

Windows PowerShell Blog

Automating the world one-liner at a time...

PowerShell DSC Resource Design and Testing Checklist

 PowerShell Team

 18 Nov 2014 12:17 PM

 1

Writing Desired State Configuration Resources is not trivial and there’s many things which can go wrong. You should always spend significant amount of time testing resources you create. This task, although critical for quality, is often overlooked or performed carelessly and insufficiently. Below you can find **DSC Resource Design and Testing Checklist** which will help you design the resource and identify critical areas for testing. Follow it the next time you want to verify quality of your resource and check whether it adheres to DSC best practices.

- [1 Resource module contains .psd1 file and schema.mof for every resource](#)
- [2 Resource and schema are correct and have been verified using DscResourceDesigner cmdlets](#)
- [3 Resource loads without errors](#)
- [4 Resource is idempotent in the positive case](#)
- [5 User modification scenario was tested](#)
- [6 Get-TargetResource functionality was verified using Get-DscConfiguration](#)
- [7 Resource was verified by calling Get/Set/Test-TargetResource functions directly](#)
- [8 Resource was verified End to End using Start-DscConfiguration](#)
- [9 Resource behaves correctly on all DSC supported platforms \(or returns a specific error otherwise\)](#)
- [10 Resource functionality was verified on Windows Client \(if applicable\)](#)
- [11 Get-DSCResource lists the resource](#)
- [12 Resource module contains examples](#)
- [13 Error messages are easy to understand and help users solve problems](#)
- [14 Log messages are easy to understand and informative \(including -verbose, -debug and ETW logs\)](#)
- [15 Resource was tested with valid/invalid credentials](#)
- [16 Resource is not using cmdlets requiring interactive input](#)
- [17 Resource functionality was thoroughly tested](#)
- [18 Best practice: Resource module contains Tests folder with ResourceDesignerTests.ps1 script](#)
- [19 Best practice: Resource folder contains resource designer script for generating schema](#)
- [20 Best practice: Resource supports -whatif](#)

1 Resource module contains .psd1 file and schema.mof for every resource

The first think you should do is to check that your resource has correct structure and contains all required files. Every resource module should contain a .psd1 file and every non-composite resource should have schema.mof file. Resources that do not contain schema will not be listed by Get-DscResource and users will not be able to use the intellisense when writing code against those modules in ISE. The sample directory structure for xRemoteFile resource, which is part of the xPSDesiredStateConfiguration resource module, could look as follows:

```
xPSDesiredStateConfiguration
├── DSCResources
│   └── MSFT_xRemoteFile
│       └── MSFT_xRemoteFile.psm1
```

	MSFT_xRemoteFile.schema.mof
Examples	xRemoteFile_DownloadFile.ps1
ResourceDesignerScripts	GenerateXRemoteFileSchema.ps1
Tests	ResourceDesignerTests.ps1
	xPSDesiredStateConfiguration.psd1

If you don't quite understand what every single item on this list is for, keep reading - I'll explain it in the sections below.

2 Resource and schema are correct and have been verified using DscResourceDesigner cmdlets

Another important aspect is verifying the resource schema (*.schema.mof) file.
Make sure that:

Property types are correct (e.g. don't use String for properties which accept numeric values, you should use UInt32 or other numeric types instead)

Property attributes are specified correctly ([key], [required], [write], [read])

- At least one parameter in the schema has to be marked as [key]
- [read] property cannot coexist together with any of: [required], [key], [write]
- If multiple qualifiers are specified except [read], then [key] takes precedence
- If [write] and [required] are specified, then [required] takes precedence

ValueMap is specified where appropriate

Example:

```
[Read, ValueMap{"Present", "Absent"}, Values{"Present", "Absent"}, Description("Says whether DestinationPath exists on the machine")] String Ensure;
```

Friendly name is specified and confirms to DSC naming conventions

Example:

```
[ClassVersion("1.0.0.0"), FriendlyName("xRemoteFile")]
```

Every field has meaningful description

Below you can find a good example of the resource schema file (this is the actual schema of xRemoteFile resource from the [DSC Resource Kit](#))

```
[ClassVersion("1.0.0.0"), FriendlyName("xRemoteFile")]
class MSFT_xRemoteFile : OMI_BaseResource
{
    [Key, Description("Path under which downloaded or copied file should be accessible after operation.")] String DestinationPath;
    [Required, Description("Uri of a file which should be copied or downloaded. This parameter supports HTTP and HTTPS values.")] String Uri;
    [Write, Description("User agent for the web request.")] String UserAgent;
    [Write, EmbeddedInstance("MSFT_KeyValuePair"), Description("Headers of the web request.")] String Headers[];
    [Write, EmbeddedInstance("MSFT_Credential"), Description("Specifies a user account that has permission to send the request.")] String Credential;
    [Read, ValueMap{"Present", "Absent"}, Values{"Present", "Absent"}, Description("Says whether DestinationPath exists on the machine")] String Ensure;
};
```

Additionally, you should use Test-xDscResource and Test-xDscSchema cmdlets from [Dsc Resource Designer](#) to automatically verify the resource and schema:

```
Test-xDscResource <Resource_folder>
```

```
Test-xDscSchema <Path_to_resource_schema_file>
```

For example:

```
Test-xDscResource ..\DSCResources\MSFT_xRemoteFile
Test-xDscSchema ..\DSCResources\MSFT_xRemoteFile\MSFT_xRemoteFile.schema.mof
```

3 Resource loads without errors

Once you verified that resource contains all necessary files and verified them using DSC Resource Designer, it's time to check whether resource module can be successfully loaded.

You can do it either manually, by running `Import-Module <resource_module> -force` and confirming that no errors occurred, or by writing test automation. In case of the latter, you can follow this structure in your test case:

```
$error = $null
Import-Module <resource_module> -force
If ($error.count -ne 0) {
    Throw "Module was not imported correctly. Errors returned: $error"
}
```

4 Resource is idempotent in the positive case

One of the fundamental characteristics of every DSC resource should be idempotence. It means that we can apply a DSC configuration containing that resource multiple times without changing the result beyond the initial application. For example, if we create a configuration which contains the following File resource:

```
File file {
    DestinationPath = "C:\test\test.txt"
    Contents = "Sample text"
}
```

After applying it for the first time, file test.txt should appear in C:\test folder. However, subsequent runs of the same configuration should not change the state of the machine (e.g. no copies of the test.txt file should be created).

To ensure our resource is idempotent we can repeatedly call Set-TargetResource when testing the resource directly, or call Start-DscConfiguration multiple times when doing end to end testing. The result should be the same after every run.

5 User modification scenario was tested

User modification is another common scenario worth testing out. It helps you verify that Set-TargetResource and Test-TargetResource function properly. Here are steps you should take to test it:

1. Start with the resource not in the desired state.
1. Run configuration with your resource
2. Verify Test-DscConfiguration returns True
3. Modify the resource out of the desired state
4. Verify Test-DscConfiguration returns false

Here's a more concrete example using Registry resource:

1. Start with registry key not in the desired state
1. Run Start-DscConfiguration with a configuration to put it in the desired state and verify it passes.
2. Run Test-DscConfiguration and verify it returns true
3. Modify the value of the key so that it is not in the desired state
4. Run Test-DscConfiguration and verify it returns false

6 Get-TargetResource functionality was verified using Get-DscConfiguration

Get-TargetResource should return details of the current state of the resource. Make sure you test it by calling Get-DscConfiguration after you apply the configuration and verifying that output correctly reflects the current state of the machine. It's important to test it separately, since any issues in this area won't appear when calling Start-DscConfiguration.

7 Resource was verified by calling Get/Set/Test-TargetResource functions directly

Make sure you test the Get/Set/Test-TargetResource functions implemented in your resource by calling them directly and verifying that they work as expected.

8 Resource was verified End to End using Start-DscConfiguration

Testing Get/Set/Test-TargetResource functions by calling them directly is important, but not all issues will be discovered this way. You should focus significant part of your testing on using Start-DscConfiguration or the pull server. In fact, this is how users will use the resource, so don't underestimate the significance of this type of tests.

Possible types of issues:

Credential/Session may behave differently because the DSC agent runs as a service. Be sure to test any features here end to end.

Verify error messages displayed by the resource make sense. For example, errors outputted by Start-DscConfiguration may be different than those displayed when calling the Set-TargetResource function directly.

9 Resource behaves correctly on all DSC supported platforms (or returns a specific error otherwise)

Resource should work on all DSC supported platforms (Windows Server 2008 R2 and newer). Make sure you install latest WMF (Windows Management Framework) on your OS to get the latest version of DSC. If resource by-design does not work on some of these platforms, a specific error message should be returned. Also, make sure your resource checks whether cmdlets you are calling are present on particular machine. Windows Server 2012 added a large number of new cmdlets that are not available on Windows Server 2008R2, even with WMF installed.

10 Resource functionality was verified on Windows Client (if applicable)

One very common test gap is verifying the resource only on server versions of Windows. Many resources are also designed to work on Client SKUs, so if that’s true in your case, don’t forget to test it on those platforms as well.

11 Get-DSCResource lists the resource

After deploying the module on the machine, calling Get-DscResource should list your resource among others as a result. If you can’t find your resource in the list, make sure that schema.mof file for that resource exists.

12 Resource module contains examples

If you intend to share the resource (which you hopefully do), creating quality examples which will help others understand how to use it. This is crucial, especially since many users treat sample code as documentation.

- First, you should determine the examples that will be included with the module – at minimum, you should cover most important use cases for your resource:
 - If your module contains several resources that need to work together for an end-to-end scenario, the basic end-to-end example would ideally be first.
 - The initial examples should be very simple -- how to get started with your resources in small manageable chunks (e.g. creating a new VHD)
 - Subsequent examples should build on those examples (e.g. creating a VM from a VHD, removing VM, modifying VM), and show advanced functionality (e.g. creating a VM with dynamic memory)
- Example configurations should be parameterized (all values should be passed to the configuration as parameters and there should be no hardcoded values):

```
configuration Sample_xRemoteFile_DownloadFile
{
    param
    (
        [string[]] $nodeName = 'localhost',

        [parameter(Mandatory = $true)]
        [ValidateNotNullOrEmpty()]
        [String] $destinationPath,

        [parameter(Mandatory = $true)]
        [ValidateNotNullOrEmpty()]
        [String] $uri,

        [String] $userAgent,

        [Hashtable] $headers
    )

    Import-DscResource -Name MSFT_xRemoteFile -ModuleName xPSDesiredStateConfiguration

    Node $nodeName
    {
        xRemoteFile DownloadFile
        {
            DestinationPath = $destinationPath
            Uri = $uri
            UserAgent = $userAgent
            Headers = $headers
        }
    }
}
```

It’s a good practice to include (commented out) example of how to call the configuration with the actual values at the end of the example script.

For example, in the configuration above it won’t be obvious for everyone that the best way to specify UserAgent is:

```
UserAgent = [Microsoft.PowerShell.Commands.PSUserAgent]::InternetExplorer
```

That’s why we should include comment with sample execution of the configuration:

```
<#
Sample use (parameter values need to be changed according to your scenario):

Sample_xRemoteFile_DownloadFile -destinationPath "$env:SystemDrive\fileName.jpg" -uri
"http://www.contoso.com/image.jpg"

Sample_xRemoteFile_DownloadFile -destinationPath "$env:SystemDrive\fileName.jpg" -uri
"http://www.contoso.com/image.jpg" `
-userAgent [Microsoft.PowerShell.Commands.PSUserAgent]::InternetExplorer -headers @{"Accept-
Language" = "en-US"}
#>
```

For each example, write a short description which explains what it does, and the meaning of the parameters.

Make sure examples cover most the important scenarios for your resource and if there’s nothing missing, verify that they all execute and put machine in the desired state.

13 Error messages are easy to understand and help users solve problems

Good error messages should be:

There: The biggest problem with error messages is that they often don’t exist, so make sure they are there ☺.

Easy to understand: Human readable, no obscure error codes

Precise: Describe what’s exactly the problem

Constructive: Advice how to fix the issue

Polite: Don’t blame user or make them feel stupid

Make sure you verify errors in End to End scenarios (using Start-DscConfiguration), because they may differ from those returned when running resource functions directly.

14 Log messages are easy to understand and informative (including –verbose, –debug and ETW logs)

Ensure that logs outputted by the resource are easy to understand and provide value to the user. Resource should output all information which might be helpful to the user, but more logs is not always better. You should avoid redundancy and outputting data which does not provide additional value – don’t make user go through hundreds of log entries in order to find what he’s looking for. Of course, no logs is not an acceptable solution for this problem either ;).

When testing, you should also analyze verbose and debug logs (by running Start-DscConfiguration with –verbose and –debug switches appropriately), as well as ETW logs. To see DSC ETW logs, go to event viewer and open the following folder: Applications and Services–Microsoft–Windows–Desired State Configuration. By default there will be Operational channel, but make sure you enable Analytic and Debug channels (you have to do it before running the configuration).

To enable Analytic/Debug channels, you can execute script below:

```
$statusEnabled = $true
# Use "Analytic" to enable Analytic channel
$eventLogFullName = "Microsoft-Windows-Dsc/Debug"
$commandToExecute = "wevtutil set-log $eventLogFullName /e:$statusEnabled /q:$statusEnabled "
$log = New-Object System.Diagnostics.Eventing.Reader.EventLogConfiguration $eventLogFullName
if($statusEnabled -eq $log.IsEnabled)
{
    write-Host -verbose "The channel event log is already enabled"
    return
}
Invoke-Expression $commandToExecute
```

15 Resource was tested with valid/invalid credentials

If your resource takes a credential as parameter:

Verify the resource works when Local System (or the computer account for remote resources) does not have access.

Verify the resource works with a credential specified for Get, Set and Test

If your resource accesses shares, test all the variants you need to support.

For example:

standard windows shares.

DFS shares.

AMBA shares (if you want to support Linux.)

16 Resource is not using cmdlets requiring interactive input

Get/Set/Test-TargetResource functions should be executed automatically and must not wait for user’s input at

any stage of execution (e.g. you should not use Get-Credential inside these functions). If you need to provide user's input, you should pass it to the configuration as parameter during the compilation phase.

17 Resource functionality was thoroughly tested

You are responsible to make sure the resource is behaving correctly, so test its functionality manually or, better yet, write automation. This checklist contains items which are important to be tested and/or are often missed. There will be bunch of tests, mainly functional ones which will be specific to the resource you are testing and are not mentioned here. Don't forget about negative test cases. **This will likely be the most time consuming part of the resource testing.**

18 Best practice: Resource module contains Tests folder with ResourceDesignerTests.ps1 script

It's a good practice to create folder "Tests" inside resource module, create ResourceDesignerTests.ps1 file and add there tests using Test-xDscResource and Test-xDscSchema for all resources in given module. This way we can quickly validate schemas of all resources from given modules and do sanity check before publishing.

For xRemoteFile, ResourceTests.ps1 could look as simple as:

```
Test-xDscResource ..\DSCResources\MSFT_xRemoteFile
Test-xDscSchema ..\DSCResources\MSFT_xRemoteFile\MSFT_xRemoteFile.schema.mof
```

19 Best practice: Resource folder contains resource designer script for generating schema

Each resource should contain a resource designer script which generates a mof schema of the resource. This file should be placed in <ResourceName>\ResourceDesignerScripts and be named

Generate<ResourceName>Schema.ps1

For xRemoteFile resource this file would be called GenerateXRemoteFileSchema.ps1 and contain:

```
$DestinationPath = New-xDscResourceProperty -Name DestinationPath -Type String -Attribute Key -
Description 'Path under which downloaded or copied file should be accessible after operation.'
$Uri = New-xDscResourceProperty -Name Uri -Type String -Attribute Required -Description 'Uri of a
file which should be copied or downloaded. This parameter supports HTTP and HTTPS values.'
$headers = New-xDscResourceProperty -Name Headers -Type Hashtable[] -Attribute Write -Description
'Headers of the web request.'
$userAgent = New-xDscResourceProperty -Name UserAgent -Type String -Attribute Write -Description
'User agent for the web request.'
$ensure = New-xDscResourceProperty -Name Ensure -Type String -Attribute Read -ValidateSet
"Present", "Absent" -Description 'says whether DestinationPath exists on the machine'
$credential = New-xDscResourceProperty -Name Credential -Type PSCredential -Attribute Write -
Description 'Specifies a user account that has permission to send the request.'
$certificateThumbprint = New-xDscResourceProperty -Name CertificateThumbprint -Type String -
Attribute Write -Description 'Digital public key certificate that is used to send the request.'
```

```
New-xDscResource -Name MSFT_xRemoteFile -Property @($DestinationPath, $Uri, $headers, $userAgent,
$ensure, $credential, $certificateThumbprint) -ModuleName xPSDesiredStateConfiguration2 -
FriendlyName xRemoteFile
```

20 Best practice: Resource supports -whatif

If your resource is performing "dangerous" operations, it's a good practice to implement -whatif functionality. After it's done, make sure that whatif output correctly describes operations which would happen if command was executed without whatif switch.

Also, verify that operations does not execute (no changes to the node's state are made) when -whatif switch is present.

For example, let's assume we are testing File resource. Below is simple configuration which creates file "test.txt" with contents "test":

```
configuration config
{
    node localhost
    {
        File file
        {
            Contents="test"
            DestinationPath="C:\test\test.txt"
        }
    }
}
```

If we compile and then execute the configuration with the -whatif switch, the output is telling us exactly what would happen when we run the configuration. The configuration itself however was not executed (test.txt file was not created).

```
PS C:\test> Start-DscConfiguration -path .\config -ComputerName localhost -wait -verbose -whatif
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, "methodName' =
SendConfigurationApply,'className' = MSFT_DSCLocalConfigurationManager,'namespaceName' =
root/Microsoft/Windows/DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer CHARLESX1 with user sid
S-1-5-21-397955417-626881126-188441444-5179871.
What if: [CHARLESX1]: LCM: [ Start Set ]
What if: [CHARLESX1]: LCM: [ Start Resource ] [[File]file]
```

```
What if: [CHARLESX1]: LCM: [ Start Test ] [[File]file]
What if: [CHARLESX1]: [[File]file] The system cannot find the file specified.
What if: [CHARLESX1]: [[File]file] The related file/directory is: C:\test\test.txt.
What if: [CHARLESX1]: LCM: [ End Test ] [[File]file] in 0.0270 seconds.
What if: [CHARLESX1]: LCM: [ Start Set ] [[File]file]
What if: [CHARLESX1]: [[File]file] The system cannot find the file specified.
What if: [CHARLESX1]: [[File]file] The related file/directory is: C:\test\test.txt.
What if: [CHARLESX1]: [C:\test\test.txt] Creating and writing contents and setting attributes
.
What if: [CHARLESX1]: LCM: [ End Set ] [[File]file] in 0.0180 seconds.
What if: [CHARLESX1]: LCM: [ End Resource ] [[File]file]
What if: [CHARLESX1]: LCM: [ End Set ]
VERBOSE: [CHARLESX1]: LCM: [ End Set ] in 0.1050 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
```

This wraps up our checklist. Please keep in mind that this list is not exhaustive, but it covers many important issues which we encountered while designing, developing and testing DSC resources. Having a checklist helps to ensure we didn't forget about any of those aspects and in fact, we use it at Microsoft when developing DSC resources ourselves.

If you developed guidelines and best practices which you use for writing and testing DSC resources, please share them in a comment.

Karol Kaczmarek, Software Engineer

Windows PowerShell Team


 Tweet 17

 Like 6

 Share 9

 Save this on Delicious

Comments



16 Dec 2014 6:42 AM

Donovan Brown

\$error = \$null is not valid. It should be \$error.Clear()

Also the example resource xRemoteFile fails Test-xDscResource for use of Microsoft.Management.Infrastructure.CimInstance[]