Création de règle PSScriptAnalyzer

Par Laurent Dardenne, le 10/10/2016.

Version 1.2

Conçu avec Powershell version 5.0.10586.117 - Windows Seven 64 bits - PSScriptAnalyzer 1.8

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

Merci à Jean-François L pour ses corrections.

Chapitres

1 Analyse de code à l'aide de règles

PSScriptAnalyzer est un vérificateur de code statique pour les modules et les scripts Windows PowerShell.

Son objectif est de contrôler différents aspects que ce soit le type d'écriture (éviter les blocs d'exception ou bien les variables globales), la conception (utiliser des verbes approuvés) ou encore la sécurité (les mots de passe doivent être des chaînes sécurisés).

Chaque contrôle est basé sur une règle, elle analyse le code *via* l'AST ou des tokens puis, si le code enfreint la règle, renvoie un objet de diagnostique.

Celui-ci peut suggérer des solutions afin de corriger l'erreur ou d'améliorer la qualité du code. On y retrouve également une propriété indiquant le niveau de sévérité de la règle enfreinte, elle peut avoir une des valeurs suivantes :

Information	Le diagnostic est trivial, mais peut être utile.
Warning	Le diagnostic peut causer un problème ou ne pas suivre les directives recommandées par PowerShell.
Error	Le diagnostic est susceptible de causer un problème ou ne suit pas les directives nécessaires de PowerShell.

Lors de tests cette sévérité permettra de filtrer les erreurs selon la gravité souhaitée.

Le module PSScriptAnalyzer est livré avec une collection de règles codées en C#, d'autres peuvent être ajoutées via des modules Powershell.

L'usage du conditionnel dans la description des valeurs n'est pas anodin car le contexte peut inciter à outrepasser une règle, nous verrons par la suite comment le faire.

Note : Le code analysé ne doit pas contenir d'erreur de parsing, si c'est le cas **PSScriptAnalyzer** n'émet pas un objet diagnostique mais déclenche une exception.

1.1 Installation

L'installation de ce module se fait via *Install-Module* :

```
Install-Module PSScriptAnalyzer
Import-Module PSScriptAnalyzer
```

Sa mise à jour à l'aide d'*Update-Module* :

```
Update-Module PSScriptAnalyzer
```

La nouvelle version 1.8 étant signée, elle nécessite quelques opérations supplémentaires.

Pour Windows Seven et Powershell version 5.0, j'ai dû mettre à jour le module **PackageManagement** 1.1.0.0 :

```
Update-Module PackageManagement
```

J'ai ensuite supprimé le répertoire

C:\Program Files\WindowsPowerShell\Modules\PSScriptAnalyzer

Enfin j'ai exécuté l'installation de la nouvelle version du module **PSScriptAnalyzer** de la manière suivante :

```
Install-module PSScriptAnalyzer -SkipPublisherCheck -Repository PSGallery
```

1.2 Documentation d'une règle

Elle est constituée d'un nom, d'un niveau de sévérité, d'une description, d'informations sur sa correction et d'un exemple. La règle suivante vérifie la présence de paramètres utilisant un nom réservé :

1.3 AvoidReservedParams

Severity Level : Error

1.3.1 Description

You cannot use reserved common parameters in an advanced function. If these parameters are defined by the user, an error generally occurs.

1.3.2 How to Fix

To fix a violation of this rule, please change the name of your parameter.

1.3.3 Example

Wrong :

```
function Test {
    [CmdletBinding()]
    Param($ErrorVariable, $b)
}
```

Correct :

```
function Test {
    [CmdletBinding()]
    Param($err, $b)
}
```

Note :

Ici le paramètre *ErrorVariable* est un nom de variable réservé.

Vérifions l'exemple erroné :

```
$Code=@'
function Test
{
    [CmdletBinding()]
    Param($ErrorVariable, $b)
}
'@
```

On appelle cmdlet **Invoke-ScriptAnalyzer** qui vérifie l'ensemble des règles sur un fichier ou une chaîne contenant du code Powershell :

```
Invoke-ScriptAnalyzer -ScriptDefinition $Code | Select-Object *
```

L'objet diagnostique résultant :

```
Line                 : 4
Column               : 11
Message              : 'Test' defines the reserved common parameter
                       'ErrorVariable'.
Extent               : $ErrorVariable
RuleName             : PSReservedParams
Severity             : Warning
ScriptName           :
ScriptPath           :
RuleSuppressionID    :
SuggestedCorrections :
```

On retrouve l'emplacement de l'erreur dans le code, un message décrivant la règle concernée pour cette partie de code ainsi que son nom et sa sévérité. Pour les fichiers on retrouve leur nom et chemin d'accès.

Corrigeons l'erreur :

```
$Code=@'
function Test
{
    [CmdletBinding()]
    Param($err, $b)
}
'@
```

Et exécutons à nouveau **Invoke-ScriptAnalyzer** :

```
Invoke-ScriptAnalyzer -ScriptDefinition $Code
```

Ce code respectant toutes les règles définies par défaut, le cmdlet ne renvoie aucun résultat.

2 Liste des règles

Vous trouverez dans le référentiel Github [la documentation des règles] par défaut.

On peut toutefois les afficher via le cmdlet **Get-ScriptAnalyzerRule** :

```
Get-ScriptAnalyzerRule -Name PSReservedParams | Select-Object *
```

Reprenons la règle précédemment utilisée :

```
RuleName    : PSReservedParams
CommonName  : Reserved Parameters
Description : Checks for reserved parameters in function definitions. If
              these parameters are defined by the user, an error generally
              occurs.
SourceType  : Builtin
SourceName  : PS
Severity    : Warning
```

La propriété *SourceType* indique la provenance de la règle :

- **Builtin** : règle interne
- **Managed** : règle externe codée en C#
- **Module** : règle externe codée dans un module Powershell.

Le code C# utilise [MEF], une bibliothèque d'extension d'application (plugin). La simple copie d'une dll dans le répertoire du module **PSScriptAnalyzer** suffit à sa découverte. Ce qui implique de répéter cette copie en cas de mise à jour du module.

Pour une règle codée dans un module, l'usage du paramètre *–CustomRulePath* sera nécessaire lors de l'appel à **Invoke-ScriptAnalyzer.**

2.1 Restreindre l'analyse

Une règle peut s'appliquer à un type de code Powershell, par exemple un workflow ou une ressource DSC :

```
Get-ScriptAnalyzerRule | Where-Object  SourceName -eq 'PSDSC'
RuleName               Severity      Description          Source
--------               --------      -----------          ------
PSDSCDscTestsPresent   Information   Every DSC resource…  PSDSC
…
```

Ces règles ne se déclencheront que si le type de code analysé correspond, ce que nous confirme l'usage du paramètre *-verbose* :

```
Invoke-ScriptAnalyzer -ScriptDefinition $code -verbose
VERBOSE: Analyzing Script Definition.
VERBOSE: Running PSAvoidUsingCmdletAliases rule.
VERBOSE: Running PSAvoidDefaultValueSwitchParameter rule.
…
```

Les règles dont le *sourceName* est 'PSDSC' ne sont pas exécutées.

Pour exclure une règle on utilisera le paramètre *–ExcludeRule* en précisant le ou les noms des règles à exclure :

```
Invoke-ScriptAnalyzer -ScriptDefinition $code`
                      -ExcludeRule PSReservedParams`
                      -verbose
```

Comme indiqué précédemment il est possible de filtrer les règles sur leur sévérité :

```
Invoke-ScriptAnalyzer -ScriptDefinition $code -Severity Error -verbose
VERBOSE: Analyzing Script Definition.
VERBOSE: Running PSAvoidUsingUserNameAndPassWordParams rule.
VERBOSE: Running PSAvoidUsingConvertToSecureStringWithPlainText rule.
VERBOSE: Running PSAvoidUsingComputerNameHardcoded rule.
```

Au lieu d'une trentaine, seules trois règles correspondent à la sévérité indiquée.

2.1.1 Annuler localement une règle

Certaines règles peuvent se déclencher sur du code correct ou supposé tel dans son contexte. Prenons l'exemple précédent en changeant uniquement le nom de la fonction :

```
$Code=@'
function New-ReportObject
{
    [CmdletBinding()]
    Param($err, $b)
}
'@
Invoke-ScriptAnalyzer -ScriptDefinition $code | Format-List
```

Pour les verbes *New, Reset Restart, Set, Start* et *Stop* **PSScriptAnalyzer** les associe à des actions modifiant l'état du système, ce qui est bien le cas du cmdlet ***Start-Service***, mais pas pour notre fonction.

On peut envisager l'exclusion de la règle mais elle le serait pour la globalité du code analysé lors de l'appel. La solution consiste à marquer la fonction avec l'attribut ***SuppressMessageAttribute***, sa présence annulera localement le déclenchement de la règle.

Ajoutons cet attribut :

```
$Code=@'
function New-ReportObject {
[Diagnostics.CodeAnalysis.SuppressMessageAttribute(
       "PSUseShouldProcessForStateChangingFunctions",
       "",
      Justification="'New-ReportObject' do not change the system state,
only the application 'context'")]
    [CmdletBinding()]
    Param($err, $b)
}
'@
```

Le premier paramètre est le nom de la règle, le second optionnel est un identifiant de la règle qui concaténé avec le premier paramètre forme un identificateur de contrôle unique, le troisième est un texte de justification précisant la ou les raisons de la suppression.

Cette suppression peut être retrouvée à l'aide du paramètre *–SuppressedOnly* :

```
Invoke-ScriptAnalyzer –SuppressedOnly -ScriptDefinition $code | Format-
List
RuleName      : PSUseShouldProcessForStateChangingFunctions
Severity      : Warning
Line          : 1
Column        : 10
Justification : 'New-ReportObject' do not change the system state, only
                 the application 'context'
```

L'annulation peut concerner une règle s'appliquant à un paramètre :

```
$code=@'
function SuppressTwoVariables{

[System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("PSAvoidDefaultV
alueForMandatoryParameter", "b")]
#[System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("PSAvoidDefault
ValueForMandatoryParameter", "a")]
    [CmdletBinding()]
    Param(
         [Parameter(Mandatory=$true)]
        [string] $a="unused",
         [Parameter(Mandatory=$true)]
        [int] $b=3
        )
}
'@
Invoke-ScriptAnalyzer -ScriptDefinition $code|Format-List
```

Dans la déclaration de l'attribut, le premier paramètre est le nom de la règle et le second le nom du paramètre :

```
RuleName : PSAvoidDefaultValueForMandatoryParameter
Severity : Warning
Message  : Mandatory Parameter 'a' is initialized in the Param block.
           To fix a violation of this rule, please leave it unintialized.
```

Une exception est déclenchée si le nom du paramètre n'existe pas :

```
[System.Diagnostics.CodeAnalysis.SuppressMessageAttribute(
 "PSAvoidDefaultValueForMandatoryParameter", "NoExist" )]
Suppression Message Attribute error at line 3 in script definition :
Cannot find any DiagnosticRecord with the Rule Suppression ID NoExist.
```

Le message affiché est un peu déroutant…

2.1.2 Portée de l'annulation

L'annulation du déclenchement d'une règle peut concerner la totalité d'un script ou d'une classe Powershell.

Dans l'exemple suivant on déclare l'attribut en début de script, avant la clause *param()*. En précisant la portée (*Scope*) on cible les deux fonctions définies :

```
$Code=@'
 [Diagnostics.CodeAnalysis.SuppressMessageAttribute(
      "PSUseShouldProcessForStateChangingFunctions","",
      Scope="Function",
      Justification='Annule')]

param()
  function New-ReportObject {
     [CmdletBinding()]
     Param($err, $b)
  }

  function Set-ReportObjet {
     [CmdletBinding()]
     Param($err, $b)
  }
'@
Invoke-ScriptAnalyzer -ScriptDefinition $code|Format-List
```

L'appel ne renvoi aucun résultat.

On peut filtrer sur le nom des fonctions ciblées en ajoutant la propriété d'attribut *Target* :

```
$Code=@'
 [Diagnostics.CodeAnalysis.SuppressMessageAttribute(
      "PSUseShouldProcessForStateChangingFunctions","",
      Scope="Function",
      Target='*Object',
      Justification='Annule')]

param()
  function New-ReportObject {
     [CmdletBinding()]
     Param($err, $b)
  }

  function Set-ReportObjet {
     [CmdletBinding()]
     Param($err, $b)
  }
'@
Invoke-ScriptAnalyzer -ScriptDefinition $code|Format-List
```

Avec ce paramétrage l'annulation ne concerne que la fonction *New-ReportObject* :

```
RuleName : PSUseShouldProcessForStateChangingFunctions
Severity : Warning
Line     : 15
Column   : 12
Message  : Function 'Set-ReportObjet' has verb that could change system
           state. Therefore, the function has to support 'ShouldProcess'.
```

Le contenu de la propriété d'attribut *Target* est une expression régulière, le filtre peut donc être plus précis. Enfin pourra l'être.

Il reste d'autres possibilités, je vous laisse consulter l'aide ou la documentation du Github.

A noter que pour le moment VSCode-Powershell ne peut utiliser les règles custom Powershell.

3 Création de règle Powershell

Voyons maintenant comment créer ses propres règle en utilisant le langage Powershell.

Pour débuter il nous faut des spécifications précises de la règle à implémenter ainsi que des exemples couvrant les différentes écritures que l'on peut rencontrer. Ceux-ci permettront de concevoir les tests de régression qui en cas d'ajout d'un nouveau cas nous assurerons que le code existant fonctionne toujours.

Cette première étape décidera si l'implémentation de la règle est possible et si elle peut être intégrée sous **PSScriptAnalyzer**.

Par exemple, le projet MeasureLocalizedData, contrôlant l'usage des clés de localisation liées au cmdlet ***Import-LocalizedData***, ne peut être intégré dans la version actuelle de PSScriptAnalyzer. La raison étant que ce contrôle se fait en deux passes, on doit d'abord retrouver l'appel du ***Import-LocalizedData*** afin d'extraire les informations nécessaires au traitement, puis parcourir le ou les fichiers utilisant les clés de localisation.

De plus pour chaque fichier traité, le module *MeasureLocalizedData* est chargé dans des runspaces différents, le partage d'informations entre les deux passes s'avère pour le moins problématique.

3.1 Nommer les règles

Il est préférable de nommer une règle pour chaque contrôle effectué, ainsi on peut les désactiver au cas par cas. Dans les noms de règles existants on retrouve les termes suivants :

Avoid, Missing, Provide, Use

L'usage de tels préfixes est similaire à l'usage de verbe pour nommer un cmdlet, ceci dit il n'y a pas à ce jour de norme de nommage à respecter. Vous pouvez vous inspirer des noms de règle de FxCop.

Une évidence à rappeler : Tout changement de nom de règle implique un breaking change.

Note

Une règle codée en C# associe un nom de classe à un nom de règle, les deux noms étant similaires.

En Powershell le moteur utilise le nom de la fonction et l'objet de diagnostique un nom de règle. Ce qui fait que le moteur de **PSScriptAnalyzer** manipule deux termes pour une seule règle.

Si on utilise le nom de la règle pour nommer la fonction on enfreint une des règles de PSScriptAnalyzer, puisqu'on peut l'utiliser pour vérifier le code de notre règle. A mon avis le plus important est de nommer correctement la règle dans l'objet diagnostique.

Le nom de fonctions étant affiché lors de l'usage du paramètre –*Verbose*.

3.2 Optimisation de boucle for

Pour cette exercice je reprends l'optimisation de boucle ***For*** utilisé dans le tutoriel sur l'AST. L'objectif est d'avertir l'utilisateur d'une possible optimisation du code suivant :

```
$Range=1..10
For($i=0; $i -lt $Range.Count; $i++) { $i }
```

La version d'origine permettait la réécriture du code par celui-ci :

```
$Range=1..10
$RangeCount = $Range.Count
For($i=0; $i -lt $RangeCount; $i++) { $i }
```

La version actuelle de **PSScriptAnalyzer** ne permet pas de renvoyer les informations de refactoring, il est possible de détourner une propriété de l'objet *Diagnostic* afin d'y insérer un objet sérialisé sous forme de string, mais cet outil étant là pour promouvoir les bonnes pratiques on va éviter la bricole ;-)

3.2.1 Limites

Dans cette implémentation les trois écritures suivantes sont prises en compte :

```
$i -lt $Range.Count
$i -lt $Range.Count-1
$i -lt ($Range.Count-1)
```

mais pas celles-ci :

```
$i -lt (-1+$Range.Count)
$Range.Count –gt $i
```

Concernant les types d'écriture à couvrir je parle des plus probables, le choix de la sévérité de niveau *Information* suffit et correspond à l'objectif d'optimisation, on peut donc ne pas traiter quelques cas. Il n'existe pas encore de tag permettant de catégoriser une règle, qui autoriseraient différents filtres :

```
@(Tag='Security',Severity= Error), @(Tag='Performance',Severity=Warning)
```

Pour les exemples vous pouvez prendre vos propres scripts ou le répertoire de modules.

3.3 Création du module

On peut le nommer comme on le souhaite, mais comme il n'existe pas de mécanisme de découverte des modules hébergeant des règles, l'ajout du postfixe '***Rule***' pourra aider à les filtrer si besoin.

3.3.1 Prérequis

Avant d'aborder la création d'une règle, il nous faut utiliser un mécanisme de log. L'exécution du code des règles se fait dans un nouveau runspace, l'usage des cmdlets ***Write-Verbose*** et ***Write-Debug*** est inopérant car **PSScriptAnalyzer** ne propage pas leur sortie dans le contexte de l'appelant comme peut le faire un job.

J'ai choisi de réutiliser mon module Log4Posh qui répond parfaitement à ce besoin. Son usage a permis de mettre en évidence ce bug, pour le moment on doit explicitement le charger dans le code du module hébergeant notre règle.

Le manifeste du module doit préciser les dépendances suivantes :

```
RequiredModules=@(
#<DEFINE %DEBUG%>
  @{ModuleName="Log4Posh"; GUID="f796dd07-541c-4ad8-bfac-a6f15c4b06a0";
    ModuleVersion="1.2.0.0"}
#<UNDEF %DEBUG%>
 @{ModuleName="PSScriptAnalyzer"; GUID='d6245802-193d-4068-a631-
8863a4342a18'; ModuleVersion="1.5.0"}
)
```

Ce module de log n'est nécessaire que pour la phase de développement, j'utilise donc cette fonction dans un script de build afin de supprimer cette dépendance lors de la construction de la livraison du module de la règle.

Enfin un dernier point, la version actuelle de **PSScriptAnalyzer** (< =1.8) n'est pas d'un grand secours lorsque notre code provoque une exception, on se retrouve souvent comme une poule devant une fourchette à se demander ce qui se passe, ne sachant si c'est notre code ou le moteur d'analyse qui est en cause.

3.4 Création d'une fonction

Pour chaque règle on crée une fonction et celle-ci peut émettre zéro ou *n* objets de type *DiagnosticRecord.*

Ici aussi il n'existe pas de convention de nommage et le choix d'un verbe approprié au traitement, dans la catégorie Powershell 'Diagnostics', est limité à *Test* ou *Measure* j'ai choisi ce dernier.

Quant au nom de ressource j'ai concaténé le pseudo tag '*Optimize*' et le type AST contrôlé :

```
Function Measure-OptimizeForStatement{
 [CmdletBinding()]
 [OutputType(
  [Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord[]]
 )]
…
```

Une fois ceci fait, on doit déclarer un seul paramètre et son nom **doit se terminer** par ***Ast*** ou ***Token***, ce nommage est imposé par PSScriptAnalyzer, c'est ce qui lui permet de découvrir les règles externes contenues dans un module :

```
 Param(
       [Parameter(Mandatory = $true)]
       [ValidateNotNullOrEmpty()]
       [System.Management.Automation.Language.ForStatementAst]
     $ForStatementAst )
```

Ici le type du paramètre *$ForStatement**Ast*** est le type de la classe Ast que la règle traitera, c'est-à-dire la classe ***ForStatementAst***. Le moteur d'analyse génère l'objet AST, visite chaque type de nœud puis déclenche la ou les règles associées à chaque type AST. Cette fonction ne recevra donc que le type d'objet déclaré et c'est la seule information que notre règle doit connaitre,

Selon les règles on traitera un type AST très précis comme ici ou un type AST englobant d'autres nœuds, dans ce cas c'est la règle qui analysera les nœuds à traiter.

La structure du corps de la fonction se résume au traitement dans un bloc try-catch :

```
process {
  try {
      #Analyse l'AST d'une instruction For()
  } catch  {
     $File= $ForStatementAst.Extent.File
     $ER= New-Object -Typename System.Management.Automation.ErrorRecord`
                     -Argumentlist $_.Exception,
                                   "OptimizeForSatement-$File",
                                   "NotSpecified",$FunctionDefinitionAst
     $PSCmdlet.ThrowTerminatingError($ER)
  }
}#process
```

Ce bloc est très important car en cas de bug on propage le contexte de l'erreur, le nom de la règle en cause et le script en cours d'analyse. Le code de votre règle ne doit pas déclencher d'exception, hormis lors du développement, par exemple des assertions.

3.4.1 Emettre le résultat

La fonction doit renvoyer le résultat du déclenchement d'une règle dans un objet de type DiagnosticRecord. J'ai créé une fonction de création d'un tel objet :

```
Function NewDiagnosticRecord{
 param ($Ast,$Correction=$null)

 $Extent=$Ast.Extent


[Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord]::ne
w(
    $RulesMsg.I_ForStatementCanBeImproved,
    $Extent,
    'ForStatementCanBeImproved',
    'Information',
    $Extent.File,
    $null,
    $Correction
 )
}
```

Les paramètres du constructeur d'un objet *DiagnosticRecord* sont dans l'ordre :

Message	Texte indiquant la raison du déclenchement de la règle
Extent	L'emplacement auquel se réfère cet objet diagnostic
ruleName	Le nom de la règle qui a créé cet objet diagnostic
severity	La sévérité de cet objet diagnostic
scriptPath	Le nom complet du script en cours d'analyse
ruleId	Identifiant de la règle
suggestedCorrections	La correction proposée par la règle pour remplacer le texte indiqué dans la propriété *Extent*

3.4.2 Ajouter une correction de règle

L'ajout d'informations de correction dans un objet diagnostic nécessite la création d'un objet de type CorrectionExtent, sa création réutilise les informations d'*extent* de l'objet AST concerné par la correction :

```
Function NewCorrectionExtent{
 param ($Extent,$Text,$Description)


[Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.CorrectionExtent]::ne
w(
    #Informations d'emplacement
  $Extent.StartLineNumber,
  $Extent.EndLineNumber,
  $Extent.StartColumnNumber,
  $Extent.EndColumnNumber,
   #Texte de la correction lié à la régle
  $Text,
    #Nom du fichier concerné
  $Extent.File,
  $Description  #Description de la correction
 )
}
```

Pour les clés de localisation on peut les préfixer de la manière suivante :

I_	Pour les messages associés à la sévérité *Information*
W_	Pour les messages associés à la sévérité *Warning*
E_	Pour les messages associés à la sévérité *Error*
C_	Pour le texte des corrections

Mais ça c'est la théorie ☺, dans la pratique on ne peut renseigner que les propriétés suivantes :

```
$Diagnostic =
[Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord[]]@{
    'Message'  = $RulesMsg.E_MaRegle
    'Extent'   = $Extent
    'RuleName' = $PSCmdlet.MyInvocation.InvocationName
    'Severity' = 'Warning'
}
```

Lors du traitement du résultat émis par notre fonction, **PSScriptAnalyzer** ne considère pas les propriétés restantes, l'ajout d'un objet correction n'est donc pas encore possible pour une règle si on utilise le langage Powershell.

3.5 Implémenter la règle

Une fois la structure construite, reste à analyser le code reçu afin de déterminer si on déclenche ou non la règle. Ce déclenchement n'étant que l'émission d'un objet de type *DiagnosticRecord*.

Le code complet se trouvant sur github, je détaille rapidement les étapes.

On détermine si la boucle ***for*** possède une condition :

```
if ($null -ne $ForStatementAst.Condition)
{
```

Si c'est le cas on boucle sur les éléments du pipeline de la condition :

```
 foreach ($Node in $ForStatementAst.Condition.PipelineElements)
 {
```

On ne traite que les expressions :

```
  if ($Node -is [System.Management.Automation.Language.CommandExpressionAst])
  {
```

Si l'expression courante est une des trois écritures recherchée :

```
    $Expression=$Node.Expression
   if ($Expression -is [System.Management.Automation.Language.BinaryExpressionAst])
    {
```

On déclenche la règle :

```
      'MemberExpressionAst' {  # cas : $I -le $Range.Count
       NewDiagnosticRecord $RulesMsg.I_ForStatementCanBeImproved`
                           Information`
                           $ForStatementAst
```

Ensuite la fonction se termine.

Pour d'autres règles, par exemple celle-ci qui analyse de possibles erreurs dans la signature d'une fonction, les règles de gestion et les différents contrôles nécessiteront bien plus de code et de tests Pester.

Reste à documenter les règles, comme indiqué au début de ce document, à l'aide fichier de balisage Markdown. La documentation de la fonction peut être succincte, le renvoi vers une documentation en ligne me semble suffisant.

3.6 Debugger la règle

La mise au point d'une règle exécutée par PSScriptAnalyzer ne peut se faire à l'aide d'un debugger, car son code est exécuté dans un autre contexte. On peut toutefois utiliser l'AST afin d'exécuter le code dans le même contexte mais sans utiliser **PSScriptAnalyzer**.

On charge le module de la règle puis on récupère l'AST du code à l'aide de cette méthode :

```
$M=Import-Module OptimizationRules -Pass

 #https://gallery.technet.microsoft.com/scriptcenter/AST-Module-eecb5a64
$Ast=Get-AST -Path $M.Modulebase

$Predicate= {
 $args[0] -is [System.Management.Automation.Language.ForStatementAst]
}
$Result=$Ast.FindAll($Predicate, $true)
```

Reste à exécuter la fonction de la règle :

```
Measure-OptimizeForStatement $Result[0]
```

3.7 Exécuter la règle

Je suppose le module installé dans un des répertoires de module :

```
$Path=(Import-Module OptimizationRules -pass).Modulebase
 #Exécute la régle sur le module de la régle
Invoke-ScriptAnalyzer -Path $Path -CustomRulePath $Path
```

Il reste un défaut, bien que l'on héberge une ou des règles dans un module on doit préciser un chemin pour que le moteur retrouve la règle. Ceci serait tout de même préférable et éviterait une dépendance sur un nom de chemin :

```
Invoke-ScriptAnalyzer -CustomRuleModule OptimizationRules
 #ou
Invoke-ScriptAnalyzer -CustomRuleModule @{
                        RootModule = 'OptimizationRules';
                        ModuleVersion = '0.2.0';
                        GUID = '607465cd-bec6-46e4-81e5-768701bf35a0'
                      }
```

Ce point devrait évoluer dans une prochaine version.

4 Conclusion

Je trouve cet outil très pratique notamment en tant que pense-bête, car bien que l'on connaisse tout ou partie des règles que cet outil vérifie, il arrive, pour de multiples raisons, que l'on ne les respecte pas lors de l'écriture de code. Il automatise donc certaines vérifications fastidieuses et permet de révéler certaines de nos incohérences. Il y a dans son usage une ressemblance avec celui du correcteur de Word ;-)

Il est utile aux débutants/es en leur rappelant les bonnes pratiques d'écriture de code sous Powershell. Pour les plus avancés son intégration dans un mécanisme de construction/livraison de script (IC, build) coule de source.

On peut aussi envisager en cas de *breaking change* des règles d'analyse spécifiques à ce changement. Ce n'est pas les idées qui manquent, juste le temps pour les coder ;-)

Cependant, il reste certains points à améliorer, notamment la gestion des exceptions, les performances et le codage de règles custom en Powershell. J'ai cité quelques bugs limitant leurs usages, il en existe quelques autres, par exemple l'exclusion de règle en ligne de commande ne fonctionne qu'avec des règles codée en C# (notez que l'usage de l'attribut *SuppressMessage* fonctionne).

Les prochaines versions devraient corriger ces bugs et lacunes. Pour le moment je n'ai trouvé que quelques règles sur Github et aucune sur Powershell Gallery.

Réglez nous ça ☺