Personnel" -Sender $UnObjet -EventArguments $Data
```

Vérifions l'événement généré :

```
$MonEvnt.SourceArgs[0]
```

```
Name           Value
----           -----
Path           C:\Temp
CS             {Au niveau de ligne : 1 Caractère : 25+ $Data=@{CS=Get-CallStack <<<< ;Path=$Pwd}, }
```

```
$MonEvnt.Sender
```

```
#ras
```

On récupère bien les arguments, mais il y a un souci avec le sender, car PowerShell ne copie pas les membres synthétiques (ce comportement semble dû à un bug). Dans ce cas, passons une variable PowerShell :

```
$MonEvnt=New-Event "Personnel" -Sender (Get-Variable UnObjet) -event $Data
$MonEvnt.Sender
```

```
Name           Value
----           -----
UnObjet        @{Nombre=4; Nom=MonObjet}
```

Cette fois-ci, on récupère bien tous les membres de notre objet que l'on peut recréer ainsi :

```
$Var=$MonEvnt.Sender.Value
$Var
```

```
Nombre  Nom
-------  ---
     4  MonObjet
```

Testons un autre cas :

```
 #On utilise un seul contexte
function Test{
 $UnObjetLocal=New-object psObject -Property @{
  Nom="MonObjetLocal";Nombre=12}
 $null=New-Event "Personnel" -Sender (Get-Variable UnObjetLocal) -event @{
  CS=Get-CallStack;Path=$Pwd}
 #Remove-Variable UnObjetLocal
}
Remove-Event *
Test
(Get-Event).Sender
```

```
Name             Value
----             -----
UnObjetLocal     @{Nombre=12; Nom=MonObjetLocal}
```

Ici, étant donné que l'objet local reste référencé en tant qu'argument du paramètre *–Sender*, sa portée n'a pas d'influence. En revanche si on supprime la variable locale dans la fonction, et ce, après avoir créé l'événement, le contenu du paramètre *–Sender* s'en trouve modifié :

```
function Test{
 $UnObjetLocal=New-object psObject -Property @{
  Nom="MonObjetLocal";Nombre=12}
 $null=New-Event "Personnel" -Sender (Get-Variable UnObjetLocal) -event @{
  CS=Get-CallStack;Path=$Pwd}
 Remove-Variable UnObjetLocal
```

```
}
Remove-Event *
Test
(Get-Event).Sender
```

```
Name                 Value
----                 -----
UnObjetLocal
```

Nous pouvons également nous abonner aux événements personnalisés, nous avons seulement besoin de l'identifiant de notre événement personnalisé, mais l'abonnement se fait avec un autre cmdlet :

```
Register-EngineEvent "Personnel" -Action { Write-Warning "Evénement personnalisé" }
Test
```

Le cmdlet **Register-EngineEvent** crée un abonnement aux événements générés par le moteur Windows PowerShell et par le cmdlet **New-Event**. Les variables automatiques sont également disponibles dans la portée du scriptblock du paramètre *-Action*.

À ce jour, le moteur Windows PowerShell ne propose que l'événement, *PowerShell.Exiting.*

Cet événement est uniquement déclenché par l'usage de l'instruction **Exit :**

```
$EventName=[System.Management.Automation.PsEngineEvent]::Exiting
Register-EngineEvent –SourceIdentifier $EventName -Action {Notepad}
#Exit
```

Pour information, c'est la méthode *Close* de la classe interne **LocalRunspace** qui émet cet événement :

```
this.Events.GenerateEvent("PowerShell.Exiting",null,new object[0],null,true);
```

4.4.1 New-Event et les abonnements existants

Pour les abonnements liés à une instance, il est possible de déclencher l'événement associé sans pour autant que l'émetteur soit réellement l'instance utilisée lors de l'abonnement de **Register – ObjectEvent :**

```
Remove-Event * ; Unregister-Event *
$EL=Get-EventLog -list|? {$_.Log -match "Windows PowerShell"}
$null=Register-ObjectEvent $EL EntryWritten -SourceIdentifier AjoutLog `
-Action {
 Write-host "Event EntryWritten" -fore green
 Write-Warning "EventSubscriber"; Write-Properties $EventSubscriber
 Write-Warning "Event"; wp $Event
 Write-Warning "Eventargs"; wp $EventArgs;
 Write-Warning "Sender"; wp $Sender;
}
New-Event AjoutLog
```

Ici ce déclenchement ne crée pas d'entrée dans l'eventlog cible, à savoir celui nommé « Windows PowerShell », c'est assez déroutant puisque c'est l'instance proposant l'événement qui normalement devrait être la seule à générer cet événement !

Ce comportement peut être mis à profit à des fins de tests, en prenant soin toutefois de renseigner l'intégralité des champs des objets utilisés comme argument des paramètres *EventArguments* et *Sender*.

Je vous recommande de mettre en place une règle de nommage des identifiants d'abonnement, par exemple le préfixe **PS**, permettant de différencier les événements personnalisés des événements d'instances.

Sachez aussi que le cmdlet **New-Event** ne vous force pas, pour un même identifiant d'événement personnalisé, à respecter la signature utilisée lors de sa première déclaration :

```
$null=New-Event "Personnel" -Sender (Get-Variable UnObjetLocal) –event @{
  CS=Get-CallStack;Path=$Pwd}
$null=New-Event "Personnel" -Sender 1 -event @{Nom="différent"}
```

Soyez donc attentif aux possibles effets de bord.

4.4.2 Objet personnalisé

Puisque nous pouvons déclarer des événements personnalisés, essayons de les associer à des objets personnalisés. Ceux-ci ne peuvent qu'émettre des événements et pas en consommer directement puisque, comme nous allons le voir, l'argument du paramètre *–Action* des cmdlets d'abonnements ***Register-xxxEvent*** est transformé en un objet spécialisé de type **PSEventJob**.

L'exemple suivant crée un objet personnalisé dont sa méthode *AffectationEvent* déclenche deux fois un même événement. L'un avant de modifier la valeur de sa propriété nommée **I**, l'autre après la modification de ce champ :

```
$MonObjet=New-object psObject
$MonObjet|add-member -membertype NoteProperty I 5
$MonObjet|add-member -membertype ScriptProperty Name {[String]"MonObjet"}
$MonObjet|add-member ScriptMethod ModificationEvent -value {
 Param ($EventArgs,$MessageData)
  New-Event "PersonnelEvent.Modification" (gv $this.Name) $EventArgs `
$MessageData
  }
$MonObjet|add-member -membertype ScriptProperty  -Name Nombre -value {
  $this.I} `
  -secondvalue {
       $Oldvalue=$this.I

       $Data=@{Old=$OldValue}
       $this.ModificationEvent($Data,"[PreProcess] Modification de la
propriété Nombre")
```

```
       $this.I=$args[0];

       $Data=@{Old=$OldValue;New=$this.I}
       $this.ModificationEvent($Data,"[PostProcess] Modification de la
propriété Nombre")
    }

UnRegister-Event "PersonnelEvent.Modification" -ea SilentlyContinue
$null=Register-EngineEvent "PersonnelEvent.Modification" -Action {
 Write-Warning $Event.MessageData
 Write-Host ("Old={0}`tNew={1} " -F $Event.SourceArgs[0].Old,
$Event.SourceArgs[0].New) -Fore green
}
$MonObjet.Nombre=10;
```

```
AVERTISSEMENT : [PreProcess] Modification de la propriété Nombre
Old=5   New=
AVERTISSEMENT : [PostProcess] Modification de la propriété Nombre
Old=5   New=10
```

Cette manière de faire peut être utile si votre événement exécute un traitement relativement long.

Le mieux dans ce cas est de créer deux événements distincts :

```
$MonObjet=New-object psObject
$MonObjet|add-member -membertype NoteProperty I 5
$MonObjet|add-member -membertype ScriptProperty Name {[String]"MonObjet"}
$MonObjet|add-member ScriptMethod PreModificationEvent -value {
 Param ($EventArgs,$MessageData)
  New-Event "PSPreModification" (gv $this.Name) $EventArgs $MessageData
}
$MonObjet|add-member ScriptMethod PostModificationEvent -value {
 Param ($EventArgs,$MessageData)
  New-Event "PSPostModification" (gv $this.Name) $EventArgs $MessageData
}
$MonObjet|add-member -membertype ScriptProperty  -Name Nombre -value {
  $this.I} `
  -secondvalue {
       $Oldvalue=$this.I

       $Data=@{Old=$OldValue}
       if (Get-EventSubscriber "PSPreModification")
        {$this.PreModificationEvent($Data,"Modification de la propriété
Nombre") }

       $this.I=$args[0]
```

```
       $Data=@{Old=$OldValue;New=$this.I}
       if (Get-EventSubscriber "PSPostModification")
        { $this.PostModificationEvent($Data,"Modification de la propriété
Nombre") }
}
```

L'ajout de la vérification de l'existence de l'abonnement évite de déclencher inutilement des événements s'il n'existe aucun abonnement s'y rattachant.

Ensuite on peut s'abonner soit à l'un soit à l'autre des événements ou aux deux :

```
UnRegister-Event "PersonnelEvent.*Modification" -ea SilentlyContinue
$null=Register-EngineEvent "PSPreModification" -Action {
 Write-Host ("[PreProcess] Old={0}`tNew={1} " -F $Event.SourceArgs[0].Old,
$Event.SourceArgs[0].New) -Fore green
}
$null=Register-EngineEvent "PSPostModification" -Action {
Write-Host ("[PostProcess] Old={0}`tNew={1} " -F $Event.SourceArgs[0].Old,
$Event.SourceArgs[0].New) -Fore green
}
$MonObjet.Nombre=7
```

```
[PreProcess] Old=10     New=
[PostProcess] Old=10    New=7
```

Notez que le cmdlet **UnRegister-Event** gère le joker (regex) présent dans le nom de l'identifiant.

On aurait pu également nommer les événements *BeforeProcess* et *AfterProcess*.

Ce type de traitement peut également être mis en œuvre à l'aide du cmdlet **Set-PsBreakPoint**.

4.4.2.1 Création de classe d'événement

On peut aussi définir une classe dérivée de la classe **EventArgs** afin de l'associer à un événement personnalisé :

```
$code=@"
using System;
public class TestEventArgs:EventArgs  {
  public int OldValue;
  public int NewValue;
  public TestEventArgs(int aOld, int aNew)
  {
    this.OldValue = aOld;
     this.NewValue = aNew;
  }
}
"@
Add-Type  $code
```

Ensuite, on crée une instance de cette classe et on la lie au paramètre *EventArguments* du cmdlet **New-Event** :

```
[Int32] $I=0
$o=New-object psObject
$o|add-member -membertype NoteProperty I $I –pass|
   add-member -membertype ScriptProperty -Name Nombre `
     -value { $this.I}`
     -secondvalue {
       #Génére un event propriétaire
       $event=new-object TestEventArgs($this.I,$Args[0])
       $this.I=$args[0];
       New-Event "PowerShell.IncValue" -Sender $this –EventArguments `
         $event -MessageData "Affectation"
     }
```

Le code d'usage de cette classe d'arguments d'événement :

```
Register-EngineEvent "PowerShell.IncValue" -Action {
  write-Warning "$($Event.MessageData). Ancienne valeur
$($EventArgs.OldValue)`tnouvelle valeur $($EventArgs.NewValue)"
}
$o.Nombre=10
```

4.4.3 Partager un abonnement entre plusieurs événements d'instances

Si on souhaite gérer le même événement sur plusieurs instances tout en partageant le traitement au sein d'un job, on doit mettre en place le mécanisme suivant :

- On s'abonne à un événement personnalisé,
- pour chaque instance on s'abonne à l'événement concerné par ce partage,
  - dans le scriptblock de ces abonnements on déclenche notre événement personnalisé :

```
 #On s'abonne à un événement personnalisé
$null=Register-EngineEvent "ProcessExited" -Action {
 Write-Host ("Fin du Process id ={0}" –F $Sender.Id) -Fore green
 Sleep 1
}
1..3| Foreach {
 $Process=Microsoft.PowerShell.Management\start-process Notepad.exe `
 –pass -WindowStyle Minimized
   #On s'abonne à l'événement de l'instance
 Register-ObjectEvent $Process Exited -SourceIdentifier `
 "monEvent$_" –Action {
    #On déclenche notre événement personnalisé
   New-Event "ProcessExited" $Sender $EventArgs } -SupportEvent }
```

L'usage du paramètre –*SupportEvent*, du cmdlet **Register-ObjectEvent,** masque l'abonné déclaré sur l'événement de chaque instance et les jobs respectifs, de plus le déclenchement de l'événement de l'instance ne renvoie aucun résultat, mais l'action associée est bien exécutée.

On a donc ici trois abonnés, un pour chaque process, plus un abonné sur l'événement personnalisé, et c'est ce dernier qui effectue un traitement.

Le cmdlet **Get-EventSubscriber** peut retrouver les abonnements masqués si on précise le switch *-Force* :

```
Get-EventSubscriber -Force
```

Affichons la liste des abonnements et des jobs :

```
Write-Warning "Abonnés"; Get-EventSubscriber
```

```
AVERTISSEMENT : Abonnés
…
SourceIdentifier : ProcessExited
…
```

```
Write-Warning "Jobs" ; Get-Job
```

```
AVERTISSEMENT : Jobs
Module      : __DynamicModule_bfca817c-df5f-4e47-817f-2052f4114ab0
…
State       : NotStarted
…
```

```
#Déclenchement
Get-Process Notepad|Stop-Process
```

```
Fin du Process id =3396
Fin du Process id =4112
Fin du Process id =5368
```

```
#Finalisation
UnRegister-Event * -Force
Remove-Job *
```

4.4.4 Déclarer plusieurs abonnements pour un même événement personnalisé

Le code suivant reste possible, bien qu'en contradiction, à mon avis, avec la documentation qui précise qu'un identifiant d'abonnement doit être unique.

Pour cet exemple on construit dynamiquement le scriptblock afin de bénéficier de la substitution de variable :

```
Unregister-Event *
1..3| Foreach {
 Register-EngineEvent -SourceIdentifier "MyEventID" -Action (
 $ExecutionContext.InvokeCommand.NewScriptBlock(@"
 Write-Host "MyEventID-$_" -Fore green
"@
))
}
```

```
Get-EventSubscriber
```

```
#3 entrées portant le même identifiant
…
```

```
New-Event "MyEventID"
```

```
MyEventID-3
MyEventID-2
MyEventID-1
```

Ainsi, on peut déclarer plusieurs traitements pour un même événement personnalisé.

La suppression d'un de ces abonnés est plus délicate et nécessiterait de mémoriser leurs ID.

Note : Un bug est ouvert à ce sujet, ne sachant si c'est une erreur dans le code ou dans la documentation.

https://connect.microsoft.com/PowerShell/feedback/details/553493/register-engineevent-the-source-identifier-is-not-unique-in-the-current-session

4.5 Gestion d'événement d'une classe statique

Les classes statiques proposant des événements doivent être placées dans une variable avant de les manipuler :

```
$SystemEvents = [Microsoft.Win32.SystemEvents]
$timeChanged = Register-ObjectEvent -InputObject $systemEvents `
-EventName 'DisplaySettingsChanged' -Action {
  Write-Host "Résolution modifiée."
}
```

Cette classe propose d'autres événements, notamment *SessionEnding* qui se produit lorsque l'utilisateur essaie de fermer une session ou d'arrêter le système, la session étant encore active.

Source : http://stackoverflow.com/questions/2335623/what-is-the-syntax-to-subscribe-to-an-objects-static-event-in-powershell

4.6 Gestion d'événement d'objet COM

Les objets COM proposent eux aussi des événements, voyons cela avec l'application Word :

```
$Word = new-object -com word.application
$Word.visible = $true
$word|Gm –Type Event
```

```
   TypeName: Microsoft.Office.Interop.Word.ApplicationClass

Name                                      MemberType  Definition
----                                      ----------  ----------
ApplicationEvents2_Event_DocumentBeforeClose  Event       Microsoft.Office.Interop.Word.Application...
…
XMLValidationError                        Event       Microsoft.Office.Interop.Word.Application...
```

Cette liste référence les événements de la classe *ApplicationClass*, d'autres objets imbriqués peuvent proposer les leurs :

```
$doc = $word.documents.add()
$doc|gm -t event
```

```
TypeName: Microsoft.Office.Interop.Word.DocumentClass

Name                                MemberType Definition
----                                ---------- ----------
DocumentEvents2_Event_Close         Event      Microsoft.Office.Interop.Word.DocumentEvents2_Close...
XMLBeforeDelete                     Event      Microsoft.Office.Interop.Word.DocumentEvents2_XML...
```

La documentation des événements des objets de Word se trouve dans le fichier VBAWD10.CHM (Word 2003).

Ajoutons un abonné sur l'événement nommé *Quit* de la classe ***ApplicationClass*** :

```
$EventName="ApplicationEvents2_Event_Quit"
Register-ObjectEvent $word $EventName "Word_Quit" -Action {
 Write-Host "L'application Word viens de se terminer." -fore White}
$word.Quit()
Unregister-Event "Word_Quit"
```

```
Unregister-Event : Une exception a été levée par la cible d'un appel.
```

L'objet COM n'étant plus accessible l'annulation de l'abonnement provoque une erreur, affichons le détail des exceptions imbriquées :

```
$Error[0].Exception.InnerException.InnerException
```

```
Le serveur RPC n'est pas disponible. (Exception de HRESULT : 0x800706BA)
```

Essayons de libérer proprement l'application une fois celle-ci terminée :

```
[void][System.Runtime.Interopservices.Marshal]::ReleaseComObject($word)
Unregister-Event "Word_Quit"
#Resolve-Error
$Error[0].Exception.InnerException.InnerException
```

```
Un objet COM qui a été séparé de son RCW sous-jacent ne peut pas être utilisé.
```

Pour utiliser un objet COM dotnet crée un wrapper RCW (Runtime Callable Wrapper), celui-ci est un intermédiaire entre l'environnement COM (Win32 natif) et l'environnement dotnet (code managé). La finalisation de cet objet COM nécessite donc deux étapes, l'appel à la méthode *Quit()*, puis l'appel à la méthode *ReleaseComObject* qui libère réellement la mémoire du RCW et donc de l'objet COM.

L'objet COM lié à un abonnement doit donc être encore accessible lors de l'appel au cmdlet d'annulation d'abonnement :

```
$word = new-object -com word.application
$word.visible = $true
$EventName="ApplicationEvents2_Event_Quit"
Register-ObjectEvent $word $EventName "Word_Quit" -Action {
   Write-Host "L'application Word est fermée." -fore White
   Write-Warning "Annulation de l'abonnement"
   Unregister-Event "Word_Quit"
   }
$word.Quit()
[void][System.Runtime.Interopservices.Marshal]::ReleaseComObject($word)
```

Dans l'exemple suivant l'appel à *ReleaseComObject* sur l'objet ***$Doc*** doit se faire après l'appel à **Unregister-Event :**

```
$word = new-object -com word.application
$word.visible = $true
$EventName="ApplicationEvents2_Event_Quit"
Register-ObjectEvent $word $EventName "Word_Quit" -Action {
   Write-Host "L'application Word est fermée." -fore White
   Write-Warning "Annulation de l'abonnement"
   Unregister-Event "Word_Quit"
   }
$doc = $word.documents.add()
Register-ObjectEvent $Doc "DocumentEvents_Event_Close" "Doc_Close" -Action
{Write-Host "Document fermé." -fore Green}
$doc.Close()
 #Exception si on libére $doc AVANT l'annulation de l'événement
 #[void][System.Runtime.Interopservices.Marshal]::ReleaseComObject($doc)
Unregister-Event "Doc_Close"
[void][System.Runtime.Interopservices.Marshal]::ReleaseComObject($doc)

Sleep 3 ; $word.Quit()
[void][System.Runtime.Interopservices.Marshal]::ReleaseComObject($word)
```

4.7 Résumé des cmdlets liés à la gestion des événements

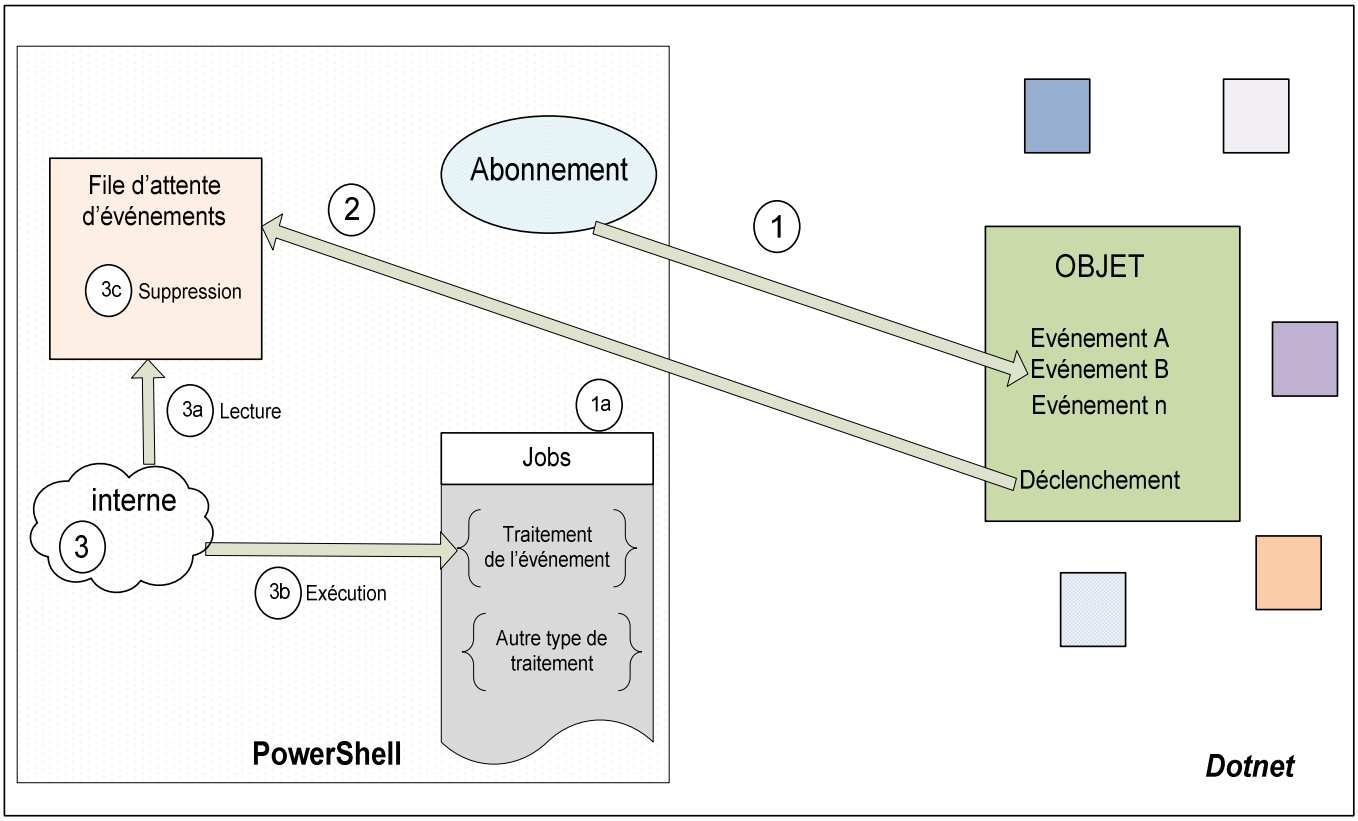
Get-EventSubscriber	Affiche la liste des abonnements.
Get-Event	Renvoie les événements présents dans la file d'attente d'événements de PowerShell.
New-Event	Déclenche un événement personnalisé.
Register-EngineEvent	S'abonne à un événement créé par **New-Event**.
Register-ObjectEvent	S'abonne à un événement d'une instance d'objet.
Register-WmiEvent	S'abonne à un événement WMI.
Remove-Event	Supprime un ou plusieurs événements dans la file d'attente d'événements
Unregister-Event	Annule un ou plusieurs abonnements.
Wait-Event	Gestion synchrone d'un événement, on attend tant qu'un événement n'est pas disponible dans la file d'attente d'événements de PowerShell.

Note :

Si, lors de l'appel à **UnRegister-Event**, des événements gérés de manière asynchrone sont encore présents dans la file d'attente des événements, ils seront tous traités avant que ce cmdlet se termine. Ce qui fait que dans certaines situations l'ordre de suppression des abonnements aura un impact sur vos traitements.

4.7.1 Synoptiques

Voici un résumé des opérations liées à la gestion d'un événement asynchrone :



En 1, dans l'environnement PowerShell, on s'abonne à l'événement d'une instance d'objet dotnet :

en 1a PowerShell crée en interne une tâche de fond contenant l'action déclenchée lors de l'arrivée de l'événement.

En 2, dans l'environnement Dotnet, l'objet déclenche l'événement qui nous intéresse, PowerShell le place dans sa file d'attente d'événements.

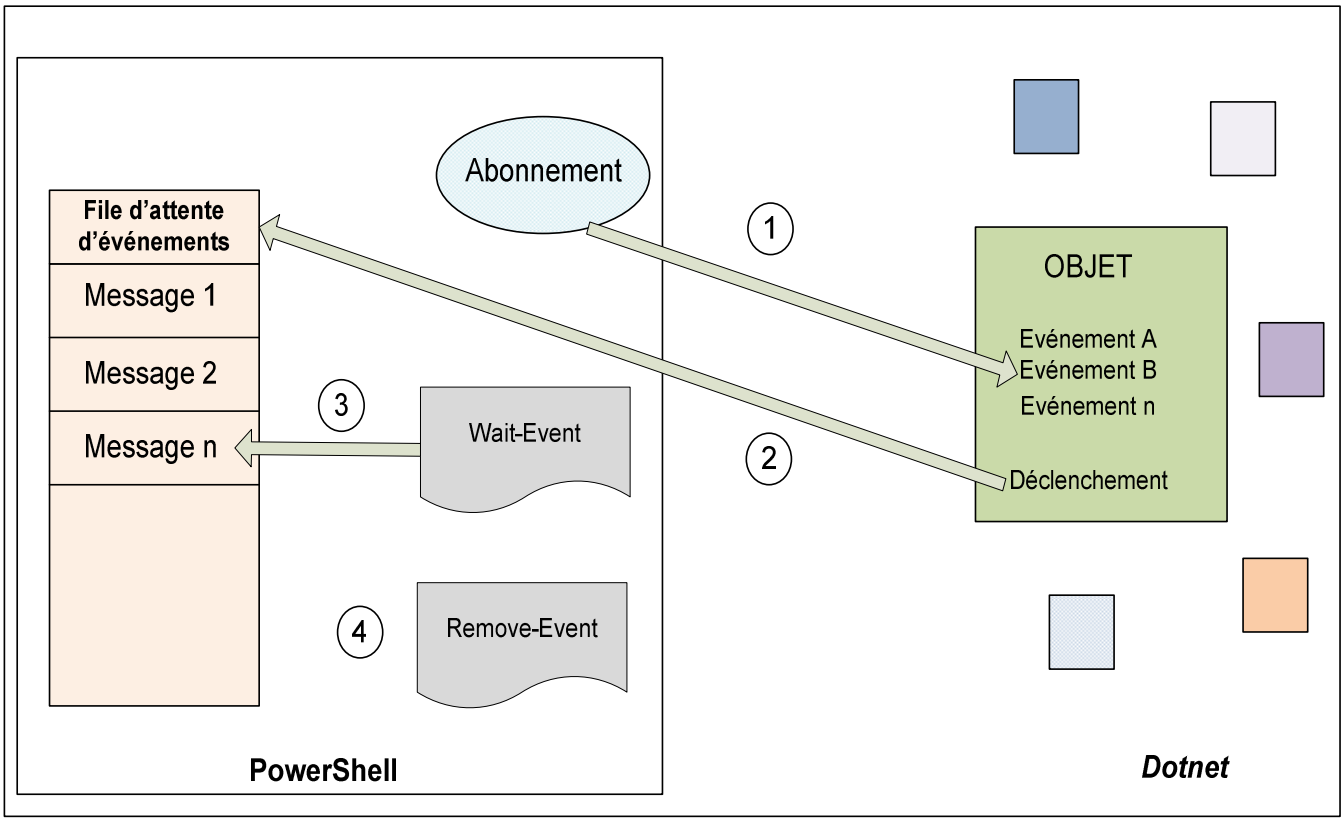
En 3 PowerShell gère en interne l'arrivée de cet événement :

en 3a il traite les événements dans l'ordre de leurs arrivées,

en 3b il exécute, au sein d'un job dédié, le traitement associé à l'événement. Il renseigne également les variables automatiques telles que $Sender,$Event, etc,

en 3c il supprime l'événement de sa file d'attente d'événements.

Voici un résumé des opérations liées à la gestion d'un événement synchrone :



En 1, dans l'environnement PowerShell, on s'abonne à l'événement d'une instance d'objet dotnet.

En 2, dans l'environnement Dotnet, l'objet dotnet déclenche l'événement qui nous intéresse, PowerShell le place dans sa file d'attente d'événements.

En 3, on attend l'arrivée du message qui nous intéresse. Au lieu d'attendre, on peut également, lorsqu'on le souhaite, lire la file d'attente à l'aide du cmdlet Get-Event. On peut ensuite exécuter ou non un traitement.

En 4, selon les besoins, on supprime ou non l'événement.

5 PowerShell délègue un job

Lors de la création d'un abonnement, le cmdlet **Register-ObjectEvent** crée et renvoie une instance de la classe ***System.Management.Automation.PSEventJob*** :

```
Unregister-event *
$EL=Get-EventLog -list|Where {$_.Log -match "Windows PowerShell"}
$Resultat=Register-ObjectEvent $EL EntryWritten AjoutLog –Action {
 Write-Host "1-Réception nouvel événement" –fore White }
$Resultat
```

```
Id     Name            State      HasMoreData     Location        Command
--     ----            -----      -----------     --------        -------
1      AjoutLog        NotStarted False                           Write-Host "1-Récepti...
```

Le code associé au paramètre *-Action* s'exécute dans un job spécialisé et dédié, c'est pour cette raison que l'émission d'un objet dans le pipeline n'est pas affichée sur la console. Ce job traite les événements les uns à la suite des autres et reste à 'l'écoute' (dans l'état *Running*) une fois le traitement de l'événement terminé, là où un job 'normal' passe à l'état *Terminated*.

Sachez que ce type de job passe de l'état *NotStarted* à l'état *Running* lors de la réception du premier message.

Sous dotnet la gestion d'un événement est basée sur un délégué, le gestionnaire d'événement (EventHandler en anglais) :

```
($EL|gm EntryWritten -T event).Definition
```

```
System.Diagnostics.EntryWrittenEventHandler EntryWritten(System.Object…
```

Sous PowerShell celui n'est plus accessible une fois l'abonnement créé, car la gestion des événements est encapsulée et accessible uniquement via les cmdlets dédiés. De plus, on ne déclare pas de délégué, car le scriptblock précisé par le paramètre *–Action* est automatiquement transformé, en interne, en un délégué du même type que l'événement ciblé.

Si vous souhaitez que le job de votre événement produise un résultat dans le pipeline, vous devrez utiliser le cmdlet **Receive-Job** pour le récupérer. Un exemple simple :

```
Remove-Event *
$J = register-engineEvent ListProcess -action { Get-Process }
New-Event ListProcess
Receive-Job $J
Unregister-Event ListProcess
```

Il reste toutefois possible d'utiliser des variables globales :

```
Remove-Event *
$null=register-engineEvent ListProcess -action { $global:P=Get-Process }
New-Event ListProcess
$P
Unregister-Event ListProcess
```

En gardant à l'esprit les risques que cette dernière approche comporte.

5.1 Gestion des erreurs dans le traitement d'un événement

Etant donné que le code associé au paramètre *-Action* est exécuté dans un job dédié, vous devez porter une attention particulière aux cas d'erreurs :

```
function New-EventLogEntryType([string]$EventLogName="Windows PowerShell")
{ $Event=new-object System.Diagnostics.EventLog($EventLogName)
 $Event.Source="TestEventing"
 $Event.WriteEntry("Test
événement",[System.Diagnostics.EventLogEntryType]::Information)
}

Unregister-event *
$EL=Get-EventLog -list|Where {$_.Log -match "Windows PowerShell"}
$Error.Clear()
$Test=15
Register-ObjectEvent $EL EntryWritten -SourceIdentifier AjoutLog –Action {

 Write-Host "Déclenche une erreur" –fore Green
  $Test=15*2
  $Error.Error() #La méthode n'existe pas
 Write-Host "Fin du traitement" –fore Green

}
$Error
```

```
# « } » de fermeture manquante dans le bloc d'instruction. # voir MS-Connect (PowerShell) le bug n° 556257
$Error.Clear()
New-EventLogEntryType
$Error
$Test
```

```
Déclenche une erreur
15
New-EventLogEntryType
```

Ici on constate que :

- ✓ le second appel à **Write-Host** n'a pas lieu,
- ✓ aucun message d'erreur n'est affiché pour indiquer l'erreur,
- ✓ la variable globale *$Error* ne contient pas d'erreur,
- ✓ la variable *$Test* est locale au job,
- ✓ le second appel à la fonction *New-EventLogEntryType* n'a aucun effet.

Notez également que le code associé à l'événement n'est pas exécuté de manière déterministe, on sait qu'il sera exécuté, mais on ne sait pas exactement quand. L'affichage lié à l'appel du premier **Write-Host** peut donc être différent.

Affichons l'état de notre job *via* l'abonnement :

```
(Get-EventSubscriber AjoutLog).Action
```

```
Id     Name      State   HasMoreData   Location    Command
--     ----      -----   -----------   --------    -------
2      AjoutLog  Failed  False                     ...
```

ou *via* la liste des jobs :

```
Get-Job
```

```
Id   Name      State     HasMoreData   Location    Command
--   ----      -----     -----------   --------    -------
1    AjoutLog  Stopped   False                     Write-Host "1-Récepti...
2    AjoutLog  Failed    False                     ...
```

On constate que notre job, suite à l'erreur, est dans l'état *Failed.*

Il est possible de récupérer l'erreur en utilisant sa propriété *Error* :

```
(Get-Job –ID 2).Error
```

```
L'appel de la méthode a échoué parce que [System.Collections.ArrayList] ne contient pas de méthode nommée « Error ».
```

Ou en utilisant le cmdlet **Receive-Job** :

```
Receive-Job -ID 2 -Keep
```

```
L'appel de la méthode a échoué parce que [System.Collections.ArrayList] ne contient pas de méthode nommée « Error ».
```

```
$Error
```

```
L'appel de la méthode a échoué parce que [System.Collections.ArrayList] ne contient pas de méthode nommée « Error ».
```

Par défaut, les résultats de la tâche sont supprimés lorsqu'ils sont récupérés, la présence du switch *–Keep* les affiche sans les supprimer. Ils ne le seront qu'une fois le job détruit.

La réception du résultat affiche une erreur et l'insertion de son contenu dans la variable globale $Error, à l'identique du cmdlet **Write-Error**. Sachez que le comportement du cmdlet **Receive-Job** reste dépendant de la valeur de la variable ***$ErrorActionPreference*** du contexte dans lequel il est exécuté.

Notez que le contexte du module peut être interrogé une fois l'événement traité :

```
Get-EventSubscriber AjoutLog |
 Foreach {
 & (Get-Job -Name $_.SourceIdentifier).Module {
     Write-host "Affiche des variables déclarées dans le contexte du module" –fore White
     $event, $eventSubscriber, $eventArgs}
 }
```

5.2 Suppression du job lié à un abonnement d'événement

On remarque aussi que les appels à **Unregister-Event** ne suppriment pas le job créé par un abonnement utilisant le paramètre *–Action*, on doit utiliser le cmdlet **Remove-Job** :

```
Unregister-event *
$EvtJob=Register-ObjectEvent $EL EntryWritten -SourceIdentifier AjoutLog `
```

```
-Action { Write-Host "Test $(Get-Date)" -fore Green }
New-EventLogEntryType
```

```
Test 01/01/2010 01:01:00
```

```
Remove-Job $EvtJob.id
```

```
Remove-Job : La commande ne peut pas supprimer la tâche avec l'identificateur de session 3, car elle n'est pas terminée.
```

```
Get-Job $EvtJob.id
```

```
Id   Name       State      HasMoreData     Location        Command
--   ----       -----      -----------     --------        -------
3    AjoutLog   Running    True                            Write-Host "1-Récepti...
```

Notre job, enfin le traitement de notre abonnement, est toujours actif, arrêtons son exécution :

```
Get-EventSubscriber
Stop-Job $EvtJob.id
$EvtJob
```

```
Id   Name       State      HasMoreData     Location        Command
--   ----       -----      -----------     --------        -------
3    AjoutLog   Stopped    True                            Write-Host "1-Récepti...
```

```
New-EventLogEntryType #Aucun affichage
Get-EventSubscriber #Aucun affichage
Get-Event #Aucun affichage
Remove-Job $EvtJob.id
```

Notez que le cmdlet **Stop-Job** arrête le job, le passe à l'état *Stopped*, tout en supprimant l'abonnement. Ce comportement est identique à l'appel du cmdlet **Unregister-Event *AjoutLog***. L'appel de l'un ou de l'autre doit de toute façon être complété par un appel à **Remove-Job.**

Notez également que la création d'une nouvelle entrée dans l'eventlog ne déclenche pas d'événement, mais si on se réabonne :

```
$EvtJob=Register-ObjectEvent $EL EntryWritten -SourceIdentifier Test -
Action { Write-Host "Test $(Get-Date)" -fore Green }
New-EventLogEntryType
```

```
Test  01/01/2010 01:01:00
Test  01/01/2010 01:01:00
```

On remarque que l'appel à la fonction *New-EventLogEntryType*, alors que le job était terminé, à bien généré un événement dans le système. Ce comportement est du, il me semble, à la persistance de l'instance $EL entre l'abonnement et le désabonnement (un autre bug ?).

Si vous lui affecter $Null ou la recréer, ce problème, cette «rémanence d'événement», ne se produit plus :

```
#$EL=$Null
$EL=Get-EventLog -list|Where {$_.Log -match "Windows PowerShell"}
…
```

Afin de vous éviter de fastidieux moments de debug, je ne peux que vous recommander de mettre en place une gestion des erreurs et de tester les cas d'erreurs.

Tous les codes des exemples précédents devront être complétés pour supprimer les jobs associés.

5.3 Le gestionnaire des événements

Le gestionnaire des événements de PowerShell est porté par la variable *$ExecutionContext*, propriété *Event* :

```
$ExecutionContext.Events
```

```
Subscribers                    ReceivedEvents
-----------                    --------------
{EntryWritten}                 {}
```

```
wp $ExecutionContext.Events
```

```
ReceivedEvents :
Subscribers      : System.Management.Automation.PSEventSubscriber
```

```
$ExecutionContext.Events.GetType()
```

```
IsPublic IsSerial Name                 BaseType
-------- -------- ----                 --------
False    False    PSLocalEventManager  System.Management.Automation.PSEventManager
```

Cet objet héberge la liste des abonnés et des événements qui ont été déclenchés, ainsi que des méthodes d'accès direct telles que *GenerateEvent*, semblables au cmdlet **New-Event**, les contrôles en moins. Sachez qu'il existe également une classe PSRemoteEventManager.

5.4 Portée des variables

Dans le répertoire *ScriptTimerPortée* vous trouverez trois scripts mettant en évidence l'usage, au sein d'un job lié à un abonné d'événement, des différents types de portées de variables.

Exécutez le script ***Main.ps1***

Vous constaterez que toutes les variables de différentes portées sont persistantes tout au long de l'exécution du job. Et bien que celui-ci soit exécuté dans la session courante, les variables de portée **Script** sont uniques dans chaque script. Seule la variable globale est partagée par chaque job.

Le script suivant met en œuvre l'accès au code interne d'un module à partir du scriptblock d'un abonnement d'événement :

http://www.nivot.org/2009/08/19/PowerShell20AConfigurableAndFlexibleScriptLoggerModule.aspx

6 Event forwarding

Le forwarding peut être traduit par réexpédition, cette mécanique est prise en charge par WinRM. Elle permet d'informer une machine cliente **A** qu'un événement a eu lieu sur une ou plusieurs machines distantes **B, C**.

Pour notre exemple nous déclarons dans la session principale un abonnement sur un événement du moteur PowerShell :

```
 #Exécutez une nouvelle session de PowerShell.exe
Register-EngineEvent –SourceIdentifier Exited –Action  {
 Write-host "Fin de process issu de la session secondaire." }
Get-EventSubscriber
```

Puis nous créons une nouvelle session sur le même poste, la session créée est considérée comme distante :

```
$session = New-PSSession -ComputerName localhost
Enter-PsSession $session
```

```
 #Notez que le prompt change en :
#[localhost] :C:\ >
Get-EventSubscriber
```

```
#ras
```

On constate que la file d'attente de message de la session en local n'est pas couplée avec celle de la session principale de PowerShell, ce qui est normal puisqu'on crée un nouvel environnement complet du moteur d'exécution de PowerShell.

La différence est que la session est démarrée dans un autre process que celui de PowerShell :

```
Get-Process -id $pid |select id,Name
 #Retour à la session principale
Exit
Get-process -id $pid |select id,Name
 #Activation de la seconde session
Enter-PsSession $session
```

*Lorsqu'un ordinateur local se connecte à un ordinateur distant, la gestion des services Web (WS-Management+WinRm) établit une connexion et utilise un plug-in pour Windows PowerShell afin de démarrer le processus hôte Windows PowerShell (**WsmProvHost.exe**) sur l'ordinateur distant.*

L'instruction suivante n'affiche pas l'application Notepad.exe :

```
$P=Microsoft.PowerShell.Management\start-process Notepad.exe –pass
$P
```

Pourtant, le process a bien été créé, ce comportement est dû au process WsmProvHost.exe qui ne permet pas d'interagir avec le bureau :

```
[Diagnostics.Process]::GetCurrentProcess()|Select name,MainWindowHandle
```

```
Name            MainWindowHandle
----            ----------------
Wsmprovhost     0
Test-path Variable:PsWindowHandle
```

```
False
```

Ce plugin ne gère pas de handle de fenêtre, ce qui fait que tous les objets en nécessitant un ne pourront pas fonctionner avec **ce** plugin.

Voir *WinRM Plug-in API* : http://msdn.microsoft.com/en-us/library/dd891129(VS.85).aspx

Testons le réacheminement de l'événement, cette fois en s'abonnant à un événement d'une instance :

```
#[localhost]:… >
Register-ObjectEvent $P Exited -SourceIdentifier Exited `
-Action { Write-host "Fin de process dans la session secondaire." } `
-forward
Register-ObjectEvent : Cette action n'est pas prise en charge lors du transfert d'événements.
```

Le paramètre *–Forward* ne peut pas être déclaré si le paramètre *-Action* existe déjà.

```
Register-ObjectEvent $P Exited -SourceIdentifier Exited –forward
exit
```

L'utilité du paramètre *-Forward* est d'émettre l'événement vers l'ordinateur qui a ouvert la session et c'est celui-ci qui prend en charge l'exécution d'une action lors du second déclenchement de l'événement. On pourrait parler de réception, mais c'est WS-Man qui réinjecte l'événement dans la file d'attente de messages de PowerShell, c'est pour cette raison que l'instance de l'événement contient un champ nommé *Forwarded*, on peut ainsi savoir pour un même événement s'il a eu lieu dans la session principale ou sur un ordinateur distant.

La différence est que les objets sont désérialisés, c'est-à-dire que ses membres ne sont plus associés directement à l'objet existant sur le poste distant, cet objet est en quelque sorte figé.

A ce sujet vous pouvez consulter le post *How objects are sent to and from remote sessions* :

http://blogs.msdn.com/b/powershell/archive/2010/01/07/how-objects-are-sent-to-and-from-remote-sessions.aspx

Si voulez interagir avec l'objet déclencheur, *$Sender*, celui-ci doit posséder un identifiant unique, par exemple la propriété ***Id*** pour un objet de type *Process* :

```
Invoke-Command $session1 {Stop-Process -id $P.Id}
```

Le cmdlet **Invoke-Command**, dédié au remoting, permet d'exécuter une commande dans une session distante.

Vous noterez que le nom de l'identifiant d'abonnement de chaque session, ***Exited,*** est identique et que l'affichage se fait sur la console de la session principale (dans notre exemple les deux sessions partagent la même fenêtre).

Il est possible de retrouver, dans le scriptblock du paramètre *–Action*, l'émetteur de l'événement, et ce, pour une session locale ou distante :

```
 # session distante sur la même machine
$Event.RunspaceId  #correspond à $Session.InstanceId

 # Session distante sur machine différente.
$Sender.MachineName
```

7 Événement WMI

Pour s'abonner aux événements WMI on utilisera le cmdlet **Register-WmiEvent** :

```
Register-WMIEvent -query "Select * From __InstanceCreationEvent within 3
Where TargetInstance ISA 'Win32_Process'" `
-sourceIdentifier "NewProcess" `
–action {Write-Host "A new process has started."}
Microsoft.PowerShell.Management\start-process Notepad.exe –pass
```

L'événement reçu est du type *EventArrivedEventArgs :*

http://msdn.microsoft.com/en-us/library/system.management.eventarrivedeventargs_members(v=VS.85).aspx

Pour retrouver la liste des classes d'événement, vous pouvez utiliser ces instructions :

```
Get-WmiObject -list -namespace "root"  –recurse|
 Where {$_.__Derivation -contains "__EVENT"}
```

Notez que chaque espace de nom contient une suite d'événements identique.

Sur ce cmdlet vous pouvez également consulter le tutoriel suivant :

http://www.microsoft.com/technet/scriptcenter/topics/winpsh/events.mspx

Voir aussi :

Auto mount/unmount new PSDrives for removable drives and network shares :

http://www.nivot.org/2008/08/16/AutoMountunmountNewPSDrivesForRemovableDrivesAndNetworkSharesInPowerShellV2.aspx

Comment surveiller une modification de clé de registre ?

http://blogs.technet.com/heyscriptingguy/archive/2009/11/05/hey-scripting-guy-november-5-2009.aspx

Utiliser WMI avec PowerShell :

http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/Utiliser-WMI-avec-PowerShell/

8 Api Windows

Le cmdlet **Add-Type** offre la possibilité, au travers de code C# compilé à la volée, de manipuler des APIs Win32 afin d'envoyer de messages Windows natifs vers des fenêtres. On peut utiliser une API dédié à l'envoi d'un certain type de message prédéfini :

```
$signature = @"
using System;
using System.Runtime.InteropServices;
 namespace Apis {
   public class Windows {
    [DllImport("user32.dll")]
    public static extern bool ShowWindowAsync(IntPtr hWnd, int nCmdShow);

    [DllImport("user32.dll")]
    public static extern IntPtr SendMessage(IntPtr hWnd, uint Msg, IntPtr
    wParam, IntPtr lParam);
  }
 }
"@

Add-Type -TypeDefinition $signature

 # Minimise la fenêtre de la console PowerShell
[Apis.Windows]::ShowWindowAsync($PsWindowHandle, 2)
Sleep 2
#Puis la restaure
[Apis.Windows]::ShowWindowAsync($PsWindowHandle, 4)
```

Ou envoyer un message directement dans la queue de message d'une fenêtre :

```
$Process=Microsoft.PowerShell.Management\start-process Notepad.exe –pass
Sleep 2
 #Message à envoyer (codé en hexadécimal)
$WM_SYSCOMMAND = [int32] 0x112
 # Information additionnelle du message
 #Ferme la fenêtre
$SC_CLOSE = [int32] 0xF060
 #$Process.MainWindowHandle est le handle de fenêtre
[Apis.Windows]::SendMessage($Process.MainWindowHandle, $WM_SYSCOMMAND,
$SC_CLOSE, [intptr]::Zero)
```

Pour manipuler un message référençant une chaîne on doit au préalable la convertir dans le format attendu par l'API :

```
$Process=Microsoft.PowerShell.Management\start-process Notepad.exe –pass
Sleep 2
 #Change le texte du titre d'une fenêtre
$WM_SETTEXT=[int32] 0x000C
$Msg = "Mon nouveau titre"
Try {
 #convertion de String vers intPtr
 $Ptr= [System.Runtime.InteropServices.Marshal]::StringToHGlobalAnsi(
       $Msg)
 [Apis.Windows]::SendMessage($Process.MainWindowHandle, $WM_SETTEXT,
  [intptr]::Zero, $ptr)
} Finally {
    #Libération de la mémoire non-managée
  [System.Runtime.InteropServices.Marshal]::FreeHGlobal($ptr)
}
```

8.1.1 Winform

L'usage d'événements PowerShell couplé à ceux d'une Winform est tout à fait possible, bien que les événements des composants graphiques soient gérés par la boucle de message de la Winform.

Voici quelques scénarios de test et les comportements associés que l'on peut retrouver avec un tel couplage.

8.1.1.1 Scénario 1

..\Winform\Event-OutWinform.ps1

On s'abonne à l'événement *Exited* d'un process Notepad.exe puis on affiche une fenêtre par l'appel à ShowDialog().

Ensuite on déclenche manuellement l'événement *Exited* en terminant le process Notepad.exe.

Tant que la fenêtre Windows est active l'événement *Exited* est en attente de déclenchement et ce jusqu'a ce qu'une portion de code PowerShell soit déclenchée via événement de la Winform, par exemple par le bouton "Fermer". Une fois celui-ci déclenché, la Winform reste figée quelques instants le temps que le traitement de l'événement *Exited* soit terminé.

La boucle de message de la Winform ne traite donc pas la file d'attente de PowerShell.

8.1.1.2 Scénario 2

..\Winform\Event-INWinform.ps1

On s'abonne à l'événement *Exited* d'un process Notepad.exe puis on affiche une fenêtre par l'appel à ShowDialog().

Ensuite dans le scriptblock de l'événement OnClick_btnAddData de la Winform, on déclenche l'événement *Exited* en terminant le process Notepad.exe par Stop-Process. Les deux gestionnaires d'événement insèrent des données dans la listbox.

Le traitement de l'événement *Exited* bloque le traitement de l'événement *OnClick_btnAddData* de la Winfor tant que le code de l'événement *Exited* n'est pas terminé, une fois ceci fait le traitement de l'événement OnClick_btnAddData reprend là où il s'était arrêté. Notez que pendant le traitement de l'événement *Exited* la Winform ne réagit plus aux demandes.

L'insertion des données dans la listbox se fait correctement, soit par l'un soit par l'autre des deux traitements d'événements (PowerShell et Winform), mais les deux ne s'exécutent pas en parallèle (thread bloquant).

Le mécanisme de gestion de la file d'attente de PowerShell ne gère pas la boucle de message de la Winform.

8.1.1.3 Scénario 3

..\Winform\CallShowDialogInActionScriptblock.ps1

La construction du script pose un problème de portée, pour le régler vous devez copier le contenu de ce script dans la console, notez également que les erreurs déclenchées dans la Winform ne sont plus affichées à l'écran (voir le chapitre « Gestion des erreurs dans le traitement d'un événement »).

On affiche une Winform dans un scriptblock d'un événement personnalisé PowerShell.

La console est inaccessible tant que la Winform est active, c'est le comportement normal, car la méthode ShowDialog() est bloquante.

8.1.1.4 Scénario 4

..\Winform\CallShowDialogInActionScriptblock-V2.ps1

Ici aussi la construction du script pose un problème de portée, pour le régler vous devez copier le contenu de ce script dans la console.

Ce scénario est une combinaison des précédents scénarios 1 et 2.

On constate que tant que la Winform est active, et bien que l'on déclenche un second événement PowerShell en son sein, le second événement reste en attente. Celui-ci sera traité dès que le traitement du premier événement se terminera (en fermant la fenêtre).

Vous pouvez également exécuter les scripts présents dans le répertoire ***..\Winform\Unregister-Event***, ceux-ci tentent de mettre en évidence la gestion des messages restant dans la file d'attente lors de la suppression d'un abonnement.

9 Conclusion

Sans pour autant offrir une gestion complète de tous les types de messages que l'on retrouve sous Windows, PowerShell version 2 offre de nouvelles possibilités de traitement. PowerShell n'ayant pas vocation à remplacer des langages tels le C, Delphi ou le C#, l'apport d'une gestion d'événement constitue un véritable enrichissement. Bien qu'il soit plus facile d'observer et de réagir aux événements que de créer des objets communiquant via des événements.

Pour des développeurs dotnet, ce mécanisme est un peu déroutant étant donné qu'il est indépendant de celui du Framework dotnet, comme quoi PowerShell adapte tout ce qu'il prend en charge. On notera, de nouveau, le manque de documentation décrivant les comportements des mécanismes de PowerShell.

Vous avez pu constater que l'usage de base des événements n'est pas complexe, mais pour des usages avancés cela nécessite d'aborder les jobs, les modules et WinRM (cf. forwarding), ce qui peut freiner l'apprentissage pour un débutant.

Heureusement, et une fois de plus avec PowerShell, il est possible de mettre en œuvre dans vos scripts une gestion d'événements étape par étape.

10 Liens

Article sur les jobs sur le site Hey, Scripting Guy !

http://blogs.technet.com/search/SearchResults.aspx?q=Powershell+job

Rappels sur la file de messages Windows

http://tcharles.developpez.com/simul

About Messages and Message Queues

http://msdn.microsoft.com/en-us/library/ms644927(v=VS.85).aspx