PowerShell DSC Settings Management

A SQL database-backed management solution of DSC settings

Document Owner: Gary M Wells

Issue Date: 08 September 2017

Version: 0.3

Classification: INTERNAL

Contents

[1. Introduction 5](#_Toc492627157)

[1.1. Overview 5](#_Toc492627158)

[1.2. Audience 6](#_Toc492627159)

[1.3. Approach 6](#_Toc492627160)

[2. Desired State Configuration (DSC) 7](#_Toc492627161)

[2.1. What is DSC? 7](#_Toc492627162)

[2.2. Benefits 7](#_Toc492627163)

[2.3. Practical Applications 7](#_Toc492627164)

[2.4. Key Concepts 8](#_Toc492627165)

[2.4.1. Configurations 8](#_Toc492627166)

[2.4.2. Resources 9](#_Toc492627167)

[2.4.3. Local Configuration Manager 10](#_Toc492627168)

[2.5. Components 11](#_Toc492627169)

[2.5.1. Windows Management Framework (WMF) 11](#_Toc492627170)

[2.5.2. DSC Pull Server 11](#_Toc492627171)

[2.5.3. DSC Pull Client 11](#_Toc492627172)

[2.5.4. Powershell 11](#_Toc492627173)

[2.5.5. Managed Object Format (MOF) file 11](#_Toc492627174)

[2.5.6. SQL Server 12](#_Toc492627175)

[2.5.7. DSCManagement Module 12](#_Toc492627176)

[3. Solution Implementation 13](#_Toc492627177)

[3.1. SQL 13](#_Toc492627178)

[3.1.1. Create the SQL Repository 13](#_Toc492627179)

[3.1.2. Populate the SQL Tables 13](#_Toc492627180)

[3.2. Pull Server 14](#_Toc492627181)

[3.2.1. Requirements for using a pull server: 14](#_Toc492627182)

[3.2.2. Install Pull Server Services 14](#_Toc492627183)

[3.2.3. Set Up Pull Server 15](#_Toc492627184)

[4. Manage Configurations 22](#_Toc492627185)

[4.1. XML Import 22](#_Toc492627186)

[4.2. Ongoing Maintenance 23](#_Toc492627187)

[5. Generate MOF 24](#_Toc492627188)

[5.1. New-DscMOF Function 24](#_Toc492627189)

[5.1.1. Parameters 24](#_Toc492627190)

[5.1.2. Operation 24](#_Toc492627191)

[5.2. Update-ConfigBlock Function 25](#_Toc492627192)

[5.2.1. Parameters 25](#_Toc492627193)

[5.2.2. Operation 25](#_Toc492627194)

[6. Deploy MOF 26](#_Toc492627195)

[7. Appendix 27](#_Toc492627196)

[7.1. Function Overview 27](#_Toc492627197)

[7.1.1. Open-SqlConnection 27](#_Toc492627198)

[7.1.2. Close-SqlConnection 27](#_Toc492627199)

[7.1.3. Get-DscDBTables 27](#_Toc492627200)

[7.1.4. Get-DscSettings 28](#_Toc492627201)

[7.1.5. Initialize-Table 29](#_Toc492627202)

[7.1.6. New-DBTableForDSCMetadata 29](#_Toc492627203)

[7.1.7. New-DBTableFromResource 30](#_Toc492627204)

[7.1.8. New-DscMOF 33](#_Toc492627205)

[7.1.9. Open-DSCSettings 34](#_Toc492627206)

[7.1.10. Update-ConfigBlock 36](#_Toc492627207)

[7.1.11. Update-ConfigBlockData 38](#_Toc492627208)

[7.1.12. Remove-Artefacts 38](#_Toc492627209)

[7.1.13. Add-DscChangeRecord 38](#_Toc492627210)

[7.1.14. Add-DscMultiValueColumn 39](#_Toc492627211)

[7.2. Sample XML 40](#_Toc492627212)

[7.3. Sample DSC Configuration Script created in memory 40](#_Toc492627213)

[7.4. Example of MOF content 40](#_Toc492627214)

Document Information

Revision History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date** | **Version** | **Status** | **Prepared by** | **Comments** |
| 2nd August 2017 | 0.1 | Draft | Gary Wells | Initial release for review |
| 4th September 2017 | 0.2 | Draft | Andrew Willows / Andrew Auret | Review and comment |
| 8th September 2017 | 0.3 | Draft | Gary Wells | Added additional content provided by Andrew Willows and Andrew Auret |

Document Control

|  |  |  |
| --- | --- | --- |
| **Role** | **Name** | **E-mail** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Approval

|  |  |  |  |
| --- | --- | --- | --- |
| **Role** | **Name** | **Signature** | **Sign-off Date** |
|  |  |  |  |
|  |  |  |  |

# Introduction

## Overview

This project grew from the need to maintain a large number of settings that were to be applied by DSC. With the advent of additional Platforms, some of these settings could have multiple values applied based on whether the Server was in ICP, GCP or other environments.

It is possible to use a flat file method to manage the settings, however a different file will have to be produced for each OS on each platform, with each application set installed – potentially a custom MOF file per server, which then makes it an onerous task if a large number of servers need to have a value changed.

So while flat files can be used to store the settings in CSV or XML format, the use of a SQL database allows for tighter control over the settings, avoidance of duplication and elimination of input errors.

The use of a SQL database also allows for the storage of settings that may not be uniform in their use of properties. For example, a WindowsFeature has a number of properties that are optional as below:

CoreID : 1

CorePlatform : BaseOS

CoreDescription : Add the Telnet Client to base builds

Name : MyTelnetClient

Ensure : Present

IncludeAllSubFeature :

LogPath :

Source : C:\Software\MTC

CoreID : 2

CorePlatform : AzureCloud

CoreDescription : Add IIS

Name : IIS

Ensure : Absent

IncludeAllSubFeature : Yes

LogPath :

Source :

As can be seen from the above examples, some properties are not populated, even though they could be. The “IncludeAllSubFeature” property, for example, requires a YES value if it is TRUE, but does not require a NO value to indicate FALSE

This difference in property use can present issues when trying to build dynamic configurations as an attempt to read a property will fail for those that have no value.

The PowerShell DSC module addresses this by building configuration data based only on the fields that are populated and then grouping them using the same properties for each resource.

## Audience

This document is for use by HSBC project Stakeholders, project sponsors, technical decision makers and technical specialists to determine if the proposed solution meets the requirements of the business as well as the requirements of the project.

## Approach

Desired State Configuration is already used within the HSBC estate. As such, we attempted to satisfy the desire to “use existing tooling where possible” and the capabilities of DSC mean that it can perform additional roles in future, such as install software packages

The approach to developing this DSC Management solution was based on expanding the current PowerShell DSC environment. This SQL-backed solution allows for automating MOF file creation, based upon the fixed parameters dictated by various security standards and then modifying specific parameters based on which platform they operate upon (ICP, GCP, AWS etc.)

This will generate a specific MOF that is built in layers:

* Specific OS Settings – 2012R2, 2016 Core, 2016 GUI
* Primary Security Standards
* Secondary Security Standards
* Platform-specific settings – ICP, GCP, AWS, AZURE

Platform-specific settings will be applied last. All other settings will already be in the MOF and only the minor customisations for platform-specific settings will be overwritten, such as NOT installing NetBackup client, or NOT installing cloud-specific management tools

# Desired State Configuration (DSC)

## What is DSC?

DSC is a new management platform in Windows PowerShell that enables deploying and managing configuration data for software services and managing the environment in which these services run.

DSC provides a set of Windows PowerShell language extensions, new Windows PowerShell cmdlets and resources that you can use to declaratively specify how you want your software environment to be configured. It also provides a means to maintain and manage existing configurations.

## Benefits

As DSC is based around PowerShell, no additional languages or syntax needs to be learned. It uses the same PowerShell that System Administrators and Engineers are used to using on a daily basis to manage / maintain / troubleshoot Windows Server environments

* Declaring a DSC configuration is PowerShell based. You can leverage all of your existing PowerShell skills to not only define a configuration, but also for troubleshooting.
* DSC is designed to support “Continuous Deployments” which means that you can deploy your configuration over and over without breaking anything
* When a DSC configuration is being applied, only those settings which do not match will be set. The rest will be skipped which can result in a faster deployment time
* You can separate the configuration data from the logic of your configuration so that you can reuse your configuration data for different resources, nodes, and configurations - see <http://technet.microsoft.com/en-us/library/dn249925.aspx>
* DSC can be used on-premise, in a public or in a private Cloud environment. You just need Windows Server 2012 R2 and local administrator permissions to execute the DSC PowerShell scripts
* You can integrate DSC with any Microsoft or non-Microsoft solutions as long as you can execute a PowerShell script on the target system. Using DSC within the Windows Azure Pack portal in conjunction with [SMA](http://blogs.technet.com/b/privatecloud/archive/2013/08/14/automation-service-management-automation-getting-started-with-sma-runbooks.aspx) is a good example

## Practical Applications

Following are some example scenarios where you can use built-in DSC resources to configure and manage a set of computers (also known as target nodes) in an automated way:

* Enabling or disabling server roles and features
* Managing registry settings
* Managing files and directories
* Starting, stopping, and managing processes and services
* Managing groups and user accounts
* Deploying new software
* Managing environment variables
* Running Windows PowerShell scripts
* Fixing a configuration that has drifted away from the desired state
* Discovering the actual configuration state on a given node

## Key Concepts

DSC is a declarative platform used for configuration, deployment, and management of systems. It consists of three primary components:

### Configurations

**Configurations** are declarative PowerShell scripts which define and configure instances of resources. Upon running the configuration, DSC (and the resources being called by the configuration) will simply “make it so”, ensuring that the system exists in the state laid out by the Configuration.

DSC configurations are also idempotent: the Local Configuration Manager (LCM) will continue to ensure that machines are configured in whatever state the configuration declares.

To define a configuration, you use the PowerShell keyword **Configuration**. (This keyword is also a function)

Configuration HSBCDscConfiguration

{

Node "TEST-SERV01"

{

WindowsFeature MyFeatureInstance {

Ensure = "Present"

Name = "RSAT"

}

WindowsFeature My2ndFeatureInstance {

Ensure = "Present"

Name = "Bitlocker"

}

}

}

HSBCDscConfiguration

Save the Configuration as a PowerShell script with an extension of .ps1

A configuration script consists of the following parts:

* The **Configuration** block. This is the outermost script block. You define it by using the **Configuration** keyword and providing a name. In this case, the name of the configuration is "HSBCDscConfiguration".
* One or more **Node** blocks. These define the nodes (computers or VMs) that you are configuring. In the above configuration, there is one **Node** block that targets a computer named "TEST-SERV01". Multiple Node blocks can be added to target more than one computer with a single configuration
* One or more **Resource** blocks. This is where the configuration sets the properties for the resource(s) that it is configuring. In this case, there are two resource blocks, each of which call the "WindowsFeature" resource. The example above will ensure that both Remote Server Administration Tools (RSAT) and BitLocker are installed on TEST-SERV01

Within a **Configuration** block, you can do anything that you normally could in a PowerShell function. For example, in the previous example, if you didn't want to hard code the name of the target computer in the configuration, you could add a parameter for the node name:

Configuration HSBCDscConfiguration

{

param(

[string[]]$ComputerName="localhost"

)

Node $ComputerName {

WindowsFeature MyFeatureInstance {

Ensure = "Present"

Name = "RSAT"

}

WindowsFeature My2ndFeatureInstance {

Ensure = "Present"

Name = "Bitlocker"

}

}

}

HSBCDscConfiguration

In this example, you specify the name of the node by passing it as the **ComputerName** parameter when you compile the configuration. The name now defaults to "localhost". This script can then be used on any machine.

### Resources

**Resources** are the imperative building blocks of DSC which are written to model the various components of a sub-system and implement the control flow of their changing states. They reside within PowerShell modules and can be written to model something as generic as a file or a Windows process or as specific as an IIS server or a VM running in Azure.

Windows PowerShell Desired State Configuration (DSC) comes with a set of built-in configuration resources. The following table provides an alphabetical list of these resources and links to topics that describe them in detail.

* [Archive Resource](https://docs.microsoft.com/en-us/powershell/dsc/archiveresource)
* [Environment Resource](https://docs.microsoft.com/en-us/powershell/dsc/environmentresource)
* [File Resource](https://docs.microsoft.com/en-us/powershell/dsc/fileresource)
* [Group Resource](https://docs.microsoft.com/en-us/powershell/dsc/groupresource)
* [Log Resource](https://docs.microsoft.com/en-us/powershell/dsc/logresource)
* [Package Resource](https://docs.microsoft.com/en-us/powershell/dsc/packageresource)
* [Registry Resource](https://docs.microsoft.com/en-us/powershell/dsc/registryresource)
* [Script Resource](https://docs.microsoft.com/en-us/powershell/dsc/scriptresource)
* [Service Resource](https://docs.microsoft.com/en-us/powershell/dsc/serviceresource)
* [User Resource](https://docs.microsoft.com/en-us/powershell/dsc/userresource)
* [WindowsFeature Resource](https://docs.microsoft.com/en-us/powershell/dsc/windowsfeatureresource)
* [WindowsProcess Resource](https://docs.microsoft.com/en-us/powershell/dsc/windowsprocessresource)
* If you need to create additional resources, see [Build Custom Windows PowerShell Desired State Configuration Resources](https://docs.microsoft.com/en-us/powershell/dsc/authoringresource)

### Local Configuration Manager

The Local Configuration Manager (LCM) is the engine by which DSC facilitates the interaction between resources and configurations. The LCM regularly polls the system using the control flow implemented by resources to ensure that the state laid out by a Configuration is maintained. If the system is out of state, the LCM uses more logic inside of the resources to “make it so” according to the Configuration declaration.

The **Set-DscLocalConfigurationManager** cmdlet applies Local Configuration Manager (LCM) settings, or **meta-configuration**, to nodes. Specify computers by specifying computer names or by using Common Information Model (CIM) sessions. If you do not specify a target computer, the cmdlet applies settings to the local computer.

Set-DscLocalConfigurationManager -Path "C:\DSC\Configurations\"

The **Get-DscLocalConfigurationManager** cmdlet gets Local Configuration Manager (LCM) settings, or meta-configuration, and the states of LCM for the node. Specify computers by using Common Information Model (CIM) sessions.

$Session = New-CimSession -ComputerName "TEST-SERV01" -Credential DOMAIN\USERNAME

Get-DscLocalConfigurationManager -CimSession $Session

If you do not specify a target computer, the cmdlet gets the configuration settings from the local computer.

Get-DscLocalConfigurationManager

## Components

### Windows Management Framework (WMF)

DSC is an integral component of the Windows Management Framework (WMF). DSC was introduced with WMF version 4 and has been enhanced and expanded with each subsequent version of WMF. Current version of WMF is 5.1

WMF is a core part of Windows Operating Systems and requires no additional application or agent to be installed or configured

### DSC Pull Server

A central repository is required that will store the Configurations. The Pull Server resource is installed on a Windows Server 2012R2 server environment. Servers will contact the Pull Server to obtain their initial settings, and also check periodically for any changes to the Configurations that would require a new file to be downloaded and applied

### DSC Pull Client

The DSC Pull Client is configured on each Server to be managed with DSC. The client can be configured automatically with PowerShell and is responsible to collecting Configurations from the Pull Server

### Powershell

DSC runs using PowerShell. The PowerShell DSC module includes a number of new language keywords, cmdlets and tools that allow:

* Creation of configurations
* Build of DSC resources
* Invoking configurations
* Management of the LCM

These cmdlets can be found in Windows as part of the PsDesiredStateConfig module (including Start-DscConfiguration, Set-DscLocalConfigurationManager, and Get-DscResource).

You can add the **Windows PowerShell Desired State Configuration service** feature with the Add Roles and Features wizard in Server Manager, or by using PowerShell.

### Managed Object Format (MOF) file

DSC applies settings and configurations by using a Managed Object Format (MOF) file. This file is generated with all of the required settings and processed on each host, which uses the Local Configuration Manager (LCM) to enforce the configuration(s) within the MOF

This process can be configured to run on a regular basis in order to find and correct settings changes, or just find and report on change, depending on whether the changed setting is within an allowable range

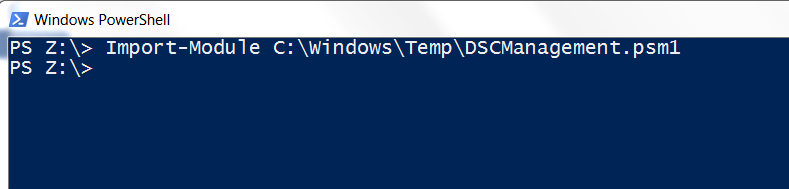
### SQL Server

The proposed solution will employ a SQL Server database to store all of the DSC Configuration elements and the possible values for these in dedicated tables.

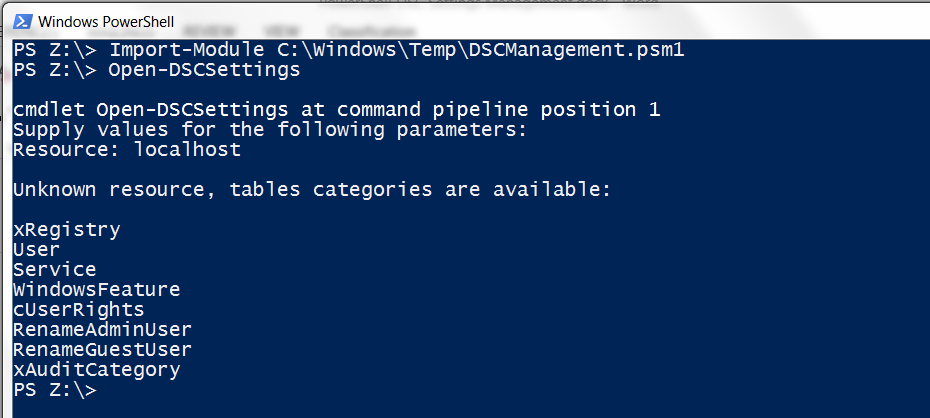
Each Configuration element will have its own table within the database such as UserRights, These are all populated from XML which is easily modified and re-imported should the requirements change and additional controls need to be added, or existing ones removed

### DSCManagement Module

All elements of DSC management will be performed from a single PowerShell MODULE – DSCManagement.psm1. This module contains a number of FUNCTIONS which are used to perform all of the DSC tasks. This module needs to be imported to the operator’s PowerShell environment



Commands and can then be launched by entering the function name



All of the functions within the module are explained in the [Function Overview](#_Function_Overview) section of the Appendix

# Solution Implementation

## SQL

### Create the SQL Repository

A PowerShell script (Create.ps1) exists to create the tables required in an empty database

Import-Module -FullyQualifiedName D:\Scripts\dsc\DSCManagement-master\DSCManagement.psm1 -Force -Verbose -DisableNameChecking

$connection = Open-SqlConnection

New-DBTableForDSCMetadata -connection $connection

$Resources = "xRegistry","User","Service","WindowsFeature","cUserRights","RenameAdminUser","RenameGuestUser","xAuditCategory"

foreach($Resource in $Resources)

{

New-DBTableFromResource -DscResName $Resource

}

Additional tables can be created by adding the names to the $Resources = section if needed, such as “WindowsProcess” or “Environment”

### Populate the SQL Tables

A PowerShell script (PopulateDSC.ps1) exists to populate the required fields in the tables

$Here = "PATH TO XML FOLDER"

$dataSource = "FQDN AND PORT OF SQL SERVER"

$database = "DSC DATABASE NAME"

$connectionString = "Server=$dataSource;Database=$database;trusted\_connection=true;"

$connection = New-Object System.Data.SqlClient.SqlConnection

$connection.ConnectionString = $connectionString

$connection.Open()

$command = New-Object System.Data.SqlClient.SqlCommand

$command.Connection = $connection

$HSBCXml = [xml](Get-Content "$Here\Registry.xml")

# Import Registry Items – THIS IS ONE OF THE XML FILES. OTHER SECTIONS EXIST IN THE SCRIPT FOR OTHER SETTINGS SUCH AS USER RIGHTS

foreach($item in $HSBCXml.Registry.Reg)

{

$command.commandtext = "INSERT INTO xRegistryEntries (KeyName,ValueType,ValueName,ValueData,AcceptedValues,Ensure,CoreDescription,CorePlatform,Mode) VALUES('{0}','{1}','{2}','{3}','{4}','{5}','{6}','{7}','{8}')" -f

$item.Key,$item.ValueType,$item.ValueName,$item.ValueData,$item.AcceptedValues,$item.State,$item.Description,"BaseOS","None"

$command.ExecuteNonQuery()

}

The XML files contain all of the values and parameters required to correctly configure the table

## Pull Server

### Requirements for using a pull server:

A DSC Web Pull Server is a Web service in IIS that uses an OData interface to make DSC configuration files available to target nodes when those nodes ask for them.

Use of a Pull Server requires a server running:

* WMF/PowerShell 5.0 or greater
* IIS server role
* DSC Service
* Some means of generating a certificate, to secure credentials passed to the Local Configuration Manager (LCM) on target nodes

You can add the IIS server role and DSC Service with the Add Roles and Features wizard in Server Manager, or by using PowerShell.

### Install Pull Server Services

A script exists (DSC\_PullServer.ps1) in order to perform this task.

This script installs and configures the WPDSC v5.x Pull Server role on a Windows Server 2012 R2 using Desired State Configuration – excerpt shown below.

configuration CreatePullServer {

Import-DscResource –ModuleName PSDesiredStateConfiguration

Import-DSCResource -ModuleName xPSDesiredStateConfiguration -ModuleVersion 3.12.0.0

# Set role and instance variables

$Roles = $AllNodes.Roles | Sort-Object -Unique

foreach($Role in $Roles)

{

$Servers = @($AllNodes.Where{$\_.Roles | Where-Object {$\_ -eq $Role}}.NodeName)

Set-Variable -Name ($Role.Replace(" ","").Replace(".","") + "s") -Value $Servers

if($Servers.Count -eq 1)

{

Set-Variable -Name ($Role.Replace(" ","").Replace(".","")) -Value $Servers[0]

}

}

Node $AllNodes.NodeName

{

if(($WindowsPowerShellDSCPullServer -eq $Node.NodeName))

{

LocalConfigurationManager

{

AllowModuleOverwrite = $True

ActionAfterReboot = 'ContinueConfiguration'

RebootNodeIfNeeded = $True

ConfigurationMode = "ApplyOnly"

}

WindowsFeature DSCServiceFeature

{

**Ensure = "Present"**

**Name = "DSC-Service**"

}

xDscWebService PSDSCPullServer

{

Ensure = "Present"

EndpointName = "PSDSCPullServer"

Port = $Node.PSDSCPullServer\_Port

PhysicalPath = $Node.PSDSCPullServer\_PhysicalPath

CertificateThumbPrint = $Node.CertificateThumbPrint

ModulePath = $Node.PSDSCPullServer\_ModulePath

ConfigurationPath = $Node.PSDSCPullServer\_ConfigurationPath

**State = "Started"**

DependsOn = "[WindowsFeature]DSCServiceFeature"

#RegistrationKeyPath = $Node.RegistrationKeyPath

AcceptSelfSignedCertificates = $False

}

File RegistrationKeyFile

{

Ensure ='Present'

Type = 'File'

DestinationPath = $Node.RegistrationKeyPath+"\RegistrationKeys.txt"

Contents = $Node.RegistrationKey

}

The LCM on the Pull Server will now test for the elements specified to be in the desired state – ie WindowsFeature DSC-Service is PRESENT and STARTED. If any of these conditions fails, DSC will “make it so” by performing an installation of the DSC-Service and Starting it if it is NOT present, or just Starting it if it IS present, so all conditions in the MOF are met

### ****Set Up Pull Server****

The following steps are required in order to set up a Pull Server

#### Satisfy Prerequisites for Installation

**DSC Files**

Before you start the deployment process, make sure you have all the setup files required for building and configuring DSC

|  |  |
| --- | --- |
| **SetUpFiles\_07.09.2016.zip**  **Contains:** | Zip file with all PowerShell and DSC scripts used to build the environment. |
| * 1\_PullServer\_Setup[Folder] | PowerShell and DSC scripts used to configure the Pull Server |
| * 2\_MOFGeneration [Folder] | PowerShell, DSC scripts and modules to create the Security Standards MOF file. |
| * 3\_PublishMOF [Folder] | PowerShell script to publish modules and MOF files to the correct locations in the proper format required by the DSC pull server. |
| * Start\_Setup\_with\_ISE | Main PowerShell ISE that opens the necessary scripts in Tabular format |
| * Start.cmd | Starts the Main Setup PowerShell script in ISE |

**Certificate**

Ensure that a valid SSL certificate for the DSC Pull server has been obtained from the HSBC Certificate Authority. The certificate received from the authority is usually in the PFX format with the correct private keys. Install the certificate on the node that will become the DSC Pull server in the default location which should be **CERT:\LocalMachine\My**.

**PowerShell Execution Policy**

Ensure that the PowerShell Execution Policy has been set to ***Unrestricted*** for the entire installation process. **Unrestricted** - No restrictions; all Windows PowerShell scripts can be run.

**DNS Entries**

Ensure that the correct **Name Records** have been created to match the certificate ***FriendlyName.*** These Name records must be included for the host names for all deployed pull server instance's.

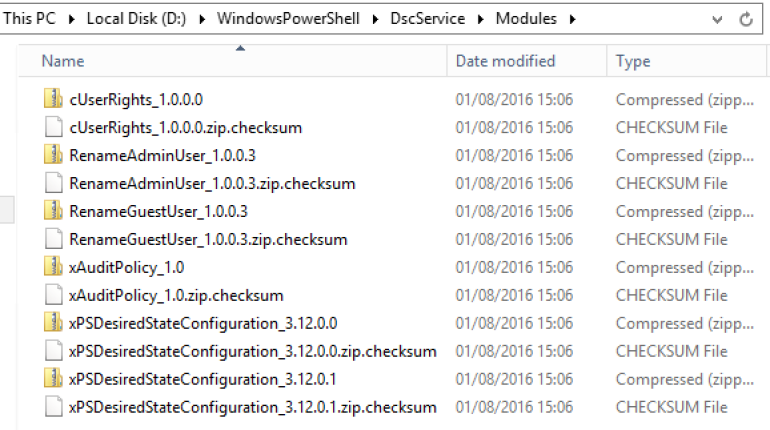
**Server Configuration**

Ensure that the following server configurations have been requested and provisioned for DSC

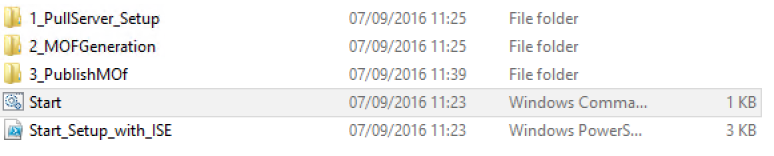
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Role | Region | Server Name | RAM | CPU | Disk Space Allocated |
| DEV | UK | gbw05552.hbeu.adroot.hsbc | 8 GB | 2 | 100 GB, Virtual Hard Disk – OS  D: 10GB – DSC Service [Configuration and Modules] |
| PROD | UK | gbw00146913.hbeu.adroot.hsbc | 8 GB | 2 | 100 GB, Virtual Hard Disk – OS  D: 10GB – DSC Service [Configuration and Modules] |
| PROD | UK | gbw00146914.hbeu.adroot.hsbc | 8 GB | 2 | 100 GB, Virtual Hard Disk – OS  D: 10GB – DSC Service [Configuration and Modules] |

#### Building DSC Pull Server Infrastructure

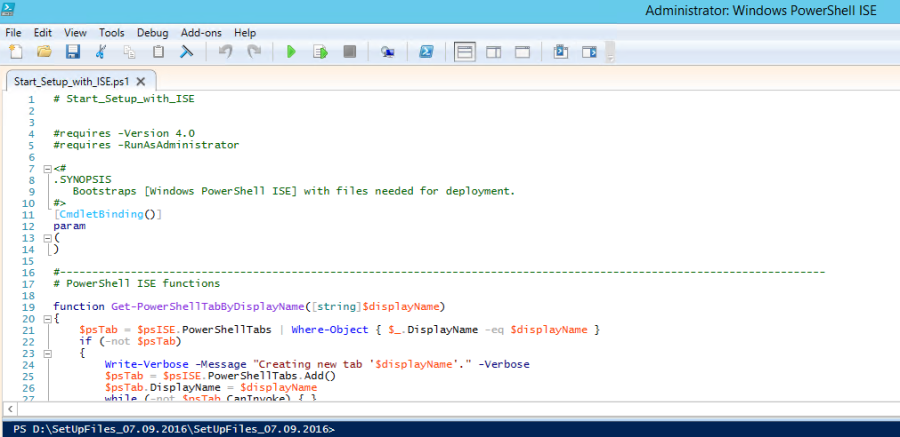
* Log on to the server that will become the Pull Server with an account that has local administrator privilege
* Extract the SetUpFiles\_07.09.2016.zip file to the local drive (for example, C:\Scripts).



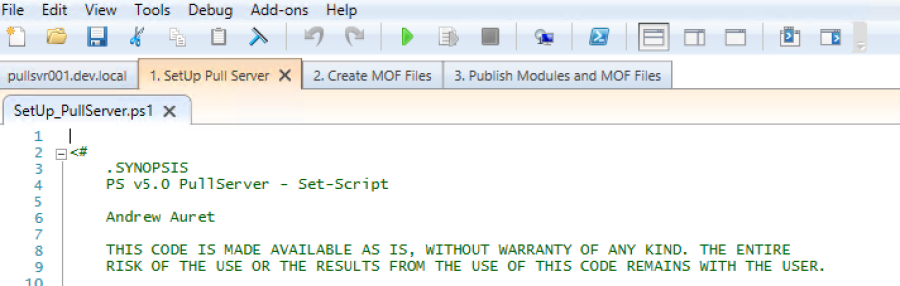
* Run the **Start.cmd** as an Administrator.



* Click on https://confluence.hk.hsbc/download/attachments/99120360/worddavdb0b9c654a1ac91836ced78cbf15783f.png?version=1&modificationDate=1473257894000&api=v2 to open up the necessary PowerShell Tabs w/script for the setup process.



* Select the Tab ***1. Setup Pull Server*** and click https://confluence.hk.hsbc/download/attachments/99120360/worddavdb0b9c654a1ac91836ced78cbf15783f.png?version=1&modificationDate=1473257894000&api=v2 to start the setup process

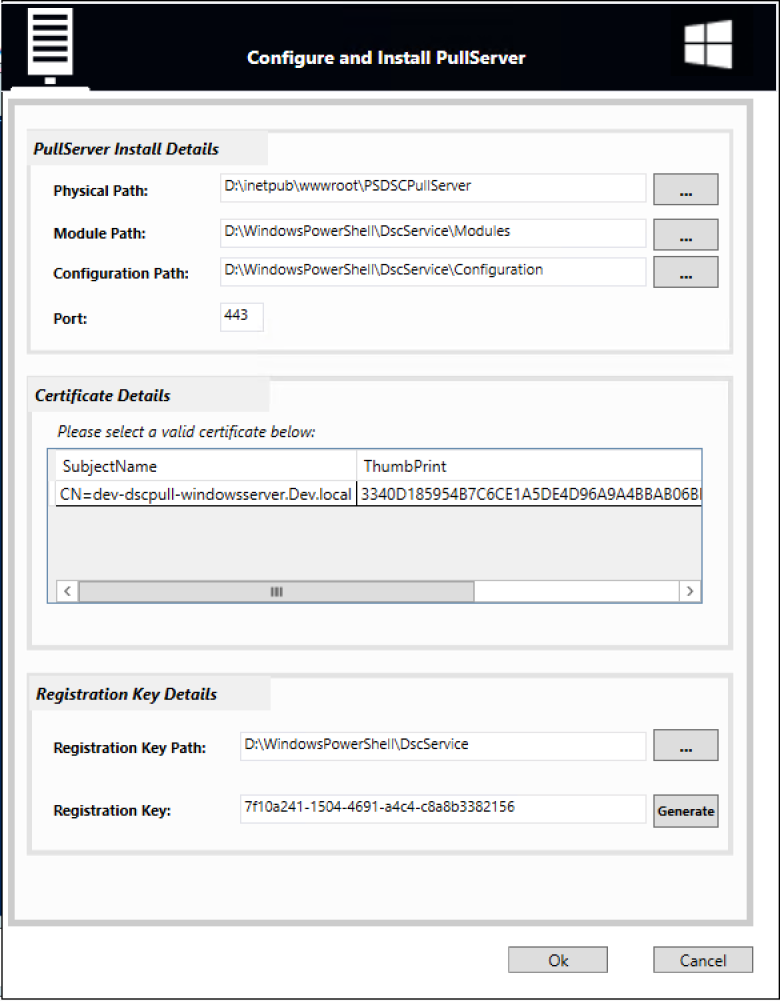


* Fill in the **PullServer Install Details** with the correct information.
* Select the correct **Certificate details** that will be used for SSL traffic.
* Fill in the **Registration Key Details** with the correct information

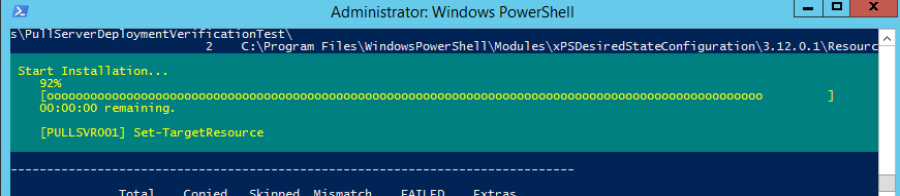
Note:

If this is the 1st Pull Server in the environment then generate a new Registration key.   
For subsequent Pull Servers, use the registration key that was generated as part of the install. This can be found in the Registration Key Path on the ***1st Pull Server*** under a text file named ***RegistrationKeys***. Open the file and Copy/Paste the Registration Key in the field.

Once all fields have been completed, click **OK**.

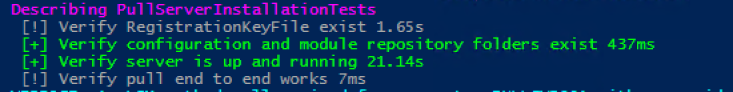


The PowerShell script will now Robocopy the required modules required for the Pull Server DSC module to execute correctly and then execute the **DSC\_PullServer.ps1** script with the correct parameters.



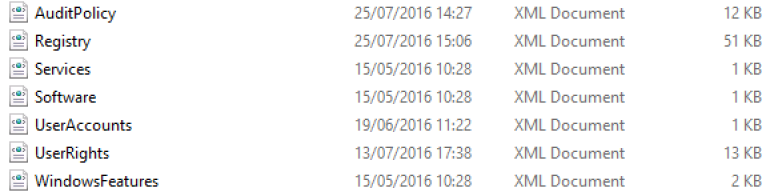
Once the **DSC\_PullServer.ps1** has completed, the ***Pester*** **PullServerSetupTests** are called to ensure that the Pull server has been setup as expected.   
  
**Pester performs the following tests:**

* Verifies that the configuration and modules folders exist.
* Ensures that the DSC Webserver is up and running and answering under the correct DNS name.   
  ***Any Errors will be reported in RED***.

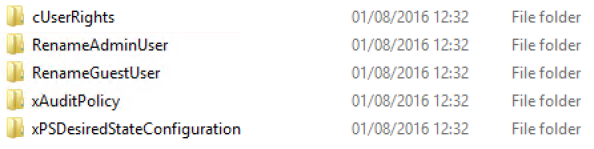


#### Creating DSC Pull Server MOF files

**Prerequisites:**  
*Ensure that all the XML documents have the correct values as per the approved Security Standards Settings.*



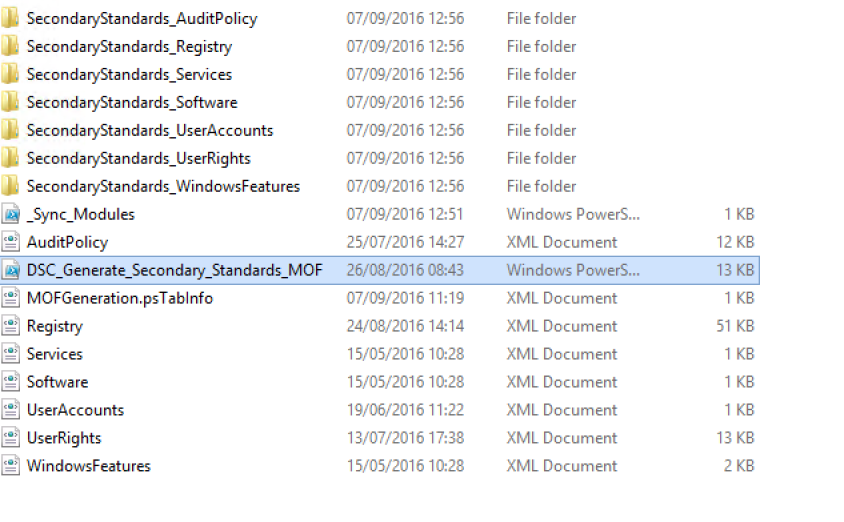
*Ensure all the correct custom DSC resource modules are located under the folder* ***SecondaryStandards.***



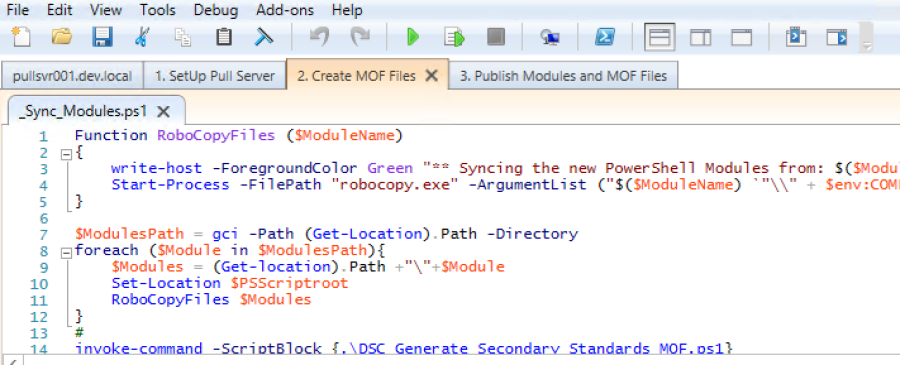
*Update the GUID for the named MOF's in* ***DSC\_Generate\_Secondary\_Standards\_MOF.ps1***

Note:

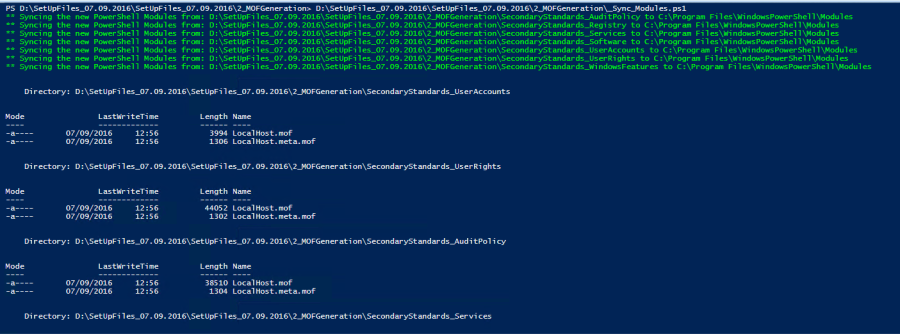
This would be the same GUID that was generated as part of the PULL Server setup and under a text file named ***RegistrationKeys*** under the DSC folder.



Select the Tab ***2. Create MOF Files*** and click https://confluence.hk.hsbc/download/attachments/99120360/worddavdb0b9c654a1ac91836ced78cbf15783f.png?version=1&modificationDate=1473257894000&api=v2 to start the sync and creation process.

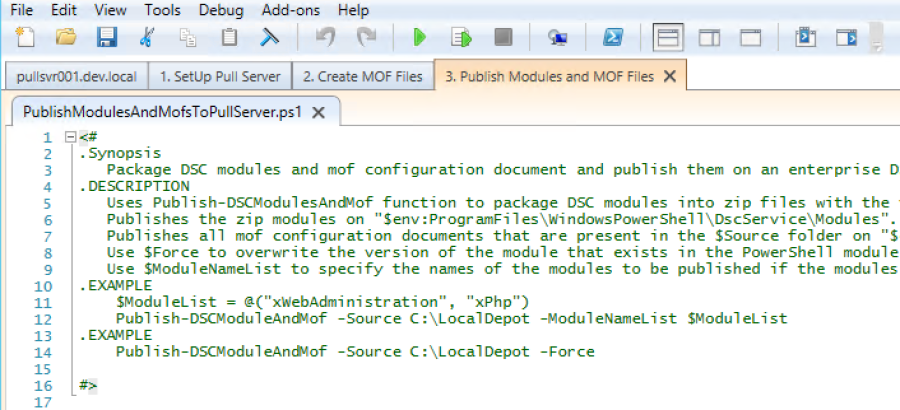


The PowerShell script will now Robocopy the modules required for the custom DSC functions to execute correctly and then executes the DSC\_Generate\_Secondary\_Standards\_MOF.ps1 script. As part of the process, each XML file is checked and executed upon with the end result being a Localhost.mof file being generated.



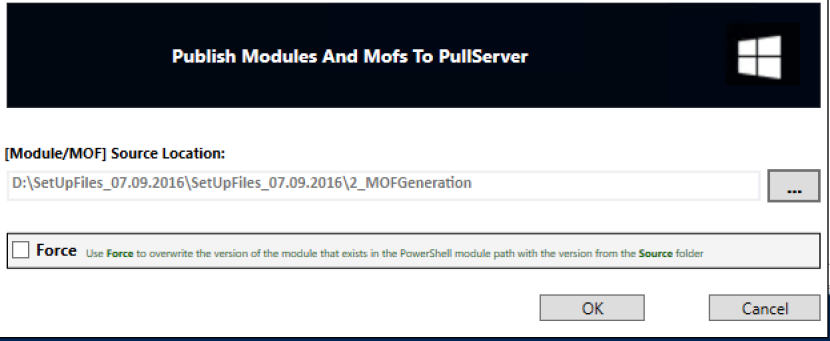
#### Publishing DSC Pull Server Resources

Select the Tab ***3.*** ***Publish Modules and MOF Files*** and click https://confluence.hk.hsbc/download/attachments/99120360/worddavdb0b9c654a1ac91836ced78cbf15783f.png?version=1&modificationDate=1473257894000&api=v2to start the sync and creation process.

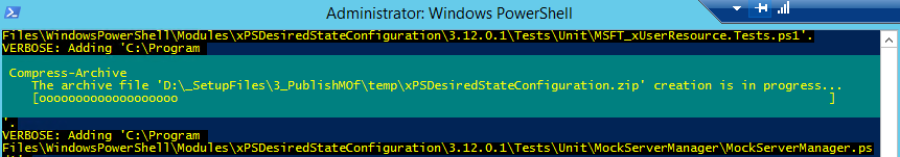


Enter the **Module/MOF Source Location** that was created in Creating DSC Pull Server MOF files.   
  
Ensure that the **Force** checkbox remains UNCHECKED.   
\*For Intial Setup and Sync – Select the Root of the 2\_MofGeneration Folder

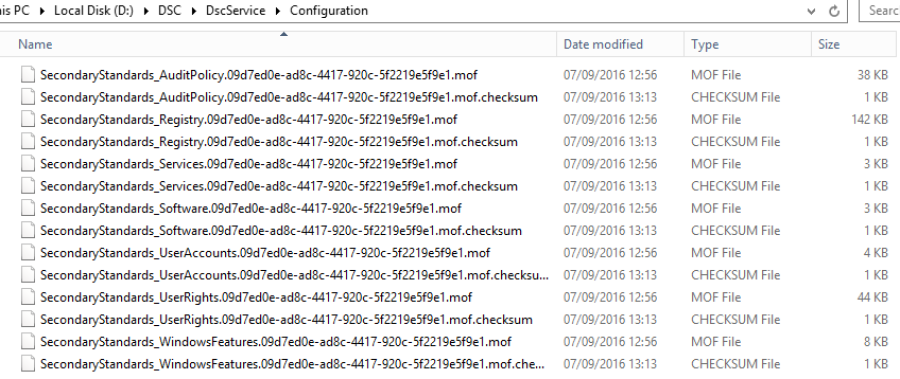
Click **OK** when complete.



The script will run and start the generation process.



All the **SecondaryStandards**.mof\* files and corresponding checksums will be created in the **Configuration path** as specified during the pull server creation.

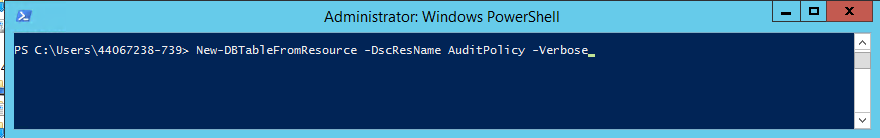


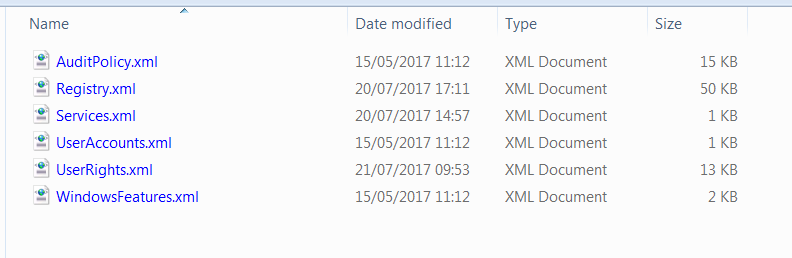
# Manage Configurations

## XML Import

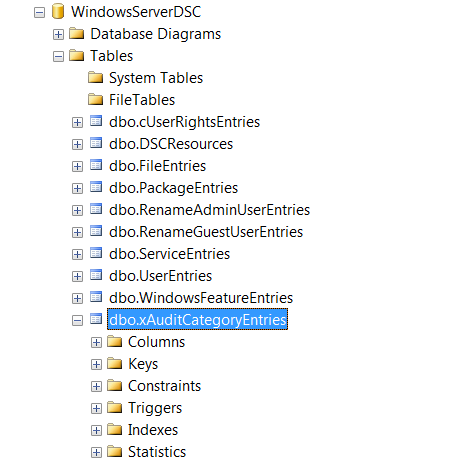
The Configurations are built from data held in the SQL database. This data is pulled in to SQL by using the [New-DBTableFromResource](#_New-DBTableFromResource) function.

A parameter provides the name of the resource which should match the name of the XML file.





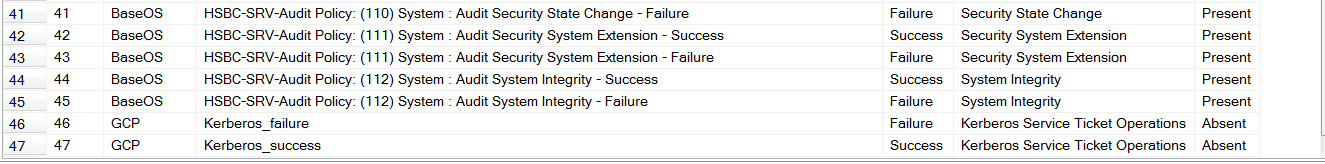
A database table will be created with all of the fields and data required to create a MOF



The table has the following columns:

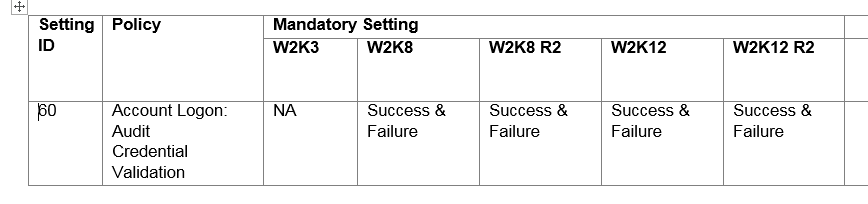


Multiple platforms can be catered for. We see GCP settings for Audit Policy in the example below



The 3rd column contains the values given in the *$Platform* parameter when generating a new MOF

Column 4 shows how the setting maps to the Secondary Security Standards. For example lines 1 and 2 satisfy Secondary Security Standard # 60



## Ongoing Maintenance

The configurations are maintained in one of 2 ways once they are is SQL. The XML file can be modified to include the changes if there are many for a particular table. This requires dropping the table and then importing again. This requires each Platform to be re-imported even if there are no changes made to some. For ad-hoc change, the SQL table values can be modified directly using SQL Server Management Studio. This is the simplest method and the preferred option in the majority of cases.

# Generate MOF

If configuration is required for a specific platform, the [New-DscMOF](#_New-DscMOF) function can be called to generate a new MOF file

## New-DscMOF Function

### Parameters

*$Platform*: Used to specify the desired platform the configuration is being targeted at. This will be passed to the [Update-ConfigBlock](#_Update-ConfigBlock) function and form part of the SQL query to return all records, from all tables, with a matching value in the CorePlatform column. (This is a **required** Parameter)

*$computerName:* Defaulting to localhost, this parameter will allow you to specify an alternate name that will be used for the MOF name. This is not used to make any connection and its value is not verified as being a valid hostname

*$ConfigName:* Currently not used

*$DebugConfig:* This switch parameter will enable you to dump the configuration script created in memory to the host for review and troubleshooting purposes

### Operation

When a call to [New-DscMOF](#_New-DscMOF) is made with a valid $Platform string specified, the following events take place:

1. A connection is opened to the SQL database using [Open-SqlConnection](#_Open-SqlConnection) function
2. All records from the DSCResources table are returned and stored in ***$dscresources***. This will provide a list of resource types that are stored in the database.
3. An array variable ***$MyConf*** is created to store the strings that will make up the DSC configuration script to be executed. Strings are appended to this array using += operator
4. DSC resource module names are extracted from ***$dscresources*** and included in the ***$MyConf*** array as part of the Import-DscResource string
5. For each of the DSC resources identified, a call will be made to [Update-ConfigBlock](#_Update-ConfigBlock_Function)
6. Following completion of the configuration script in ***$MyConf***, a call to the *New-ScriptBlock* method is made. This will create a new script block from the string array and assign it to variable ***$MyConfScript***
7. If no variables are found that are used to store matching table records from the specified platform, a message will be presented that no records were found and the script is written to the host
8. ***$MyConfScript*** will then be executed and any exceptions caught and handled
9. If successfully executed, a new MOF will be written to the subfolder .\MyStandards in the current working folder

## Update-ConfigBlock Function

### Parameters

*$dbTable:*  Database table name to query

*$ConfigBlock:* Configuration block read from the DSCResources table. The format of this string is as follows:

foreach ($row in $LogEntries) { Log $row.CoreDescription {Message = $row.Message;}}

This allows the script to dynamically build a MOF from records returned from individual tables

*$Platform:* Platform that the configuration will be used on

### Operation

When a call to Update-ConfigBlock is made from New-DscMOF, the following events take place:

1. A connection is opened to the SQL database using [Open-SqlConnection](#_Open-SqlConnection) function
2. All records from the database table specified in $dbtable that have a matching value for $Platform in the CorePlatform column are returned
3. If no records are returned, an output is made so this is recorded as a comment in the configuration script
4. For records returned, these are organised into sets of values populated and stored in a variable **$newDSC<TableName><BitmaskValue>**. This ensures no NULL strings are read during the configuration script execution.
5. Based on the variables created, the ConfigBlocks are now updated to remove any unpopulated fields. There will be one new ConfigBlock per variable to be included in the new configuration script.
6. The array of ConfigBlocks is now returned to New-DscMOF for inclusion in the configuration script.

# Deploy MOF

Once a MOF has been generated, it needs to be placed on the [Pull Server](#_Publishing_DSC_Pull). The MOF should be named in a way that identifies it as containing the particular settings within

***For example***

A new GCP MOF with no application-specific settings could be called **GCP-Base.MOF**

A new GCP MOF with SQL server settings could be called **GCP-SQL.MOF**

Similarly a new AWS MOF collection could be created, as could an Azure collection

The correct MOF for a client to use is determined by the Meta Configuration used to configure the Local Configuration Manager on the client

Should the settings change for a particular Configuration, only that MOF needs to be re-generated.

The MOF should be placed in the [Pull Server](#_Publishing_DSC_Pull) repository and given the same name as the one it replaces.

**For example**

GCP audit policy changes to include additional setting, or the existing setting changes.

Make the required changes in the SQL database

Generate new MOF

Move to Pull Server and name accordingly

Only the servers that are configured to use that MOF will collect it and apply the new settings

# Appendix

## Function Overview

### Open-SqlConnection

Usage: Opens a SQL connection.

function Open-SqlConnection

{

[CmdletBinding()]

param (

[string]$computername = " FQDN HOSTNAME,PORT",

[string]$database = " DATABASE NAME"

)

$connectionString = "Server=$computername;Database=$database;trusted\_connection=true;"

$connection = New-Object System.Data.SqlClient.SqlConnection

$connection.ConnectionString = $connectionString

Write-Verbose "Opening Connection $($connection.ConnectionString)"

$connection.Open()

return $connection

}

This is called from many functions but is used in different ways in different functions. It does contain defaults which will need to be changed to reflect environment specifics. (FQDN of SQL Server, SQL Port for the Database, Database name)

### Close-SqlConnection

Usage: Closes an open SQL connection.

|  |
| --- |
|  |
| function Close-SqlConnection  {  [CmdletBinding()]  param (  [object]$connection  )  Write-Verbose "Closing Connection $($connection.ConnectionString)"  $connection.Close()  } |
|  |  |

This is called when a connection to the database is no longer needed

### Get-DscDBTables

Usage: List the tables currently in the database.

function Get-DscDBTables

{

[CmdletBinding()]

param (

[object]$connection

)

$connection = Open-SqlConnection

$query = "SELECT Name FROM Sys.Tables WHERE Name LIKE '%Entries'"

$table = Initialize-Table -connection $connection -query $query

return $table

}

In its current form it will just query ‘sys.tables’ for a list of all tables in the DSC Database and output a list as below:

Name

----

ArchiveEntries

cUserRightsEntries

Entries

EnvironmentEntries

FileEntries

GroupEntries

GroupSetEntries

LogEntries

PackageEntries

ProcessSetEntries

RegistryEntries

ScriptEntries

### Get-DscSettings

Usage: Outputs all the configuration data stored for a particular resource.

function Get-DscSettings

{

[CmdletBinding()]

param (

[string]$Resource,

[switch]$ListDBTables

)

$connection = Open-SqlConnection

# Target table based on user input - DB standard for table names is '<DscRescoureName>Entries'

$requiredTable = $Resource + "Entries"

# Create an array of available tables

$dbTables = Get-DscDBTables -connection $connection

if($PSBoundParameters.ContainsKey("ListDBTables"))

{

Write-Output "`nThe following DSC Resources have entries in the database:"

$dbTables

return

}

# If a non-existent table has been specified then display what is available (drops the 'Entries' part of the table name)

if(!$dbtables.Name.ToUpper().Contains($requiredTable.ToUpper()))

{

Write-Host "`nUnknown resource, tables categories are available:`n"

$dbTables.Name.Replace('Entries','')

return

}

# A valid table has been requested, assign the query value and send

$query = "SELECT \* FROM $requiredTable"

$table = Initialize-Table -connection $connection -query $query

# Troubleshooting

Write-Verbose "Row Count from table in Get-DscSettings is $($table.Rows.Count)"

Close-SQLConnection $connection

return $table

}

Objects returned are of type ‘System.Data.DataRow’ allowing for pipeline operations.

Get-DscSettings -Resource Log | Where-Object {$PSItem.Message -eq 'Hello'}

CoreID CorePlatform CoreDescription Message

------ ------------ --------------- -------

4 BaseOS Team Message Hello

### Initialize-Table

Usage: Initialize-Table is a help function that sends the SELECT query to the SQL server and populates a DataTable

function Initialize-Table

{

[CmdletBinding()]

param (

[object]$connection,

[string]$query

)

# Verbose output for test

Write-Verbose "Populating table from query '$query'"

# Create command from passed in connection, assign the query and execute.

$command = $connection.CreateCommand()

$command.CommandText = $query

$results = $command.ExecuteReader()

# Save the results to a table to return to the caller

$table = new-object “System.Data.DataTable”

$table.Load($results)

# Troubleshooting - this will be removed at some point but was intended to verify data being returned.

Write-Verbose "Row Count from table in Initialize-Table is $($table.Rows.Count)"

return $table

}

It returns a ‘System.Data.DataTable’ object which allows manipulation of the results via the standard PowerShell pipeline.

### New-DBTableForDSCMetadata

Usage: This builds the table for the storage of DSC Resource metadata

function New-DBTableForDSCMetadata

{

[CmdletBinding()]

param (

[object]$connection

)

$command = $connection.CreateCommand()

try

{

$command.commandtext = "CREATE TABLE [dbo].[DSCResources](

[ResourceName] [varchar](max) NOT NULL,

[ResourceModule] [varchar](max) NOT NULL,

[ResourceModuleVersion] [varchar](max) NOT NULL,

[ResourceType] [varchar](max) NOT NULL,

[ConfigBlock] [varchar](max) NOT NULL

) ON [PRIMARY] TEXTIMAGE\_ON [PRIMARY]"

$command.ExecuteNonQuery()

}

catch

{

# The database already exists, at this point we will bail. I may add another switch parameter to update in case

# a resource is updated and another property is added.

Write-Host "$($\_.Exception.Message)" -ForegroundColor White -BackgroundColor Red

return

}

}

This contains Resource Type, Module Version etc. and a ConfigBlock string which acts as a template for reading in configuration settings.

We currently store the properties below:

$TableEntries | gm -MemberType Properties

TypeName: System.Data.DataRow

Name MemberType Definition

---- ---------- ----------

ConfigBlock Property string ConfigBlock {get;set;}

ResourceModule Property string ResourceModule {get;set;}

ResourceModuleVersion Property string ResourceModuleVersion {get;set;}

ResourceName Property string ResourceName {get;set;}

ResourceType Property string ResourceType {get;set;}

We reference the values in this table for the creation of the configuration script ‘New-DscMOF’ builds.

### New-DBTableFromResource

Usage: Extracts the properties from a DSC resource and creates a new table based on these for storing configuration.

function New-DBTableFromResource

{

[CmdletBinding()]

param (

[string]$DscResName,

[object]$connection

)

[string]$PropBlock = ""

if(($DscResObj = Get-DscResource $DscResName).Count -gt 1)

{

Write-Host "There appears to be more than one version of this resource present." -BackgroundColor Red

$DscResObj

$version = Read-Host "Please enter the version you wish to use: "

$DscResObj = $DscResObj | Where-Object {$\_.Version -like $version}

}

$props = $DscResObj | Select-Object -ExpandProperty Properties

# Open a connection to SQL and create a 'System.Data.SqlClient.SqlCommand' object

if(!$connection)

{

$connection = Open-SqlConnection

}

$command = $connection.CreateCommand()

# Define a name for the new table based on resource if all succeeded above

$tablename = $DscResName + "Entries"

# Create the table with default columns of ID (primary key), partialset this will apply to and a identifier for MOF generation. This names are prefixed to avoid conflicts.

try

{

$command.commandtext = "CREATE TABLE [dbo].[$tablename](

[CoreID] [int] IDENTITY(1,1) NOT NULL,

[CorePlatform] [varchar](255) NOT NULL,

[CoreDescription] [varchar](255) NOT NULL,

PRIMARY KEY CLUSTERED

(

[CoreID] ASC

)WITH (PAD\_INDEX = OFF, ‘

STATISTICS\_NORECOMPUTE = OFF, ‘

IGNORE\_DUP\_KEY = OFF, ‘

ALLOW\_ROW\_LOCKS = ON, ‘

ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]"

$command.ExecuteNonQuery()

}

catch

{

# The database already exists, at this point we will bail.

BackgroundColor Red

return

}

# Now add the columns, loop through the properties returned

foreach($prop in $props)

{

# We have to do a bit of string manipulation here as the '[' causes unexpected behaviour in

# string comparisons. We strip them off and wild card to handle arrays.

# SQL contains some keywords which cannot be used for columns. Update the array below in case these are encountered

$SqlKeyWords = @('Key','Table','Index','Database')

# Check if $prop.Name is considered a keyword and change.

if($SqlKeyWords.Contains($prop.Name)){$prop.Name = $prop.Name+"Name"}

# Determine type and add to table definition

switch -wildcard ($($prop.PropertyType).TrimStart('[').TrimEnd(']'))

{

"string\*"

{

if($prop.Name -notlike "DependsOn")

{

$command.commandtext = "ALTER TABLE $tablename ‘

ADD $($prop.Name) [varchar](max)"

$command.ExecuteNonQuery()

if ($prop.name -eq "Members" -or $prop.name -eq "ValueData")

{

$PropBlock += $prop.Name + ' = $($row.' + $prop.Name + ' -split ",")'+ ';'

}

else

{

$PropBlock += $prop.Name + ' = ' + '$row.' + $prop.Name + ';'

}

}

}

"bool"

{

$command.commandtext = "ALTER TABLE $tablename ‘

ADD $($prop.Name) [bit]"

$command.ExecuteNonQuery()

$PropBlock += $prop.Name + ' = ' + '$row.' + $prop.Name + ';'

}

"UInt32\*"

{

$command.commandtext = "ALTER TABLE $tablename ‘

ADD $($prop.Name) [int]"

$command.ExecuteNonQuery()

$PropBlock += $prop.Name + ' = ' + '$row.' + $prop.Name + ';'

}

}

}

# Start the ConfigBlock which will be saved to the DSCResource Table in order to build MOFs

$ConfigBlock = 'foreach($row in $' + $tablename + ') { ' + $DscResName + ' $row.CoreDescription {' + $PropBlock + '}}'

# Fixup the Config Block so any reserved SQL keyword is stored correctly as it's DSC resource property name.

foreach($word in $SqlKeyWords)

{

$ConfigBlock = $ConfigBlock.Replace($word + "Name =",$word + ' =')

}

# Update DSCResource metadata table

$command.commandtext = "INSERT INTO DSCResources (ResourceName,ResourceModule,ResourceModuleVersion,ResourceType,ConfigBlock) ‘

VALUES('{0}','{1}','{2}','{3}','{4}')" -f

$DscResName,$DscResObj.ModuleName,$DscResObj.Version.ToString(),$DscResObj.ResourceType,$ConfigBlock

# Send Command

$command.ExecuteNonQuery()

# Clean up connection

Close-SQLConnection -connection $connection

}

The column types created will be based on the property types extracts. We currently support:

**DSC Property Type | SQL Property Type**

------------------ ----------|------------------------------

bool | bit

string | varchar(max)

UInt32 | int

If required the ‘switch’ statement can be extended to support new types.

This function does accept pipeline input so an array of DSC Resources names can be passed in to create multiple tables in one batch.

Following the creation of a new table, the DSCResources table is populated with the correct module details and a configuration block for use in ‘New-DscMOF’.

### New-DscMOF

Usage: Outputs a new MOF based on a platform selected.

function New-DscMOF

{

[CmdletBinding()]

param (

[string]$Platform,

[string]$ComputerName = "localhost",

[string]$ConfigName = "MySettings"

)

$connection = Open-SqlConnection

# DSC Resource metadata is stored in the DSCResources table. Read all of the settings from here so we can look to build configuration blocks

$query = "SELECT \* FROM DSCResources"

$dscresources = Initialize-Table -connection $connection -query $query

Close-SQLConnection -connection $connection

# We will build an array of strings that will make up the script. This will then be executed once complete to produce the .MOF

$MyConf = @()

$MyConf += "Configuration MySettings{"

$MyConf += 'Import-DscResource -ModuleName PSDesiredStateConfiguration'

# Add modules used, this will not be reflected in final MOF if no settings for a particular resource are required.

foreach($row in $dscresources)

{

if($row.ResourceModule -notlike "PSDesiredStateConfiguration")

{

$MyConf += "Import-DscResource -ModuleName $($row.ResourceModule)" + " -ModuleVersion $($row.ResourceModuleVersion)"

}

}

# Write the compunter name, defaults to localhost.

$MyConf += "node $ComputerName{"

# Add the ConfigBlock for each resource, this needs to review what columns are in use and build some globals for reference.

foreach($row in $dscresources)

{

$dbTableName = "$($row.ResourceName)Entries"

$MyConf += Update-ConfigBlock -dbTable $dbTableName -ConfigBlock $row.ConfigBlock -Platform $Platform

}

# Close the statements and add the call, there may be a need to add parameters here.

$MyConf += '}}'

$MyConf += "MySettings"

# Build the Config from the string array and add some line breaks.

$MyConfScript = ""

$MyConf | ForEach-Object {$MyConfScript += $\_.ToString() + "‘n"}

$MyConfScript = $ExecutionContext.InvokeCommand.NewScriptBlock($MyConfScript)

try

{

& $MyConfScript -Verbose

}

catch [System.UnauthorizedAccessException]

{

Write-Host "$($\_.Exception.Message)" -ForegroundColor White -BackgroundColor Red

Write-Host "Please ensure you are running with Administrator privileges" -ForegroundColor White -BackgroundColor Red

Remove-Artefacts

return

}

catch

{

Write-Host "$($\_.Exception.Message)" -ForegroundColor White -BackgroundColor Red

Remove-Artefacts

return

}

# Clean up variables to avoid issues on subsequent runs

Remove-Artefacts

}

By referencing the values stored in the DSCResources table a configuration script is built dynamically and then executed. There is a ‘-DebugConfig’ switch available if you would like to view the script as this can be useful for troubleshooting.

The function makes a call to ‘[Update-ConfigBlock](#_Update-ConfigBlock)’ to ensure we don't hit issues with empty properties being read by the configuration script.

If no records are found for a specified platform we will dump the in-memory script for review.

NOTE:

The ‘-Platform’ parameter value is wildcarded before being sent to SQL. This should be taken into account as creating similar platform names could lead to unexpected results. For example, if using ‘BaseOS’ and ‘BaseOSv2’, searching for ‘BaseOS’ will return both. This can be avoided by Full-Text indexing the tables and amending the query to perform a CONTAINS().

### Open-DSCSettings

Usage: This presents a Windows Form for editing and viewing table data.

function Open-DSCSettings

{

[CmdletBinding()]

param (

[object]$connection,

[parameter(mandatory)]

[string]$Resource

)

$connection = Open-SqlConnection

# Target table based on user input - DB standard for table names is '<DscRescoureName>Entries'

$requiredTable = $Resource + "Entries"

# Create an array of available tables

$dbTables = Get-DscDBTables -connection $connection

# If a non-existent table has been specified then display what is available (drops the 'Entries' part of the table name)

if(!$dbtables.Name.ToUpper().Contains($requiredTable.ToUpper()))

{

Write-Host "‘nUnknown resource, tables categories are available:’n"

$dbTables.Name.Replace('Entries','')

return

}

# DSC Settings - Uses a PowerShell Form Object

Add-type -AssemblyName System.Windows.Forms

[System.Windows.Forms.Application]::EnableVisualStyles()

$form1 = New-Object 'System.Windows.Forms.Form'

$datagridview1 = New-Object 'System.Windows.Forms.DataGridView'

$buttonOK = New-Object 'System.Windows.Forms.Button'

$InitialFormWindowState = New-Object 'System.Windows.Forms.FormWindowState'

$DGVhasChanged = $false

# Load the form and populate with selected table data

$form1\_Load = {

$connection = Open-SqlConnection

$cmd = $connection.CreateCommand()

$cmd.CommandText = "SELECT \* FROM $Resource" + "Entries"

$script:adapter = New-Object System.Data.SqlClient.SqlDataAdapter($cmd)

$dt = New-Object System.Data.DataTable

$script:adapter.Fill($dt)

$datagridview1.DataSource = $dt

$cmdBldr = New-Object System.Data.SqlClient.SqlCommandBuilder($adapter)

}

$buttonOK\_Click = {

if ($script:DGVhasChanged -and [System.Windows.Forms.MessageBox]::Show('Do you wish to save?', 'Data Changed', 'YesNo')) {

$script:adapter.Update($datagridview1.DataSource)

}

}

$datagridview1\_CurrentCellDirtyStateChanged = {

$script:DGVhasChanged = $true

}

$Form\_StateCorrection\_Load ={

$form1.WindowState = $InitialFormWindowState

}

$form1.SuspendLayout()

## Create the button panel to hold the OK and Cancel buttons

$buttonPanel = New-Object Windows.Forms.Panel

$buttonPanel.Size = New-Object Drawing.Size @(400,40)

$buttonPanel.Dock = "Bottom"

## Create the Cancel button, which will anchor to the bottom right

$cancelButton = New-Object Windows.Forms.Button

$cancelButton.Text = "Cancel"

$cancelButton.DialogResult = "Cancel"

$cancelButton.Top = $buttonPanel.Height - $cancelButton.Height - 10

$cancelButton.Left = $buttonPanel.Width - $cancelButton.Width - 10

$cancelButton.Anchor = "Right"

## Create the OK button, which will anchor to the left of Cancel

$okButton = New-Object Windows.Forms.Button

$okButton.Text = "Ok"

$okButton.DialogResult = "Ok"

$okButton.Top = $cancelButton.Top

$okButton.Left = $cancelButton.Left - $okButton.Width - 5

$okButton.Anchor = "Right"

$okButton.add\_Click($buttonOK\_Click)

## Add the buttons to the button panel

$buttonPanel.Controls.Add($okButton)

$buttonPanel.Controls.Add($cancelButton)

# form1

$form1.Controls.Add($datagridview1)

$form1.Controls.Add($buttonPanel)

$form1.AcceptButton = $okButton

$form1.CancelButton = $cancelButton

$form1.ClientSize = '646, 374'

$form1.FormBorderStyle = 'Sizable'

$form1.MaximizeBox = $False

$form1.MinimizeBox = $False

$form1.Name = 'form1'

$form1.StartPosition = 'CenterScreen'

$form1.Text = $ResourceType + "Entries Table Data"

$form1.add\_Load($form1\_Load)

$datagridview1.Anchor = 'Top, Bottom, Left, Right'

$datagridview1.ColumnHeadersHeightSizeMode = 'AutoSize'

$datagridview1.Location = '13, 13'

$datagridview1.Name = 'datagridview1'

$datagridview1.Size = '621, 309'

$datagridview1.TabIndex = 1

$datagridview1.add\_CurrentCellDirtyStateChanged($datagridview1\_CurrentCellDirtyStateChanged)

$buttonOK.Anchor = 'Bottom, Right'

$buttonOK.DialogResult = 'OK'

$buttonOK.Location = '559, 339'

$buttonOK.Name = 'buttonOK'

$buttonOK.Size = '75, 23'

$buttonOK.TabIndex = 0

$buttonOK.Text = '&OK'

$buttonOK.UseVisualStyleBackColor = $True

$buttonOK.add\_Click($buttonOK\_Click)

$form1.ResumeLayout()

$InitialFormWindowState = $form1.WindowState

$form1.add\_Load($Form\_StateCorrection\_Load)

# Display for the Form

$form1.ShowDialog()

}

Any changes made generate a prompt asking the operator if they wish to save or discard them

### Update-ConfigBlock

Usage: Removes empty / null values

function Update-ConfigBlock

{

[CmdletBinding()]

param (

[string]$dbTable,

[string]$ConfigBlock,

[string]$Platform

)

# Open a connection to the DB to return records and run the query.

$connection = Open-SqlConnection

$query = "SELECT \* FROM $dbTable WHERE CorePlatform LIKE '%$Platform%'"

$TableEntries = @(Initialize-Table -connection $connection -query $query)

if($TableEntries.Count -eq 0)

{

Write-Output "#No records found for $dbTable"

return

}

# We have data so extract the Table Columns so we can count the number and reference the names

[System.Data.DataTable] $table = $TableEntries[0].Table

# prefix name for the variables used in the script.

$tablePrefix = "newDSC$dbTable"

# Create a hash table with column names to bits, this will be checked with the bitmask to determine those column names in use

# Raising 2 to the power of i$ (starting at 0) will provide the Keys as bit values.

$columnNames = @{}

for($i = 0;$i -lt $table.Columns.Count;$i++)

{

$columnNames.Add([Math]::Pow(2, $i),$table.Columns[$i].ColumnName)

}

# Check each column in the DataRow and build a bitmask representing those in use.

foreach($row in $TableEntries)

{

$bitMaskValue = 0

for($i = 0;$i -lt $table.Columns.Count;$i++)

{

if(!$row.IsNull($i))

{

# Need to build bitmask of populated columns

$bitMaskValue = $bitMaskValue + [Math]::Pow(2, $i)

}

}

# Crearte a variable for each column 'in use' variation. Variables may already be present from previous run, if so

# clear, if not create.

# TODO clean up variables

if(!(Test-Path Variable:\$($tablePrefix + $bitMaskValue)))

{

New-Variable -Name ($tablePrefix + $bitMaskValue) -Value @() -Scope Global

}

# Add the row to the correct array based on columns in use.

(Get-Variable -Name ($tablePrefix + $bitMaskValue)).Value += $row

}

# We should now have all the values placed into new tables based on the columns used. Output a new $configblock to an array

# that will then be written to the in memory configuration script

$configBlockArray = @()

foreach($newVariable in (Get-Variable -Name "newDSC$dbTable\*"))

{

# We need to assign the $ConfigBlock passed in to a new varaible to avoid clashes.

$newBlock = $ConfigBlock

$newBlock = $newBlock.Replace($dbTable,"$($newVariable.Name)")

# Extract the bitmask from the end of the table name so we can use the value to evalate what needs to be removed from

# the configBlock string. The column names not in use will be added to

$bitmask = $newVariable.Name.Replace("$tablePrefix","")

$columnsToRemove = $columnNames.Keys | Where-Object {!($\_ -band $bitmask)} | Foreach-Object {$columnNames.Get\_Item($\_)}

# Loop around the values to remove. This is quite straightforward as the text pattern is fixed when we wrote to DB.

foreach($column in $columnsToRemove)

{

# We will need a special case here where a known keyword needs to be removed.

# This string pattern is based on what is written into the DSCResources table

$newBlock = $newBlock.Replace("$column = " + '$row' + ".$column;","")

}

# Add the new block to the array and loop back around if needed.

$configBlockArray += $newBlock

}

# Return the strings for inclusion in the configuration script.

return $configBlockArray

}

This will take a base configuration block and remove the redundant entries based on the columns used in the DataRow record. If we do not remove these, errors will be thrown as any empty string will be used for the value which is not supported by most properties.

For Example, for the record below:

Name : All

Ensure :

IncludeAllSubFeature : True

LogPath : C:\Logs

Source :

We will change this ConfigBlock:

{Name = $row.Name;Ensure = $row.Ensure;IncludeAllSubFeature = $row.IncludeAllSubFeature;LogPath = $row.LogPath;Source = $row.Source;}}

To this:

{Name = $row.Name;IncludeAllSubFeature = $row.IncludeAllSubFeature;LogPath = $row.LogPath;}}

This ensures the unused columns of ‘Ensure’and ‘Source’ are not referenced during configuration script compile.

In order to identify those properties to be removed we create a hash table of all properties and an associated bit value.

Name Value

---- -----

1 CoreID

2 CorePlatform

4 CoreDescription

8 Name

16 Ensure

32 IncludeAllSubFeature

64 LogPath

128 Source

Once we've created this hashtable we can then the loop through the records and build a bitmask that will map empty properties. These records will then be added to a new table.

if(!$row.IsNull($i))

{

# Need to build bitmask of populated columns

$bitMaskValue = $bitMaskValue + [Math]::Pow(2, $i)

}

The result will be multiple arrays of Datarows that all share a common set of populated properties.

### Update-ConfigBlockData

Usage: Cleans up the global variables used to store rows.

function Update-ConfigBlockData

{

param(

[string]$ConfigBlock,

[string]$Resource

)

$connection = Open-SqlConnection

$command = $connection.CreateCommand()

$command.commandtext = "UPDATE [dbo].[DSCResources]

SET [ConfigBlock] = '{0}'

WHERE [ResourceName] = '{1}';" -f $ConfigBlock,$Resource

$command.ExecuteNonQuery()

return

}

Called from New-DscMOF after success or fail.

### Remove-Artefacts

Usage: Clean up variables to avoid issues on subsequent runs

function Remove-Artefacts

{

Get-Variable "newDSC\*" | Remove-Variable -Scope Global

}

Called from New-DscMOF

### Add-DscChangeRecord

Usage: Optional Function to allow changes to the DSC Database to be tracked and logged

function Add-DscChangeRecord

{

[CmdletBinding()]

param (

[object]$connection,

[string]$user,

[string]$operation,

[string]$description,

[string]$date

)

$command = $connection.CreateCommand()

# T-SQL INSERT INTO string

$command.commandtext = "INSERT INTO ChangeLog (Name,Operation,Description,Date)

VALUES('{0}','{1}','{2}','{3}')" -f

$user,$operation,$description,$date

# Send Command

$command.ExecuteNonQuery()

}

Not enabled at this point in time

### Add-DscMultiValueColumn

Usage: Optional Function to allow the default delimitator (comma) to be substituted for an alternative character, such as semi-colon

function Add-DscMultiValueColumn

{

param(

[string]$Resource,

[string]$Property

)

$connection = Open-SqlConnection

$query = "SELECT \* FROM DSCResources WHERE ResourceName LIKE '$Resource'"

$dscresource = Initialize-Table -connection $connection -query $query

# Bit of a catch all, if the resource isn't in the DB or the property is invlaid it will fail.

if(!($dscresource.ConfigBlock -like "\*$Property;\*"))

{

Write-Output "Cannot locate $Property in ConfigBlock for Resource $Resource"

Write-Output "This is what was found:`n"

$dscresource.ConfigBlock

return

}

else

{

$newConfigBlock = $dscresource.ConfigBlock.Replace("$Property;","$Property.Split(`",`");")

Update-ConfigBlockData -ConfigBlock $newConfigBlock -Resource $dscresource.ResourceName

}

}

## Sample XML

<UserAccounts>

<RenameAdministrator>

<Description>Administrator</Description>

<Name>40968192</Name>

<State>Present</State>

</RenameAdministrator>

<RenameGuest>

<Description>Guest</Description>

<Name>05121024</Name>

<State>Present</State>

</RenameGuest>

<DisableGuest>

<Description>Guest Account - Disable</Description>

<Name>05121024</Name>

<Disabled>True</Disabled>

<DependsOn>[RenameGuestUser]Guest</DependsOn>

<State>Present</State>

</DisableGuest>

</UserAccounts>

## Sample DSC Configuration Script created in memory

Configuration MySettings {

Import-DscResource -ModuleName PSDesiredStateConfiguration -ModuleVersion 1.1

Import-DscResource -ModuleName cUserRights -ModuleVersion 1.0.0.1

Import-DscResource -ModuleName xPSDesiredStateConfiguration -ModuleVersion 3.12.0.1

node localhost {

# No records found for RegistryEntries with Platform of BaseOS

foreach($row in $newDSCLogEntries15) { Log $row.CoreDescription {Message = $row.Message;}}

# No records found for cUserRightsEntries with Platform of BaseOS

foreach($row in $newDSCWindowsFeatureEntries15) { WindowsFeature $row.CoreDescription {Name = $row.Name;}}

# No records found for xRegistryEntries with Platform of BaseOS

}}

MySettings

## Example of MOF content

/\*

@TargetNode='localhost'

@GeneratedBy=04811712

@GenerationDate=07/20/2017 21:23:39

@GenerationHost=GBW02855

\*/

instance of MSFT\_xRegistryResource as $MSFT\_xRegistryResource1ref

{

ResourceID = "[xRegistry]Disable IPv6";

ValueName = "DisabledComponents";

Ensure = "Present";

Key = "HKEY\_LOCAL\_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\Tcpip6\\Parameters";

AcceptedValues = "";

Mode = "None";

SourceInfo = "::9::48::xRegistry";

ValueType = "Dword";

ModuleName = "xPSDesiredStateConfiguration";

ValueData = {

"255"

};

ModuleVersion = "3.12.0.1";

ConfigurationName = "MySettings";

};

instance of MSFT\_xRegistryResource as $MSFT\_xRegistryResource2ref

{

ResourceID = "[xRegistry]HSBC-SRV-Event Log: (115) Maximum Application Log Size";

ValueName = "MaxSize";

Ensure = "Present";

Key = "HKEY\_LOCAL\_MACHINE\\SOFTWARE\\Policies\\Microsoft\\Windows\\Eventlog\\Application";

AcceptedValues = "";

Mode = "None";

SourceInfo = "::9::48::xRegistry";

ValueType = "Dword";

ModuleName = "xPSDesiredStateConfiguration";

ValueData = {

"1024000"

};

ModuleVersion = "3.12.0.1";

ConfigurationName = "MySettings";

};

instance of MSFT\_xRegistryResource as $MSFT\_xRegistryResource3ref

{

ResourceID = "[xRegistry]HSBC-SRV-Event Log: (116) Maximum Security Log Size";

ValueName = "MaxSize";

Ensure = "Present";

Key = "HKEY\_LOCAL\_MACHINE\\Software\\Policies\\Microsoft\\Windows\\Eventlog\\Security";

AcceptedValues = "";

Mode = "None";

SourceInfo = "::9::48::xRegistry";

ValueType = "Dword";

ModuleName = "xPSDesiredStateConfiguration";

ValueData = {

"1024000"

};

ModuleVersion = "3.12.0.1";