

PowerShell Desired State Configuration

for DevOps and ALM
practitioners

Visual Studio ALM Rangers



Microsoft



Visual Studio

Config as Code – Foreword

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, you should not interpret this to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Microsoft grants you a license to this document under the terms of the Creative Commons Attribution 3.0 License. All other rights are reserved.

© 2014 Microsoft Corporation.

Microsoft, Active Directory, Excel, Internet Explorer, SQL Server, Visual Studio, and Windows are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Table of Contents

Foreword	4
Introduction	6
Setting the context for PowerShell DSC.....	7
Overview	7
Flow – Resource Development.....	12
Checklist – Resource Development.....	12
Bug report and resolution process.....	14
Resource library	15
Practical implementations.....	15
References and tooling - where to find which gems.....	17
Interesting Questions and Answers	18
Walkthrough - File Server & Share Custom Resource	20
Know what your DSC configuration is going to do.....	20
Decompose your configuration into the smallest logical steps.....	20
Generating the initial outline	21
Understanding Ensure Present and Absent.....	22
Configuring the CreateFileShare resource.....	22
Configuring the SetSharePermission resource.....	24
Unit Testing the Resources	24
Creating the configuration.....	26
Deploying your custom DSC resource and configuration.....	28
Executing your configuration on the target server.....	29
Troubleshooting	30
Walkthrough - Deploy TFS 2013 using DSC.....	32
Introduction.....	32
Building the Resources	35
Steps to Configure TFS on a Single Server.....	37
Appendix – PowerShell 101	38
Code Samples.....	49
Walkthrough - File Server & Share Custom Resource	49
Walkthrough – Deploy TFS 2013 using PowerShell DSC	69
In Conclusion	72
Quick Reference Sheets / Posters	73

Foreword

We hear the term “DevOps” frequently in current press, popular books, and presentations from many product vendors. The DevOps concept focuses on how to build deep integration between Development and Operations, organizations who have historically been at odds. This has been brought about, in part, by the need to increase the velocity of delivery from development into production, while ensuring that the Production environments are always running.

Many of the difficulties are well understood, and documented in those same DevOps articles, books and presentations. One of the big problems to solve is ensuring that everyone is working with the same processes, systems, and tools, throughout the product development and delivery cycle.

The Development team needs to know that changes they deliver to Operations will work in Production, so they need to develop and test their changes in environments that match Production systems. The same automation scripts Operations use in Production should be used in the early test VMs and pre-Production test labs. Any changes to the system that were required by the Developers would be made in the automation scripts directly, and included with their updates. Those scripts then document the modified environment, improving the ability to review changes with the Operations team before they would be applied.

The Operations team needs to know that what they are deploying has already been proven to work in Production, or spend critical time re-validating changes during “outage windows”. Automating system setup and configuration is not new, but historically the scripts were monolithic, requiring the Operations team to modify the core code when moving from test labs into Production. Separating the actions in the automation code from the environmental data (such as server names & accounts) simplifies the updates, and reduces the risk of error. In addition, by restricting changes to what is in automation scripts, updates can be validated in advance, rolled back if something bad occurs, and all common systems can be ensured to be consistent.

The development of PowerShell Desired State Configuration (DSC) was based in part on these requirements. PowerShell DSC is a set of extensions to the PowerShell language that enable creating PowerShell functions that are repeatable, and by design scripts into are separated into components for repeatable actions, structural configuration, and environmental data. In PowerShell DSC, the separate components are called Resources, Configurations, Configuration Data. This separation of action from structure from data is a new way of scripting, and requires changes in thinking. PowerShell is new to most Developers, and the style of coding is new to experienced PowerShell users.

This is not simply changing about the structure of automation scripts. With any system built using PowerShell DSC Configurations, the PowerShell DSC engine periodically tests the current state of the machine against the desired state, monitoring and alerting, and optionally automatically correcting configuration drift. Configuration changes that do not come via changes to the PowerShell DSC code are quickly identified and resolved. In addition, Visual Studio is integrating PowerShell DSC into the Development process and the release management pipeline – something that will be covered in future articles. Packages built using Visual Studio can now include the DSC resources and configurations needed to deploy the system.

The change to DevOps has broad impact, and we cannot cover it all in one place. This document focuses on getting users familiar with PowerShell DSC. For those unfamiliar with PowerShell, it gives some guidance on where and how to learn the capabilities of PowerShell, and leverage it in their coding projects. For those already familiar with PowerShell, it explains how DSC is different, and how to leverage the tools and contributions from

others to rapidly adopt these concepts when developing their own PowerShell DSC Resources and Configurations. It also directs users to repositories of open-source repositories where they will find DSC Resources and examples that are available for their use, which will help speed the adoption of DSC.

Integration of the code required for a smooth transition between Development and Operations is becoming a reality in many places. While PowerShell DSC helps to enable this, it does require some changes to the design and development of automation scripts. We hope this document will make the transition faster and easier.

Keith Bankston – Senior Program Manager, WSSC WS PM USA

Introduction

This guidance delivers practical, scenario-based guidance for the use of PowerShell Desired State Configuration (DSC) to design, develop, and deploy custom configuration as code resources. We guide you through the basics and practical walkthroughs, based on a real-world proof-of-concept deployments and experience from the ALM Rangers.

Intended audience

We expect the majority of our audience personas to be **Doris** – Developer, whereby **Dave** – TFS Server Administrator, Jane – Infrastructure specialist and **Bill** – ALM Consultant can leverage the guidance. See [ALM Rangers Personas and Customer Types](#)¹ for more information on these and other personas.

The guide assumes a good knowledge of the PowerShell, Windows, Visual Studio and Team Foundation Server (TFS) and an operational administration mindset – in other words, intermediate to advanced developers and possibly TFS Administrators, who are looking at the option of automating the configuration and deployments of their environments.

Visual Studio ALM Rangers

The Visual Studio ALM Rangers are a special group with members from the Visual Studio Product group, Microsoft Services, Microsoft Most Valuable Professionals (MVP), and Visual Studio Community Leads. Their mission is to provide out-of-band solutions to missing features and guidance. A growing Rangers Index is available [online](#)².

Contributors

Brian Blackman, Giulio Vian, Hamid Shahid, Jahangeer, Jeff Levinson, Jim Szubryt, Keith Bankston, Mathias Olausson, Mattias Skold, Richard Fennell, Rob Jarratt, Rui Melo, Shawn Cicoria, Tiago Pascoal Willy-Peter Schaub, and Vinicius Hana Scardazzi

Using the sample source code, Errata and support

All source code in this guide is available for download from [Config as Code for DevOps and ALM practitioners](#)³ CodePlex site and [Script resources for IT professionals](#)⁴.

Additional ALM Rangers Resources

Understanding the ALM Rangers – <http://aka.ms/vsarunderstand>

Visual Studio ALM Ranger Solutions – <http://aka.ms/vsarsolutions>

¹ <http://aka.ms/treasure4>

² <http://aka.ms/vsarindex>

³ <https://vsardevops.codeplex.com/>

⁴ <http://gallery.technet.microsoft.com/scriptcenter>.

Setting the context for PowerShell DSC

In this section, we will explore some of the recommended best practices when writing DSC resources using PowerShell and the base template used by the Visual Studio ALM Rangers and provide a simple checklist we use to validate that we have met our minimum quality bar for our resources.

NOTE

For a more comprehensive list and coverage, we strongly recommend the book [Windows PowerShell Best Practices](#) ⁵, by [Ed Wilson](#) ⁶ and the [Windows PowerShell Best Practices and Patterns: Time to Get Serious](#) ⁷, by [Don Jones](#) ⁸.



Figure 1 - Windows PowerShell TechEd 2014 North America event

If you are not experienced in **PowerShell**, we recommend that you visit the Appendix – PowerShell 101, page 38

Overview

What is DSC

DSC is short for Desired state Configuration, and is a new way to manage deployments, that is fundamental different from traditional, instruction-based deployment. Instead of providing instructions for how to deploy your environment, you simply describe the Desired State you want to achieve in a PowerShell script file. You can then deploy or rather apply a Desired State Configuration to a server using the PowerShell engine.

"Desired State Configuration is Microsoft's technology, introduced in Windows Management Framework 4.0, for declarative configuration of systems. At the risk of oversimplifying a bit, DSC lets you create specially formatted text files that describe how you should configure your system. You then deploy those files to the actual systems, and they magically configure themselves as directed. At the top level, DSC isn't programming – it's just listing how you want a system to look." (Steve Murawski, Don Jones, Stephen Owen)

The DSC script format is declarative and constitute simply of a list of **resources** and their desired properties. A resource is just a couple of PowerShell scripts to support DSC for a configuration object. Microsoft provides DSC (both out of the box and for installation) resources.

NOTE

Please refer to **Quick Reference Cheat sheet / Posters**, page 73, for information on a visual cheat sheet poster that guides you through the artefacts and samples (DSC – **Overview**).

⁵ http://www.amazon.com/Windows-PowerShell-Best-Practices-Wilson/dp/0735666490/ref=tmm_pap_title_0?ie=UTF8&qid=1393958424&sr=1-1

⁶ http://www.amazon.com/s/ref=ntt_athr_dp_sr_1?encoding=UTF8&field-author=Ed%20Wilson&ie=UTF8&search-alias=books&sort=relevancerank

⁷ <http://channel9.msdn.com/Events/TechEd/NorthAmerica/2014/DCIM-B418#fbid=>

⁸ <http://channel9.msdn.com/Events/Speakers/Don-Jones>

A PowerShell DSC **module** contains a **manifest**, one or more **resources**, each with a **script** and **schema**. We will explore the granularity of resources and the design of DSC modules as part of the guidance and walkthrough contained herein.

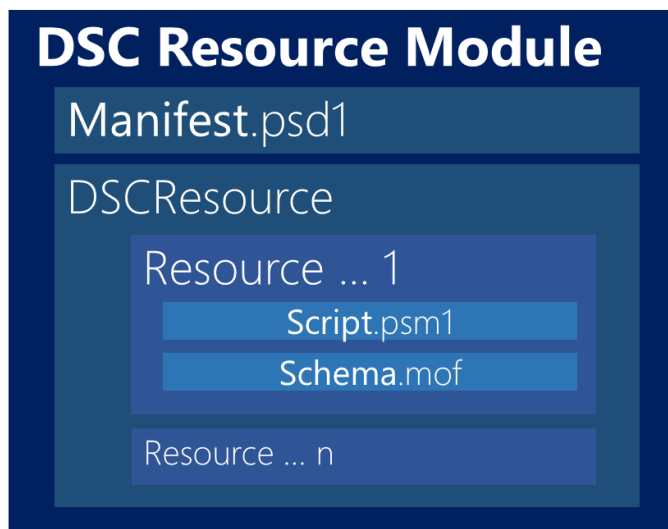


Figure 2 – DCS module with resource artefacts

The following configuration example uses resources provided out of the box to manage Window features, roles, and servers.

Code 1 – DSC Script: Manage Windows Features

```

Configuration ContosoWeb
{
    # A Configuration block can have zero or more Node blocks
    Node "Server001"
    {
        # Next, specify one or more resource blocks
        # WindowsFeature is one of the built-in resources you can use in a Node block
        # This example ensures the Web Server (IIS) role is installed
        WindowsFeature IIS
        {
            Ensure = "Present" # To uninstall the role, set Ensure to "Absent"
            Name = "Web-Server"
        }
        WindowsFeature ASP
        {
            Ensure = "Present" # To uninstall the role, set Ensure to "Absent"
            Name = "Web-Asp-Net45"
        }
        # File is a built-in resource you can use to manage files and directories
        # This example ensures files from the source directory are present in the destination directory
        File SiteCatalog
        {
            Ensure = "Present" # You can also set Ensure to "Absent"
            Type = "Directory" # Default is "File"
            Recurse = $true
            SourcePath = $WebsiteFilePath # This is a path that has web files
            DestinationPath = "C:\inetpub\wwwroot"
            # The path where we want to ensure the web files are present
            DependsOn = "[WindowsFeature]MyRoleExample"
            # This ensures that MyRoleExample completes successfully before this block runs
        }
    }
}
  
```


This very simple script, uses the [WindowsFeature](#) ⁹ (a friendly name for the Role resource) to define the features that need to be present in the *Server001* node.

In this case, it ensures we install the Web Server Role (with ASP.NET 4.5 features).

We specify that feature should be present in the configuration by specifying the **Present** value in the Ensure key. (We can also specify **Absent**, in that case when we run the configuration we will be sure the role will not be part of the machine configuration).

If you want to get the list of available features, you can run the PowerShell command [Get-WindowsFeature](#) ¹⁰

We also use the [File](#) ¹¹ resource to copy the folder content (specified in the **WebsiteFilePath** variable) to the *C:\inetpub\wwwroot* (IIS default root).

With such a simple configuration script, we ensure we have a basic web role machine with a running website.

Using PowerShell DSC for defining deployment in your project

PowerShell DSC is very useful for deployments and continuous deployment, as you do not need to worry or care about the current state of the servers to which you are deploying. Your DSC deployment scripts will only focus on the desired state you want to accomplish. Furthermore, DSC can be faster, as it only affects those things that you need to adjust to meet your requirements.

Separate data from configuration

PowerShell DSC can also separate your configuration data from the configuration logic, which is very useful in situations where you deploy to different environments in a release pipeline. For, example, when you have a bunch of machines, which perform the same functions, it is easy to add a machine with little effort and the exact same configuration using DSC.

It is also highly beneficial when you have a bunch of machines that perform the same function (for example a load balanced web farm). This allows you to add easily on demand equally configured machines that perform the same rule any differences in the configuration.

You can do this by parameterizing the configuration.

Code 2 – Parameterized Configuration

```
Configuration ContosoWeb
{
    param ($AppTier, $DataTier)

    # A Configuration block can have zero or more Node blocks
    Node $AppTier
    {
        WindowsFeature IIS
        {
            Ensure = "Present" # To uninstall the role, set Ensure to "Absent"
            Name = "Web-Server"
        }
        WindowsFeature ASP
        {
            Ensure = "Present" # To uninstall the role, set Ensure to "Absent"
            Name = "Web-Asp-Net45"
        }
        File SiteCatalog
        {
            Ensure = "Present" # You can also set Ensure to "Absent"
            Type = "Directory" # Default is "File"
            Recurse = $true
            SourcePath = $WebsiteFilePath
            DestinationPath = "C:\inetpub\wwwroot"
            DependsOn = "[WindowsFeature]ASP"
        }
    }
}
```

⁹ <http://technet.microsoft.com/en-us/library/dn282127.aspx>

¹⁰ <http://technet.microsoft.com/en-us/library/jj205469.aspx>

¹¹ <http://technet.microsoft.com/en-us/library/dn282129.aspx>

```
}
}
Node $DataTier
{
    ...
}
```

Notice that after the Configuration keyword we no longer have the **Server001** node name, but we now have the **AppTier** variable that will be passed when the configuration is run. You can also specify the configuration data by using the **Node** function.

Using PowerShell DSC: Push and Pull Modes

PowerShell DSC can be set up to run in both Push and Pull modes. The difference is that in Push mode, DSC actively pushes the configuration to the target servers. See [Push and Pull Configuration Modes](#)¹² for more information on push and pull servers.

The Push model is the simplest one and gives you control over the events and configurations, since they start manually.

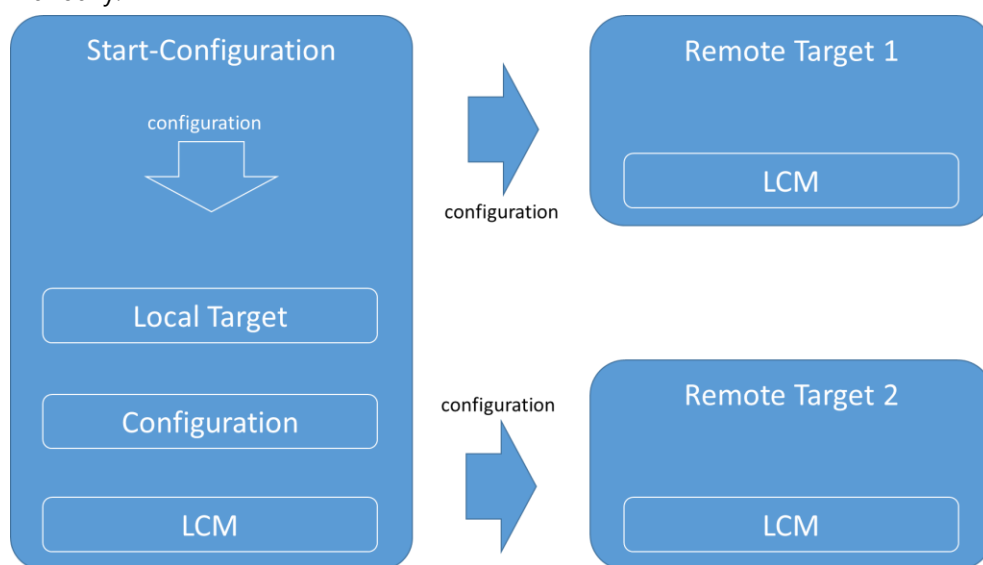


Figure 3 – Push Model

In the pull model, the target servers automatically pull the current configuration from a DSC Pull Server.

The Pull model requires software installation and configuration, but provides scalability and conformability. The pull server allows a single repository of configurations. You can apply these to any number of VMs delivering a given role, hence giving scalability. Because this is a VM, we recheck its state against this pull server on a regular schedule to avoid configuration drift if you manually update a machine's state.

¹² <http://blogs.msdn.com/b/powershell/archive/2013/11/26/push-and-pull-configuration-modes.aspx>

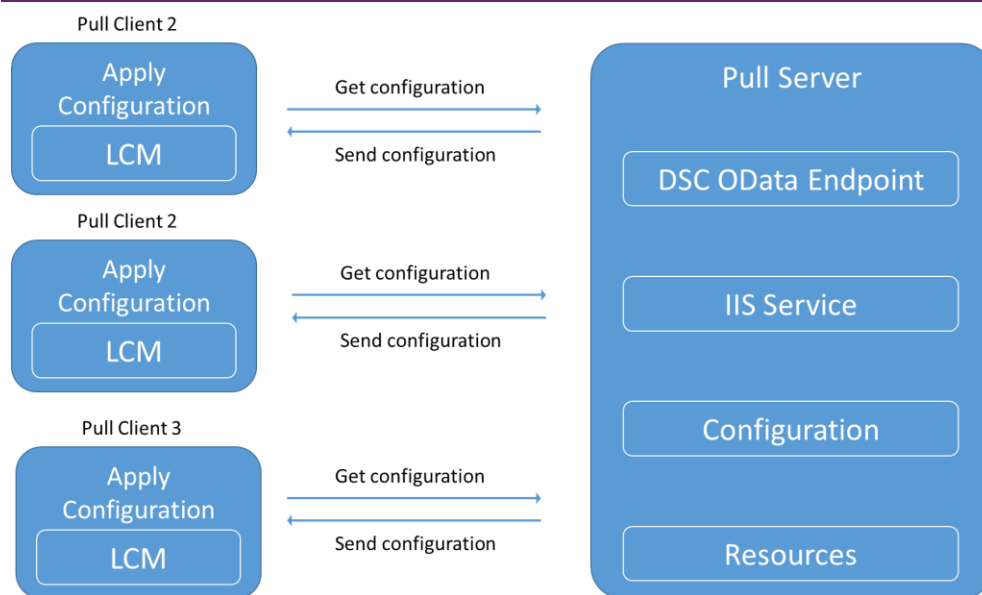


Figure 4 – Pull Model

Developing PowerShell DSC resources

Apart from using Microsoft and community resources, you might sometimes need to create your own PowerShell DSC resource for a subsystem or third party solution. As previously mentioned a DSC resource is just a PowerShell script module implementing the DSC interface.

To make a DSC resource you need to plan and define the resource, as well as implement three PowerShell functions

- **Get-TargetResource** – This method should receive the keys of the resource and return a hash table with all the resource properties as configured on the system.
- **Test-TargetResource** – This method receives the keys and properties (as defined in the schema) and checks if the current configuration for the resource exists in the system. If any of the property values do not match the resource instance, \$false should be returned and \$true otherwise.
- **Set-TargetResource** - This method will be called to guarantee the resource instance matches the property values. This method must be idempotent so that after it DSC runs the method, DSC must ensure the resource instance is exactly as defined. It needs to be able to create a resource from scratch or to make some changes to an existing resource to make sure its state matches the state defined in the configuration.

You also have to define your resource schema (MOF format), which defines the properties of the resource will receive in the configuration.

When you have your script module (with the three functions) and the schema defined you will need to create your module manifest (your module can implement more than one resource).

Flow – Resource Development

The following flow diagram summarizes the setup, development and deployment steps you should consider when you start building your custom DSC resources.

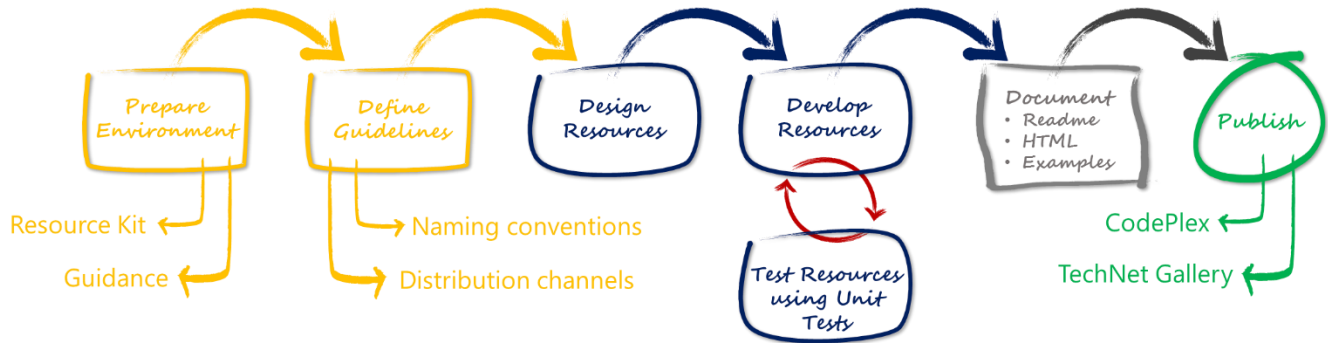


Figure 5 - Resource Development Flow

Checklist – Resource Development

Category	Description	
Naming	<ul style="list-style-type: none"> When writing DSC Resources, please use the following naming conventions [CompanyNamePrefix]_[prefix][ResourceName] Where, <ul style="list-style-type: none"> CompanyNamePrefix is the prefix for your company. Prefix is an optional prefix you would want to give your resource to indicate whether experimental (x) or community (c). ResourceName is the name of your resource. The resource name must be noun and ideally the resource that you are looking to configure. Naming Schema <ul style="list-style-type: none"> Resources <ul style="list-style-type: none"> A_xB <ul style="list-style-type: none"> x → Indicates that resource is experimental A → Four digit organization abbreviation, i.e. MSFT, VSAR¹³ B → Unique and descriptive name Example: MSFT_xIPAddress, VSAR_xSample Modules <ul style="list-style-type: none"> xB <ul style="list-style-type: none"> x → Indicates that resource is experimental B → Unique and descriptive name Example: xNetworking 	<input type="checkbox"/>
Folders	<ul style="list-style-type: none"> Folder Structure created by DSC Resource Designer from the DSC Resource Kit. <ul style="list-style-type: none"> \$env: psmodulepath (folder) - <ModuleName> (folder) - <ModuleName>.psd1 (file, mandatory) - DSC Resources (folder) - <ResourceName1> (folder) - <ResourceName1>.psd1 (file, optional) - <ResourceName1>.psm1 (file, required) - <ResourceName1>.schema.mof (file, required) 	<input type="checkbox"/>

¹³ VSAR identifies the module as being from the Visual Studio ALM Rangers.

Config as Code – Setting the context for PowerShell DSC

Category	Description	
	<ul style="list-style-type: none"> - <ResourceName2>(folder) <ul style="list-style-type: none"> - <ResourceName2>.psd1 (file, optional) - <ResourceName2>.psm1 (file, required) - <ResourceName2>.schema.mof (file, required) • Module <ul style="list-style-type: none"> ○ <ModuleName>.psd1 manifest file ○ <ModuleName>_Readme.html • DSC Resources <ul style="list-style-type: none"> ○ Folder per resource, containing: <ul style="list-style-type: none"> ▪ A_x<resource>.schema.mof file ▪ A_x<resource>.psm1 script file • Examples <ul style="list-style-type: none"> ○ One file one example ○ Example is based on what resource does, demonstrating basic features 	<input checked="" type="checkbox"/>
Module	<ul style="list-style-type: none"> • The <ResourceName>.psd1 file defines the metadata for custom resource module. Example: <pre>@{ # Version number of this module. ModuleVersion = '' # ID used to uniquely identify this module GUID = '' # Author of this module Author = '' # Company or vendor of this module CompanyName = '' # Copyright statement for this module Copyright = '' # Description of the functionality provided by this module Description = '' # Minimum version of the Windows PowerShell engine required by this # module PowerShellVersion = '' # Minimum version of the common language runtime (CLR) required by # this module CLRVersion = '' # Modules that must be imported into the global environment prior to # importing this module RequiredModules = @("") # Modules to import as nested modules of the module specified in # RootModule/ModuleToProcess NestedModules = @("") # Functions to export from this module FunctionsToExport = @("Get-TargetResource", "Set-TargetResource", "Test-TargetResource") # Cmdlets to export from this module CmdletsToExport = '' # HelpInfo URI of this module HelpInfoURI = '' }</pre> • <ModuleName>_ReadMe.html or HELP document sections <ul style="list-style-type: none"> ○ Introduction ○ Installation ○ Requirements ○ Description ○ Examples ○ Versions 	<input type="checkbox"/>
MOF Schema	<ul style="list-style-type: none"> • The MOF Schema file defines the properties of the resource and must be defined in the following way <pre>[ClassVersion("Resource version"), FriendlyName("Resource Friend Name")] class ResourceName : OMI_BaseResource</pre> 	

Config as Code – Setting the context for PowerShell DSC

Category	Description	
	<pre>{ [Key Write Read Required, Description("")] Type PropertyName; };</pre> <p>where,</p> <ul style="list-style-type: none"> ○ ResourceName is the name of the DSC Resource. ○ PropertyName is the name of one of the properties in the resource. ○ Each property can have one of the following qualifier <ul style="list-style-type: none"> ▪ Key - uniquely identifies the resource. ▪ Write - value assigned when invoking the resource. ▪ Read - value is read-only. ▪ Required - property is required. ○ Each property must have a meaningful description. ○ At least one parameter in schema must be marked as Key. ○ EmbeddedInstance class should be a DSC resource class i.e. derived from OMI_BaseResource class. 	<input type="checkbox"/>
Resource Module Header Comments	<ul style="list-style-type: none"> • The resource header should define the version, copyright, and overview as a bare minimum. • Example: <pre>#requires -version <PowerShell> #Copyright © <Year,> <Company Name>. License conditions. #Resource Overview - A couple of lines about the purpose and functionality of the resource</pre> 	<input type="checkbox"/>
Resource Module	<ul style="list-style-type: none"> • The resource module script contains the logic of resource and must include the following three methods. • Each of the function should have exactly the same parameters as defined in the MOF file. <ul style="list-style-type: none"> ○ Get-TargetResource: This function checks the state of the resource instances whose keys are passed as parameter. The function returns a hash table listing all of the Key and Required resource properties. ○ Set-TargetResource: This function attempts to configure the resource as per the desired state. ○ Test-TargetResource: This function checks the state of the resource against the desired state. If it is same, the function returns true otherwise it returns false 	<input type="checkbox"/>
Versioning	<ul style="list-style-type: none"> • All modules versioned • <ModuleName>.psd1 used A.B.C.D convention, where: <ul style="list-style-type: none"> A → Major B → Revision, large change C → Revision, minor change D → non-functional cosmetic change 	<input type="checkbox"/>
Testing	<ul style="list-style-type: none"> • The "test" function in each resource needs to be as performant as possible. 	<input type="checkbox"/>

Table 1 – DSC resource best practices checklist

Bug report and resolution process

If a bug is found in a Microsoft provided resource

- Go to [Microsoft Connect for PowerShell](https://connect.microsoft.com/powershell) ¹⁴
- Select Report a Bug and then the type of bug you want to report (code, documentation etc.).
- In the Bug report form, be sure to add the module and/or resource name to the title where applicable and then describe the problem with as much context as possible.

¹⁴ <http://connect.microsoft.com/powershell>

Microsoft will generally resolve the bug and release the fix in an upcoming update.

If a bug is found in community supplied resource

@SORRY FOR THE INCONVENIENCE! THIS SECTION WILL BE COMPLETED IN V1 UPDATES OR THE V2 RELEASE@

In the interim please post bugs to the resource kit location where you found the resource or contact us.

Forum

If you have questions in general about PowerShell DSC then the following forums provides a great way to interact with other DSC experts.

- [Microsoft TechNet forum](#) ¹⁵
- [PowerShell.org DSC forum](#) ¹⁶

Resource library

The best place to find out about new resources available from Microsoft is the [DSC Resource Kit](#) ¹⁷, on the TechNet Script Center. This contains all experimental resources for users who have installed the latest Windows Management Framework, you can use the Find-Module command to locate PowerShell and DSC updates that are available in the online gallery. This gallery will follow the module naming convention identified above, so you can do things like the following in PowerShell:

```
PS C:\windows\system32> find-module x*
```

Version	Name	DateUpdated	Description
2.0	xActiveDirectory	7/29/2014 1:42:57 AM	The xActiveDirectory module is a ...
0.1.1	xAzure	7/27/2014 10:12:51 AM	The xAzure module is a set of DSC...
1.2.1	xComputerManagement	7/29/2014 1:42:57 AM	The xComputerManagement module is...
1.1.2	xDatabase	7/27/2014 10:37:52 PM	The xDatabase module is a part of...
1.1	xDhcpServer	7/29/2014 1:42:57 AM	The xDhcpServer module is a part ...
1.0	xDnsServer	7/27/2014 10:12:51 AM	The xDnsServer module is a part o...
2.0	xDscDiagnostics	7/28/2014 9:37:54 AM	Module to help in reading details...
1.1.1.1	xDscResourceDesigner	7/29/2014 12:37:59 PM	The xDscResourceDesigner module i...
2.6.0.0	xEXOUUserAvailability	7/27/2014 10:12:51 AM	xEXOUUserAvailability can help you...
1.1.1	xFailOverCluster	7/27/2014 10:12:51 AM	The xFailOverCluster module is a ...
2.1.1	xHyper-V	7/29/2014 1:42:57 AM	The xHyper-V module is a part of ...
0.2.16.1	xJea	7/29/2014 4:12:59 PM	Module with DSC Resources for Jus...
1.0.0.0	xMySQL	7/29/2014 4:08:00 PM	The xMySQL module is a part of th...

Practical implementations

PowerShell DSC within a small to medium team environment

PowerShell DSC provides a mechanism for achieving configuration as code on windows platform. You are determining infrastructure with code when you define a script that describes in an objective and practical manner the desired configuration for any given infrastructure.

Configuration as code leverages many possibilities, especially when it comes down to automation, among other benefits. However, it is crucial to keep in mind some inherent aspects that may potentially lead to problems and difficulties. The main idea here is to show those aspects, how they may affect you, and what can be you can do about them.

¹⁵ <http://social.technet.microsoft.com/Forums/windowsserver/en-US/home?forum=winserverpowershell>

¹⁶ <http://powershell.org/wp/forums/forum/dsc-desired-state-configuration>

¹⁷ <http://gallery.technet.microsoft.com/DSC-Resource-Kit-All-c449312d>

Treat configuration files as first-class citizens

One of the benefits of using PowerShell DSC is that configuration files can be self-explanatory and replaces the need of documents. Thus, DSC needs to consider the configuration files as first-class citizens and treat them as such. In practice, this means that:

- They must be standard of use
- They must be kept up-to-date
- They must always represent the current and proper state of configuration

Put your configuration files under source control

Putting your configuration files under source control will allow you to maintain a history of all the changes done to them. In addition, having them under source control facilitates backup processes, avoiding situations where scripts may be lost. With the proper branching strategies, we can make changes in parallel as well optimize productivity.

Use configuration files to describe end states

Always keep in mind that the configuration described will be tested and ensured against a machine. Using partial and/or sequential configurations burdens the whole deployment process by adding unnecessary orchestration steps. A configuration file has to be sufficient, unique, and self-contained. Configurations may be nested, see this article: [Reusing Existing Configuration Scripts in PowerShell Desired State Configuration](http://blogs.msdn.com/b/powershell/archive/2014/02/25/reusing-existing-configuration-scripts-in-powershell-desired-state-configuration.aspx) ¹⁸

Test your configurations

Like any other code, testing is crucial for delivering reliable solutions. The same applied to configuration files: thoroughly test them in test environments before deploying them to production environments. Make sure that everything works as expected; otherwise, any problems brought by malfunctioning configurations may be hard to track down.

Use a pull server as a single point-of-truth

A pull server will keep all the production configurations in one place, with the machines configured by it constantly pull the configurations they use. This leverages updating configurations, since you would just have to update a configuration file at the pull server and the machines that use it will pull it and update their configuration themselves. Imagine that you need to update 100 servers using the same configuration - a pull server allows you to do it easily, instead of updating one by one.

Don't make manual changes to machines

When machines are manually changed, this means you are not tracking these changes into the configuration script – you would have to do this manually, which might be unreliable. Manually changing a machine will present some challenging situations when applying the same configuration to another machine, since there will be no way to ensure that the configurations are the same.

Consider rollback scenarios

Sometimes, you may need to roll back the configuration changes for various reasons. For those situations, most of the time you just have to pick up a previous version from the configuration script and apply it, which means you must save versions of your PowerShell DSC configurations to support rollback. However, reality tends to be more complex, and you will need to re-run the previous configuration scripts.

¹⁸ <http://blogs.msdn.com/b/powershell/archive/2014/02/25/reusing-existing-configuration-scripts-in-powershell-desired-state-configuration.aspx>

References and tooling - where to find which gems

Videos

- [TechEd 2014, A Practical Overview of Desired State Configuration](#) ¹⁹

Blogs

- [Building Clouds Blog](#) ²⁰

Tutorials

- [Windows PowerShell Desired State Configuration Overview](#) ²¹ gives a good overview of DSC content on TechNet.
- [Introducing PowerShell Desired State Configuration \(DSC\)](#) ²² for a high-level overview of DSC
 - Desired State Configuration Blog Series
 - Part 1 - [Information about DSC](#) ²³
 - Part 2 - [Authoring DSC Resources when Cmdlets Already Exist](#) ²⁴
 - Part 3 - [Testing DSC Resources](#) ²⁵
 - Part 4 - [How-To use PowerShell DSC from your workstation to test PowerShell JEA, in Azure](#) ²⁶
 - Part 5 - [Point in Time List of DSC Resources](#) ²⁷
- [Building a Desired State Configuration Pull Server](#)

Other

- [DSC Resource Kit](#) ²⁸
- Windows Management Framework 5
 - [Windows Management Framework 5.0 Preview May 2014](#) ²⁹
 - [Windows Management Framework V5 Preview Blog](#) ³⁰

¹⁹ <http://channel9.msdn.com/Events/TechEd/NorthAmerica/2014/DCIM-B417>

²⁰ <http://blogs.technet.com/b/privatecloud/>

²¹ <http://technet.microsoft.com/en-us/library/dn249912.aspx>

²² <http://blogs.technet.com/b/privatecloud/archive/2013/08/30/introducing-powershell-desired-state-configuration-dsc.aspx>

²³ <http://blogs.technet.com/b/privatecloud/archive/2014/04/25/desired-state-configuration-blog-series-part-1-learning-about-dsc.aspx>

²⁴ <http://blogs.technet.com/b/privatecloud/archive/2014/05/02/powershell-dsc-blog-series-part-2-authoring-dsc-resources-when-cmdlets-already-exist.aspx>

²⁵ <http://blogs.technet.com/b/privatecloud/archive/2014/05/09/powershell-dsc-blog-series-part-3-testing-dsc-resources.aspx>

²⁶ <http://blogs.technet.com/b/privatecloud/archive/2014/05/16/powershell-blog-series-part-4-how-to-use-powershell-dsc-from-your-workstation-to-test-powershell-jea-in-azure.aspx>

²⁷ <http://blogs.technet.com/b/privatecloud/archive/2014/06/06/powershell-dsc-blog-series-part-5-point-in-time-list-of-dsc-resources.aspx>

²⁸ <http://gallery.technet.microsoft.com/scriptcenter/DSC-Resource-Kit-All-c449312d>

²⁹ <http://www.microsoft.com/en-us/download/details.aspx?id=42936>

³⁰ <http://blogs.technet.com/b/windowsserver/archive/2014/04/03/windows-management-framework-v5-preview.aspx>

Interesting Questions and Answers

How does a third party review their schema?	18
How is it possible to externalize common configurations?	18
How to deal with a process pinned in memory?	19
How to decide the scope of a module?	19
How to know when configuration has actually completed?	19
Sometimes I have to reboot in order for a change to my resource script to take effect. Which process or service do I need to cycle to avoid a full reboot?	19
What to do with helper functions shared between modules?	19
When you start a DSC config you get log messages on the console. If the machine reboots during configuration, where can you see those log messages after a reboot?	19

How does a third party review their schema?

The best way to ensure you are developing your PowerShell DSC Resource properly is to use the [DSC Resource Designer Tool](#) ³¹. This module includes tools such as

- New-DscResource: generates the code skeleton and MOF for a new DSC Resource
- Test-DscResource, which checks the resource against the MOF, and the basic rules for DSC.

Watch for updates to this module in the DSC Resource kit, as the plan is for new tools will be added to this module.

How is it possible to externalize common configurations?

Context: Scenario, you are automating several kind of "roles" (over 10 different types) and each role will have several nodes. Most of the roles, have configurations that are common to them (e.g.: Basic IIS capabilities, NLB, and so on). You want to externalize all these configurations into an external file to avoid repetition among other things. How?

The best way to create support for common configurations is to use Composite Resources. Composite Resources are actually DSC Configurations that are built to be called from another Configuration. See [Reusing Existing Configuration Scripts in PowerShell Desired State Configuration](#) ³².

Not part of this topic, but related: When there are multiple organizations that have ownership over the configuration of a single system, creating a single DSC Configuration can be difficult. The classic example is when the storage, networking, and database teams must collaborate to build out a SQL Server instance. To solve this, the Windows Management Framework (WMF) 5.0 September Preview ³³ has added support for Partial Configurations. This allows the Configuration to be managed as multiple components, drawn from a single Pull Server. There is good information in the Release Notes for the WMF 5.0 September Preview on how to take advantage of this. Also see Composite Resources, in The DSC Book ³⁴.

³¹ <http://blogs.msdn.com/b/powershell/archive/2013/11/19/resource-designer-tool-a-walkthrough-writing-a-dsc-resource.aspx>

³² <http://blogs.msdn.com/b/powershell/archive/2014/02/25/reusing-existing-configuration-scripts-in-powershell-desired-state-configuration.aspx>

³³ <http://blogs.msdn.com/b/powershell/archive/2014/09/04/windows-management-framework-5-0-preview-september-2014-is-now-available.aspx>

³⁴ aka.ms/dscPsoBook

How to deal with a process pinned in memory?

Context: This problem would manifest if you cannot debug into a module and old code you just deleted seems to be running.

If you encounter issues with resources that are cached in memory, you have two options of resetting your environment:

- Kill the WMI provider host that is hosting your PowerShell or PowerShell ISE window, rather than unloading and re-loading it.
- Or-
- Create and run a script to unload cached modules

Code 3 – Sample unload / import script

```
# Check that the required PowerShell module is loaded if it is remove it as it might be an older version
if ((get-module -name AlmRangers.Tfs.Utilities) -ne $null)
{
    remove-module AlmRangers.Tfs.Utilities
}
import-module .\AlmRangers.Tfs.Utilities.psml -verbose
```

How to decide the scope of a module?

When deciding the scope of a module consider if the actions you wish to perform on a system can be expressed as verbs on a noun in sentence e.g. Enable sharing on server X. If you can express your intent in this manner then you probably have the correct scope of the DSC module. If not you probably need to consider different scope

How to know when configuration has actually completed?

The [Windows Management Framework 5.0 Preview](http://blogs.msdn.com/b/powershell/archive/2014/09/04/windows-management-framework-5-0-preview-september-2014-is-now-available.aspx) ³⁵ has added a new cmdlet called Get-DSCConfigurationStatus. This will allow a user to check the status of current or recently-executed DSC Configurations. You can get information about the cmdlet from the Release Notes document for WMF 5.0 September.

Sometimes I have to reboot in order for a change to my resource script to take effect. Which process or service do I need to cycle to avoid a full reboot?

Have this command (cmd) file on your desktop:

```
robocopy <your source>\Modules\ "%ProgramFiles%\WindowsPowerShell\Modules" /MIR
net stop winmgmt
net start winmgmt
```

What to do with helper functions shared between modules?

See example in the DSC Resource Kit ³⁶, for example, xDatabase for good examples.

When you start a DSC config you get log messages on the console. If the machine reboots during configuration, where can you see those log messages after a reboot?

For messages other than debug and verbose, you should use the Event Viewer and refer to Microsoft-Windows-Desired State Configuration/Operational. Also consider using the DebugView ³⁷ tool from sysinternals ³⁸.

³⁵ <http://blogs.msdn.com/b/powershell/archive/2014/09/04/windows-management-framework-5-0-preview-september-2014-is-now-available.aspx>

³⁶ <http://gallery.technet.microsoft.com/scriptcenter/DSC-Resource-Kit-All-c449312d>

³⁷ <http://technet.microsoft.com/en-us/sysinternals/bb896647>

³⁸ <http://technet.microsoft.com/en-US/sysinternals>

Walkthrough - File Server & Share Custom Resource

This walkthrough will show you how to create and deploy a PowerShell Desired State Configuration module, which will create a file share and assign permissions for that file share. A practical use for this could be configuring a drop folder for a build server or a shared folder for specific team members. As a practical matter, there is an xSmbFileShare DSC that are part of the core set of modules.

NOTE

Please refer to **Quick Reference Cheat sheet / Posters**, page 73, for information on a visual cheat sheet poster that guides you through this (DSC – **Custom Resource** Walkthrough) and other walkthroughs.

Know what your DSC configuration is going to do

Before authoring configuration scripts, it is critical to understand the resources they will affect and whom else they will affect. The results of this question will lead to wildly different implementations of the configuration.

Important terminology:

- **Shared Resource** (i.e. a folder is a resource): A resource accessed by multiple users and processes for purposes which may or may not align with the purposes of the specific DSC configuration being created
- **Private Resource**: A resource required and accessed only for the purpose for which it is being configured
- **Module**: In DSC terms a module is a package which contains one or more resources
- **Resource (DSC)**: A specific script which performs an action that furthers the configuration of a target system
- **Configuration (DSC)**: A single file containing end state configuration of a target system which passes values to one or more DSC Resources

The first question to ask is, “**is this a shared resource or a private resource?**” The answer to this question will affect every piece of script. The rule here is that when modifying a shared resource, target a minimum configuration, and when modifying a private resource target an exact configuration.

What does this mean exactly?

SITUATION

In this walkthrough, the desired actions are to create a shared folder and set certain permissions on it. For argument's sake, John requires Read permission to the file share. When the configuration runs it is determined, that John has Change permissions. Should John be removed from the Change group and re-assigned to the Read group or is the best approach to say, John is in the Change group which means he has read permissions already, leave the resource alone?

This is the simplest way to illustrate the decision making process when it comes to creating a DSC module. In this walkthrough, the answer to the question is to set a minimum configuration. To do otherwise may affect some other operation that John may need to perform.

Decompose your configuration into the smallest logical steps

Trying to do too much in a configuration becomes problematic from a maintenance perspective and potentially the required number of scripts. Consider the current scenario – to create a file share and set permissions. You can do this in single step but it becomes more complex and is not as flexible.

In a single step process, writing extra code is required to see if the file share exists and then set permissions. In addition, if the user wants to just set permissions and not create a file share at all, they will not have any option to do that. The simplest solution is to provide more choice.

In order to get around this scenario you can create a PowerShell DSC Module with multiple resources and then set the configuration such that the processing of a module is dependent on the processing of another module. This allows you to create re-usable packages where people can pick what they want to configure.

For this walkthrough, the decomposition will require a **CreateFileShare** and **SetSharePermissions** resources in the module.

Once the configuration is decomposed, determine how to use existing PowerShell commands to configure the target system. Knowing this is critical when generating the initial outline of the DSC module, because the script must declare the parameters for the resources. This DSC configuration is going to call the **New-Item**, **Grant-SmbShareAccess**, and **Remove-SmbShareAccess** commands.

Generating the initial outline

This part of the walkthrough will generate the skeleton structure of the module. Microsoft has created the **xDscResourceDesigner** (x stands for experimental so this is subject to change). It is not required but it makes the generation process much faster. In addition, the use of **Test-xDscResource** and **Test-xDscSchema** are not necessarily required because this generates the correct skeleton automatically.

Step	Instructions
1 Start Environment ☐ - Done	<ul style="list-style-type: none"> Download and setup supporting resources, for example DSC Resource Kit ³⁹, as mentioned on page 17. Start PowerShell ISE
2 Create schema script ☐ - Done	<ul style="list-style-type: none"> Create the Generate_cFileShare_Schema.ps1 script. See page 49 for an example. Note the naming convention is "Generate_" + DSC Module name + "_Schema.ps1" This file should be version controlled in case it needs to be re-generated or schema changes are required in the future
3 Run schema script ☐ - Done	<ul style="list-style-type: none"> Run the script The Path argument in the New-xDscResource command determines the output location of the resource. The result of this command is that the following folder and file structure is generated: <pre> WindowsPowerShell (folder) Modules (folder) cFileShare (folder) DSCResources (folder) VSAR_cCreateFileShare (folder) VSAR_cCreateFileShare.psml (file) VSAR_cCreateFileShare.schema.mof (file) VSAR_cSetSharePermissions (folder) VSAR_cSetSharePermissions.psml (file) VSAR_cSetSharePermissions.schema.mof (file) </pre>
4 Create manifest file ☐ - Done	<div style="display: flex; align-items: flex-start;"> <div style="background-color: red; color: white; padding: 5px; writing-mode: vertical-rl; transform: rotate(180deg); font-weight: bold; margin-right: 10px;">WARNING</div> <div> <p>As of the time of this writing there is a known bug in the xDscResourceDesigner – it should have output one additional file named cFileShare.psdl in the cFileShare folder. This will be corrected in a future release if the DSC Resource Kit.</p> <p>To generate this, run the following PowerShell command:</p> <pre>New-ModuleManifest -Path "%ProgramFiles%\WindowsPowerShell\Modules\cFileShare\cFileShare.psdl"</pre> </div> </div>

³⁹ <http://gallery.technet.microsoft.com/scriptcenter/DSC-Resource-Kit-All-c449312d>

Step	Instructions
	<ul style="list-style-type: none"> This will generate the missing file in the correct location.

Table 2 – Generate initial outline

Understanding Ensure Present and Absent

Before writing the actual resources, it helps to understand **ensure present** and **ensure absent**.

Almost anything on a system – an application, file, registry key, permissions, etc. – may need to be confirmed as existing (Present), or explicitly not existing (**Absent**). If a resource exposes **Ensure** as a property, you have control over the existence of that object in the final state of the system. In this example, if the configuration should be **Present** it will create the resource and assign permissions. If the configuration should be **Absent**, it will remove the resource.

There are no specific steps to follow here, but keep this in mind because the **Set-TargetResource** and **Test-TargetResource** methods will both have an if...then statement to handle this condition.

Configuring the CreateFileShare resource

The generated output is a set of skeleton files that need some modifications.

The *.**schema.mof** files and the *.**psd1** files do not need to be edited (the cFileShare.psd1 may be edited to update version numbers and provide more detailed information but the generated output is perfectly acceptable at this point). This means that the *.psm1 files do need to be edited. Each psm1 file contains three specific methods:

- Get-TargetResource
- Set-TargetResource
- Test-TargetResource

This section describes each methods use and function. This section describes the edits to both the VSAR_cCreateFileShare.psm1 and the VSAR_cSetSharePermissions.psm1 files.

Get-TargetResource

This method checks to determine the state of the system and returns the key and required data in a hash table.

NOTE

The **Get-TargetResource** and the other Set-* and Test-* methods should **not ever throw an exception** as part of the normal process that is not caught and handled. For example, the code below passes the common parameter – ErrorAction SilentlyContinue to the Get-SmbShare. This is because Get-SmbShare will throw an exception if the share does not exist. If this happens, the configuration will not succeed. However, if Get-SmbShare does throw an exception, the variable **\$shareInfo** will be null. Therefore, checking the value of \$shareInfo after this call is made will determine whether the given share exists. A try...catch block can also be used but –ErrorAction SilentlyContinue seems to work best in most cases.

In addition, if the process should **stop because of an error**, throwing an exception is a perfectly acceptable way of managing the situation. It writes Exceptions thrown in this manner to the Desired State Configuration Operational log (discussed in the troubleshooting section below).

Always include verbose statements here so the end user can get an exact description of what is happening and that way if the resource throws an exception they can determine exactly where it threw the exception but they can also see the variables in use along the way.

Config as Code – Walkthrough - File Server & Share Custom Resource

Step	Instructions
1 Implement Get-* method <input type="checkbox"/> - Done	<ul style="list-style-type: none"> Open the generated VSAR_cCreateFileShare.psm1 file Implement the Get-TargetResource method, using the sample VSAR_cCreateFileShare Get-TargetResource, page 50.

Table 3 – What should the Get-TargetResource method do and why

Test-TargetResource

The **Test-TargetResource** method determines whether the state of the system matches the configuration requested. For example, the CreateFileShare resource needs to create a share.

The Get-TargetResource returned a hash table saying that the share does or does not exist simply by looking for the share. The Test-TargetResource looks at it in the context of the configuration you requested. If you requested that the file share be present, then the method will return true if it does and false otherwise. It is not simply getting the state of the system it is returning whether the system matches your desired configuration.

Step	Instructions
1 Implement Test-* method <input type="checkbox"/> - Done	<ul style="list-style-type: none"> Implement the Test-TargetResource method, using the sample VSAR_cCreateFileShare Test-TargetResource, page 51. Design the Test-TargetResource to run quickly. Each time DSC scans the system for drift, it runs the Test-TargetResource, so design it to execute quickly, and have the least performance impact on the system possible.

Table 4 – What should the Test-TargetResource method do and why

Set-TargetResource

Set-TargetResource does the work of getting the system into the desired configuration. Remember to take into account the Present and Absent aspects of the Ensure parameter.

WARNING

This gets into a very grey area when dealing with a shared resource. In this case, ensuring a file share is absent means that there is only one recourse – removing the file share. This is somewhat in conflict with the statements made earlier regarding a minimum state versus an absolute state. One important thing to note is that when pushing configurations to a target machine, you can only push (or pull) one configuration per server.

In other words, it is not possible for two parties to create different configurations that might overwrite each other – all of the configurations for a single machine must be in a single file – thoroughly review the configuration file and version control it. We cover the configuration file later in this document.

Step	Instructions
1 Implement Set-* method <input type="checkbox"/> - Done	<ul style="list-style-type: none"> Implement the Set-TargetResource method, using the sample VSAR_cCreateFileShare Set-TargetResource, page 52.
2 Save changes <input type="checkbox"/> - Done	<ul style="list-style-type: none"> Save the modified the generated VSAR_cCreateFileShare.psm1 file

Table 5 – Configuring the system with Set-TargetResource

? Why is there no error handling around creating the file share?

? What if the file share already exists?

? Why does the code not check to see if the file share exists before trying to create it?

The answer is the Test-TargetResource method. Based on the results of the Test-TargetResource method, it may not even call the Set-TargetResource method. If the configuration calls for the file share to exist and it does, then the configuration skips the Set-TargetResource method – the system is already in the desired state. Therefore, the fact that this code is being called, by definition means that the share does not already exist.

Configuring the SetSharePermission resource

Step	Instructions
1 Open resource file	<ul style="list-style-type: none"> Open the generated VSAR_cSetSharePermission.psm1 file
2 Implement Get-*, Test-* and Set-* methods ☐ - Done	<ul style="list-style-type: none"> Implement the Get-*, Test-* and Set-* methods. Get-Target method, see page 53. Test-Target method, see page 54. Set-Target method, see page 57.
2 Save changes ☐ - Done	<ul style="list-style-type: none"> Save the modified generated VSAR_cCreateFileShare.psm1 file.

At this point, the DSC Module and Resources are complete and ready to deploy.

Unit Testing the Resources

The ALM Rangers are big fans of unit testing, quality and “doing it right the first time.” In line with that, it is appropriate to unit test resources as best as can be done. Scripts that configure system resources require a bit more time and energy to unit test but it is well worth it. The major reason for this is that it is virtually impossible to mock these unit tests. How you perform unit testing is entirely up to you, as long as you do unit testing.

You can find various resources on the web for unit testing PowerShell scripts – they may or may not work with a given resource. This walkthrough shows a manual process for creating the unit tests. Alternatively, the completed unit test files are included in the download.

These files are **VSAR_cCreateFileShare_UnitTests.ps1**, page 59 and **VSAR_cSetSharePermissions_UnitTests.ps1**, page 62.

This is more an “anatomy” of a unit test than a walkthrough.

Code 4 – VSAR_cCreateFileShare unit test extract

```
#Unit tests for VSAR_cCreateFileShare
Import-Module "%ProgramFiles%\WindowsPowerShell\Modules\cFileShare\DSCResources\VSAR_cCreateFileShare"

#Variable Declarations
$ShareName = "TestShare"
$Path = "C:\Test"

$PassCounter = 0
$FailCounter = 0

#####
```

```
#
# Tests for Get-TargetResource
#
#####

#####
# Test #1 - If share exists, Ensure returns "Present"
#####

#Setup for Test #1
$SetupResult = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
if (!$SetupResult)
{
    New-SmbShare -Path $Path -Name $ShareName
}

$Result = Get-TargetResource -ShareName $ShareName -Path $Path

if ($Result.Ensure -ne "Present")
{
    $FailCounter += 1
    "Test 1 Failed"
}
else
{
    $PassCounter += 1
    "Test 1 Passed"
}
```

Overview of the unit test sample script

- The first line clearly defines what this file tests – a good rule of thumb is one file per resource
- Import the resource which is to be tested
- Declare any variables which will be used throughout the unit tests
 - The Pass/Fail counters are used to provide a summary at the end of the test
- Test each section of the resource separately – as noted here this test is part of the Get-TargetResource tests
- Comment the test with a number and what is being tested
 - Note that this can also be done with each test being in a separate method and the method names providing the detail as is done with standard unit tests
- Perform any test setup required and mark that section as the setup section
- Run the test
- Examine the result and increment the appropriate counter and output the result

The output of the test run, using the VSAR_cCreateFileShare_UnitTests.ps1 script, see page 59, should be as follows:

```
PS C:\windows\system32> C:\Users\DemoUser\Desktop\DSC Walkthrough Final
Files\VSAR_cCreateFileShare_UnitTests.ps1
Test 1 Passed
Test 2 Passed
Test 3 Passed
Test 4 Passed
Test 5 Passed
Test 6 Passed

Passed: 6, Failed: 0

PS C:\windows\system32>
```

NOTE

These tests should always pass no matter how many times you run them, since they perform their own setup.

NOTE

Write the DSC configuration and perform these tests on system as similar as possible to the target systems.

NOTE

Run these tests as an Administrator to ensure they will not fail. For example, this test creates a folder in the root of the c drive – this requires administrative permissions and the tests will fail if run as a regular user

Creating the configuration

NOTE

This section assumes that you have a pull server and a target server already configured. See [Push and Pull Configuration Modes](#)⁴⁰ and [Building a Desired State Configuration Pull Server](#)⁴¹ for more information on push and pull servers.

The configuration actually tells the target server what configuration it needs to be in and therefore passes values to the previously created PowerShell DSC Resources.

Step	Instructions
1 Create config script ☐ - Done	<ul style="list-style-type: none"> Start the PowerShell ISE and enter the following in a new script tab alternatively, use the provided file and make changes according to the highlighted areas below as discussed in the following section): <pre>Configuration CreateDropShare { Import-DscResource -ModuleName cFileShare Node appserver { VSAR_cCreateFileShare CreateShare { ShareName = 'DropShare' Path = 'C:\DropShare' Ensure = 'Present' } VSAR_cSetSharePermissions SetPermissions { ShareName = 'DropShare' DependsOn = '[cCreateFileShare]CreateShare' Ensure = 'Present' FullAccessUsers = @('dx\jeff') ChangeAccessUsers = @('dx\steven') ReadAccessUsers = @('dx\shad') } } } CreateDropShare</pre> A brief tour of some of the information present in the configuration file: <ul style="list-style-type: none"> CreateDropShare is the name given to the configuration – it can be anything you want and just serves as a way to identify it Import-DscResource imports the previously created module and all resources The Node line specifies the target system that will be configured – in this case the server name is called appserver VSAR_cCreateFileShare is the name of the Resource this section will pass configuration information too. You must give the Resource a name, for example CreateShare, although the name can actually be anything. <div> <p>NOTE</p> <p>With your cursor on the resource name (VSAR_cCreateFileShare) pressing Ctrl+SpaceBar will show the list of parameters for the resource.</p> </div>

⁴⁰ <http://blogs.msdn.com/b/powershell/archive/2013/11/26/push-and-pull-configuration-modes.aspx>

⁴¹ <http://powershell.org/wp/2013/10/03/building-a-desired-state-configuration-pull-server/>

Config as Code – Walkthrough - File Server & Share Custom Resource

Step	Instructions
	<div>NOTE</div> <p>When entering parameter names in this section, typing in any letter and pressing Ctrl+SpaceBar will bring up the IntelliSense list of parameters to select (for instance, type 'a' and Ctrl+SpaceBar and you can select <code>ShareName</code>)</p> <ul style="list-style-type: none"> In the SetPermissions section <code>DependsOn</code> absolutely ensures that the configuration section that this section relies on will be configured first The last line, CreateDropShare simply instructs the DSC engine to run the configuration in order to create (not perform) a configuration.
2 Set working directory ☐ - Done	<ul style="list-style-type: none"> Change the working directory to another location (the default is <code>C:\Windows\System32</code>, in this example the directory is <code>c:\users\[username]\desktop\dsc</code>)
3 Save script ☐ - Done	<ul style="list-style-type: none"> As a best practice, save the above configuration script to a file and version control it since it represents the formatted configuration of the server in question.
4 Run script ☐ - Done	<ul style="list-style-type: none"> Once the file is saved, run it and verify that the output matches the following: <pre>PS C:\Users\[username]\Desktop\DSC> .\CreateDropShare.ps1 Directory: C:\Users\[username]\Desktop\DSC\CreateDropShare Mode LastWriteTime Length Name ---- - -a--- 8/14/2014 2:43 PM 2358 appserver.mof</pre>
5 Verify results ☐ - Done	<ul style="list-style-type: none"> Verify that the result is a new folder called CreateDropShare, which contains a single file – appserver.mof. <div>NOTE</div> <p>The output file name will be <code>[server name].mof</code> not <code>appserver.mof</code> unless this is the name of the server.</p>
6 Connect to remote server ☐ - Done	<ul style="list-style-type: none"> The configuration has been generated, but some additional steps are required. The unique identifier of the target system needs to be determined. While <code>appserver</code> is the name of the server, it is not guaranteed to be unique. Each server is identified by a GUID that is created when you configure the target server as the target of a pull server. Connect to the remote server with RDP or a remote PowerShell session.
7 Determine server GUID ☐ - Done	<ul style="list-style-type: none"> Once connected to the remote system, open a PowerShell prompt and run Get-DscLocalConfigurationManager. The results should look like the following if you configure everything correctly. The import part of this is the <code>ConfigurationID</code> which is needed to finalize the configuration: <pre>[dxdemo.cloudapp.net]: PS C:\Users\jeff.DX\Documents> Get-DscLocalConfigurationManager AllowModuleOverwrite : False CertificateID : ConfigurationID : 1f970101-df17-444f-914d-ac6cb5f246cf ConfigurationMode : ApplyOnly ConfigurationModeFrequencyMins : 30 Credential : DownloadManagerCustomData : {MSFT_KeyValuePair (key = "ServerUrl"), MSFT_KeyValuePair (key = "AllowUnsecureConnection") } DownloadManagerName : WebDownloadManager RebootNodeIfNeeded : False RefreshFrequencyMins : 15 RefreshMode : Pull</pre>

Config as Code – Walkthrough - File Server & Share Custom Resource

Step	Instructions
	<div>PSComputerName :</div> <div>NOTE Your GUID will be different. Please use that GUID in place of this one!</div>
8 Rename schema file ☐ - Done	<ul style="list-style-type: none"> Rename the appserver.mof file 1f970101-df17-444f-914d-ac6cb5f246cf.mof Format is: ([GUID].mof)
9 Run command ☐ - Done	<ul style="list-style-type: none"> In PowerShell, navigate to the previously created CreateDropShare folder and run the following command: New-DSCChecksum .\1f970101-df17-444f-914d-ac6cb5f246cf.mof This will generate a second file in the folder with the name 1f970101-df17-444f-914d-ac6cb5f246cf.mof.checksum ([GUID].mof.checksum).

Table 6 – Creating the configuration

The configuration is complete. It is now ready to deploy and test.

NOTE As you start assigning GUID's to various servers, it's helpful to create a central repository of server names and GUID's to make it easier and faster to look up – especially as you may not have permission to each server.

Deploying your custom DSC resource and configuration

Deploying the PowerShell DSC resource involves not only deploying the configuration but the resource itself. The resource, when deployed to the pull server is in a zip format and both the resource and you can simply copy the configuration to the pull server.

Step	Instructions
1 Navigate to modules folder ☐ - Done	<ul style="list-style-type: none"> Before deploying a custom resource, it needs to be zipped and a checksum created for the zipped file. <div>WARNING Because of a peculiar bug in the current version of DSC, resources can only be zipped in the manner shown in this walkthrough – all other attempts will result in failures.</div> <ul style="list-style-type: none"> Navigate to the %ProgramFiles%\WindowsPowerShell\Modules folder
2 Zip module ☐ - Done	<ul style="list-style-type: none"> Right-click the cFileShare folder and select Send To > Compressed (zipped) folder You must run as an admin in order to create this file in this folder. A save location prompt will be displayed and it defaults to the desktop which poses no problem.
3 Rename zip file ☐ - Done	<ul style="list-style-type: none"> Rename the zip file cFileShare_1.0.zip The DSC engine requires this particular naming convention. The version number at the end of this file must match the module version number in the ALM_FileShare.psd1 file
4 Open PS window ☐ - Done	<ul style="list-style-type: none"> Open a PowerShell window and navigate to the location of the zip file

Config as Code – Walkthrough - File Server & Share Custom Resource

Step	Instructions
5 Generate checksum ☐ - Done	<ul style="list-style-type: none"> Enter the command <code>New-DSCChecksum .\ cFileShare_1.0.zip</code> This will result in a new file being created in the same folder named <code>cFileShare_1.0.zip.checksum</code> At this point the module and configuration are ready to be deployed to the pull server
6 Remote into pull server ☐ - Done	<ul style="list-style-type: none"> Remote into the pull server and navigate to <code>%ProgramFiles%\WindowsPowerShell\DscService\Modules</code> folder Copy the <code>cFileShare_1.0.zip</code> and <code>cFileShare_1.0.zip.checksum</code> to this folder
7 Copy custom resource to pull server ☐ - Done	<ul style="list-style-type: none"> Copy the <code>[GUID].mof</code> and <code>[GUID].mof.checksum</code> file to the <code>%ProgramFiles%\WindowsPowerShell\DscService\Configuration</code> folder. The configuration has now been deployed to the pull server and by default will be picked up within 30 minutes (this is the <code>ConfigurationModeFrequencyMin</code> value from above)

Table 7 – Deploying your custom DSC resource and configuration

Executing your configuration on the target server

Frequently when testing, waiting 30 minutes for a configuration to kick off is too long. You can manually kick off the configuration manually for a “quick” test. We recommend that you use **Push** mode to do testing and once you have validated the config, switch to **Pull** mode.

WARNING

Do not run `Start-DSCConfiguration` from the pull server. Doing this will change the state of the target server from pull to push which means it will no longer poll the pull server for updates (if it is configured in this way).

Step	Instructions
1 Logon to target server ☐ - Done	<ul style="list-style-type: none"> Log on to the target server. Open a PowerShell prompt (this can also be accomplished through a remote PowerShell session)
2 Invoke CimMethod ☐ - Done	<ul style="list-style-type: none"> Run the following command: <pre>Invoke-CimMethod -ComputerName appserver -Namespace root/microsoft/windows/desiredstateconfiguration ` -Class MSFT_DscLocalConfigurationManager -MethodName PerformRequiredConfigurationChecks ` -Arguments @{Flags = [UInt32]1} -Verbose</pre> This command forces the target server to verify its configuration against the pull server. When it finds a change, it will run the configuration. The resulting message should look like the following: <pre>PS C:\Users\jeff.DX\Desktop> Invoke-CimMethod -ComputerName appserver -Namespace root/microsoft/windows/desiredstateconfiguration ` -Class MSFT_DscLocalConfigurationManager -MethodName PerformRequiredConfigurationChecks ` -Arguments @{Flags = [UInt32]1} -Verbose VERBOSE: Performing the operation "Invoke-CimMethod: PerformRequiredConfigurationChecks" on target "MSFT_DscLocalConfigurationManager." VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, 'methodName' = PerformRequiredConfigurationChecks, 'className' = MSFT_DscLocalConfigu rationManager, 'namespaceName' = root/microsoft/windows/desiredstateconfiguration'. VERBOSE: An LCM method call arrived from computer APPSERVER with user sid S-1-5-21- 1100927510-3167963274-3869165624-500. VERBOSE: [APPSERVER]: [] Starting consistency engine.</pre>

Config as Code – Walkthrough - File Server & Share Custom Resource

Step	Instructions
	<pre> VERBOSE: [APPSERVER]: [] Consistency check completed. ReturnValue PSComputerName ----- appserver VERBOSE: Operation 'Invoke CimMethod' complete. 0 </pre>
3 Verify ☐ - Done	<ul style="list-style-type: none"> Verify that the DropShare folder was created and all permissions assigned correctly

Table 8 – Executing your DSC on the target server

Troubleshooting

What to do if you get an error message

Inevitably, errors will occur when deploying to the target server. This is a quick bit of help for what to do.

Step	Instructions
1 Review results ☐ - Done	<ul style="list-style-type: none"> On the target server, when executing the command in step two above you may get a result that looks like the following: <pre> PS C:\Users\jeff.DX\Desktop> Invoke-CimMethod -ComputerName appserver -Namespace root/microsoft/windows/desiredstateconfiguration ` -Class MSFT_DscLocalConfigurationManager -MethodName PerformRequiredConfigurationChecks ` -Arguments @{Flags = [UInt32]1} -Verbose VERBOSE: Performing the operation "Invoke-CimMethod: PerformRequiredConfigurationChecks" on target "MSFT_DscLocalConfigurationManager." VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, 'methodName' = PerformRequiredConfigurationChecks, 'className' = MSFT_DscLocalConfigu rationManager, 'namespaceName' = root/microsoft/windows/desiredstateconfiguration'. VERBOSE: An LCM method call arrived from computer APPSERVER with user sid S-1-5-21- 1100927510-3167963274-3869165624-500. Invoke-CimMethod : The SendConfigurationApply function did not succeed. At line:1 char:1 + Invoke-CimMethod -ComputerName appserver -Namespace root/microsoft/windows/desir ... + ~~~~~ + CategoryInfo : NotSpecified: (root/microsoft/...gurationManager:String) [Invoke-CimMethod], CimException + FullyQualifiedErrorId : MI RESULT 1,Microsoft.Management.Infrastructure.CimCmdlets.InvokeCimMethodCommand + PSComputerName : appserver VERBOSE: Operation 'Invoke CimMethod' complete. </pre> On its face, it is a particularly unhelpful message as nothing tells you what is wrong.
2 Open Event Viewer ☐ - Done	<ul style="list-style-type: none"> Open the Event Viewer. Expand the Event Viewer > Applications and Services Log > Desired State Configuration. Select the Operational log (at this time it is the only log there)
3 Review Log ☐ - Done	<ul style="list-style-type: none"> When selecting the Operational log, for each exception that occurred you will see four error entries. The bottom (first) error in the list is the only that concerns troubleshooting a failed configuration deployment.
4 Review sample error	<ul style="list-style-type: none"> When running this deployment, I purposely entered an invalid username (one that was not in active directory) and the error log showed the following: <pre> Job {807B3C9E-603E-4B43-AF20-C3F3710BEBE3} : </pre>

Config as Code – Walkthrough - File Server & Share Custom Resource

Step	Instructions
☐ - Done	<p>This event indicates that a non-terminating error was thrown when DSCEngine was executing Set-TargetResource on VSAR_cSetSharePermissions provider. FullyQualifiedErrorId is Windows System Error 1332, Grant-SmbShareAccess. ErrorMessage is No mapping between account names and security IDs was done.</p> <ul style="list-style-type: none"> • This error is straightforward – there is a problem in the VSAR_cSetSharePermissions resource and the error is with the Grant-SmbShareAccess. • It happens to be that “No mapping between account names...” is generated when an invalid ID is provided. • However, the error does not tell you which ID was incorrect. • To fix it, look at the DropShare folder, which was created, and examine the permissions. • The configuration is run in order to it is easy to look down the list of already configured users and figure out which one failed. • Once you have identified the correct account and fixed it, fix the configuration file, regenerate it, and re-perform the steps from generating the Configuration section onward. • After the configuration is re-invoked everything will be configured correctly

Table 9 – What to do if you get an error message

What to do if you cannot debug into a module and old code you just deleted seems to be running?

See **How to deal with a process pinned in memory** in the **Interesting Questions and Answers** section, page 18, for more information.

Walkthrough - Deploy TFS 2013 using DSC

Introduction

In this walkthrough, we look at the process of deploying Microsoft Team Foundation Server 2013 on a base installation of Windows Server 2012 R2. The premise is that we need to get a single-server TFS deployment up and running quickly. This could be for demonstration purposes, or for supporting multiple isolated virtualized development environments. To keep the deployment simple, the scope is limited to a single-server deployment, with an application tier and build; we have excluded reporting and SharePoint integration for the time being.

NOTE

Please refer to **Quick Reference Cheat sheet / Posters**, page 73, for information on a visual cheat sheet poster that guides you through this (DSC – **Implementation** Walkthrough) and other walkthroughs.

In order to build this, the main components you must deploy are:

1. The SQL Server Database Engine
2. SQL Server Analysis Services
3. Team Foundation Server Application Tier
4. Team Foundation Server Build

Consequently, we mapped each of these to a PowerShell DSC resource. The sections below discuss the resources that we built to support these components.

NOTE

You must be a local administrator on the server where you run the configuration. You will also need:

- We will use TFS Administrator as the credentials for the account. We use this account to run the TFS Configuration Wizard and the account requires local administration rights on the server.
- [Optional]. The credentials of an account used to access the share where the install media are located.

SQL Server Resources

The Resource Kit already has a resource for **SQL Server**. However, we found number of deficiencies, which made it simpler to rewrite the resource. The problems identified with the Resource Kit resource for SQL Server (xSqlServerInstall) were as follows:

1. It provided a parameter that allows the configuration of the set of features to be installed. However, it does not provide a way to supply the other parameters that might be required by other features, meaning that installation could never succeed. It is also not particularly intentional design just to give access to the raw list of features.
2. It always reboots the machine, no matter what feature was installed, and without checking to see whether the installation succeeded. This would result in an infinite reboot cycle if the installation fails.
3. It imposes SQL Server authentication, and provides no option for integrated security, which is usually preferred.
4. It imposes specific service accounts and imposes the local System account as the sysadmin server role.
5. It does not support removal of SQL Server.

Design of the Resources

Reveal the Intention

Rather than provide the list of features as a parameter, we defined a number of separate intention-revealing resources instead. Therefore, there is one resource for the SQL Server Engine, another resource for SQL Analysis

Services, and another one for SQL Server Management Studio. When the time comes to add Reporting Services, we can define an additional resource.

This design means we do not need the features parameter any more. In some cases, we control sub features through additional parameters. For example, the SQL Database Engine has a Full Text option, which we install as a feature. The Full Text option is required when deploying TFS. Therefore, the Resource has a FullText parameter, indicating whether we need Full Text. Inside the Resource, we modify the list of features to install according to the value of the Full Text parameter. Similarly, there is a basic and full install of SQL Management Studio, and that is a feature controlled by an intention-revealing parameter on the Resource.

It should be clear that the principal that we have applied is to make the Resource parameters intentional, rather than just providing a technical list of features. Having a technical list of features requires the person authoring the configuration to know about how the Resource works, what the internal names of the features are etc.

Error Handling

The new SQL Server resources actually check whether the install succeeded or not. Most importantly, they do not set the reboot flag if the install failed. This will prevent an infinite reboot cycle, something that the Resource Kit resource is prone to do.

This raises the question of how to report errors. The answer is to throw an exception. This will tell the Local Configuration Manager that something went wrong; otherwise, it will just blindly carry on with the other configuration items we have given it, and assume your resource is in its desired state.

Give Important Choices

The Resource Kit resource imposes one, arguably less desirable, authentication mode, and forces us to use certain accounts as the service accounts and for the sysadmin server role. The new resources allow you to parameterize all these aspects to, so that the person defining the configuration is in control of these important aspects of the server.

We have defined other important parameters, allowing the locations of the database directories to be controlled. Most organizations have standards for where things are placed that must be respected, and so it is important to allow these aspects to be controlled.

Consider Supporting Ensure=Absent

This is arguably less useful. It would seem unlikely that anyone would use a SQL Server resource to ensure that you don't install SQL Server. However, it could be that an organization wants to move certain SQL Server components to other servers as their implementation grows.

Team Foundation Server Resources

There were no pre-existing Resources for Team Foundation Server, so we designed these from scratch. They assume a single server deployment.

Design of the Resources

Granularity

A Resource has been defined for the Application Tier and for Build, as these are the components that an Administrator would think in terms of. I also wanted a Team Project Collection Resource as well, but the only way to do this is through the TFS Object Model (see <http://blogs.msdn.com/b/granth/archive/2010/02/27/tfs2010-create-a-new-team-project-collection-from-powershell.aspx>) and there was not enough time to do this. This could be added later. It should be noted, though, that the Object Model does not appear to allow for the deletion of a Team Project Collection (which is probably a good thing!) which may make it difficult to create a fully-fledged PowerShell DSC Resource for Team Project Collections.

There was discussion within the team about whether there should be a Resource to represent just the Team Foundation Server binaries being installed on the machine. This is because you must install these binaries first since both the Application Tier and Build require them. However, we decided not to do this because it hides intention. It is not the intention to install the binaries; it is the intention to install an Application Tier. Furthermore, it is unlikely anyone would want to install the binaries and then do nothing with them, so installing an Application Tier (or Build) would always need two steps, and you would have to remember the dependency, there is more to get wrong. Consequently, we defined the Resources as much as possible in functional terms, rather than technical terms.

NOTE

Instead of the above approach, you should really consider to design principle of decomposing to smallest components. Use a composite resource to bundle the deployment with the configuration, see The DSC Book ⁴² for details.

This has a small impact on the [Get-TargetResource](#) function. It needs to detect if we have installed the binaries and if we have done the configuration. This is important for idempotency, as installation of a Resource could be interrupted for any reason. The hash table returned by [Get-TargetResource](#) returns the status of both items, so that [Test-TargetResource](#) can check if everything has been done, and [Set-TargetResource](#) knows which bits still need to be done.

Identity

One problem we encountered creating the TFS Resources was that the TFS Administrator must run the `tfscfg.exe` program. We normally run Resource scripts in the context of the Local System account, which cannot be the TFS Administrator.

To run the `tfscfg.exe` program under the identity of another account requires the use of PowerShell remote sessions. Specifically, the `-ComputerName` and `-Credential` options of the [Invoke-Command](#) cmdlet have been used to do this.

We need further steps if the `tfscfg.exe` program needs access to other systems in the domain. In that case, we also need the `-Authentication CredSSP` option, and we need to enable CredSSP. To enable CredSSP, enter the following commands on the server where we will use CredSSP:

```
winrm set winrm/config/client/auth @{CredSSP="true"}
winrm set winrm/config/service/auth @{CredSSP="true"}
```

It is also necessary to edit the Group Policy LocalComputerPolicy -> ComputerConfiguration -> Administrative Templates -> System -> Credentials Delegation -> Allow delegating fresh credentials. Enable this policy, and in the list of servers enter "WSMAN/<server FQDN>".

You can find more information about CredSSP here: [Multi-Hop Support in WinRM](#) ⁴³

⁴² aka.ms/dscPoBook

⁴³ [http://msdn.microsoft.com/en-us/library/ee309365\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ee309365(v=vs.85).aspx)

WARNING

Limitations

The Resources described here are not fully “make it so.” Once we have installed and configured the component we deem that successful. If the configuration is changed, the state will not be changed if the configuration is run again. In some cases, the component makes it impossible to check the configuration. For example, there is no way to find out what Team Project Collections there are, and we cannot rename the Team Project Collection if the configuration says it should be called something else.

Similarly, changes to the SQL Server configuration, although possible, could be quite hard to do. For example, if the directory locations are changed, there could be quite a lot of work to move databases.

We have not tested the Resources in a multiple server configuration, where, for example, SQL Server is on a different computer to TFS. Furthermore, we have not tested them in a domain with domain accounts; it is likely that some of the resource parameters will have to be changed to support this.

Building the Resources

In this section, we cover how we built the resources. The example used here is the resource used to install the SQL Server Engine.

Step	Instructions
1 Copy an existing resource ☐ - Done	<p>Rather than re-invent the wheel, the easiest thing to do is to make a copy of an existing resource and then change it into the module and resource you are designing. For the resources we are building here, we used the xSqlPs resource from the resource kit. We used the following steps:</p> <ul style="list-style-type: none"> • Install the Resource Kit. • In %ProgramFiles%\WindowsPowerShell\Module, copy the xSqlPs directory and create a new cSql directory. • Inside the %ProgramFiles%\WindowsPowerShell\Module\cSql directory rename the .psd1 file to cSql.psd1. • Edit the cSql.psd1 file; replace the GUID with a new GUID (use the Guidgen program, which is part of Visual Studio, to do this). Also, amend the Author, CompanyName, Copyright and Description settings as appropriate. • Inside the %ProgramFiles%\WindowsPowerShell\Module\cSql\DSCResources directory, delete all but one of the directories. Rename the directory you leave behind to match your resource name (in this case VSAR_cSqlServerEngine). • Rename the mof file to VSAR_cSqlServerEngine.schema.mof. • Rename the psm1 file to VSAR_cSqlServerEngine.psm1.
2 Update and test schema ☐ - Done	<p>The next step is to define the schema. We can do this with PowerShell cmdlets that generate properties. However, in this case we edit the MOF file directly.</p> <ul style="list-style-type: none"> • Open the MOF file in a text editor (notepad or Visual Studio will do). • Change the FriendlyName to cSqlServerEngine and the class name to VSAR_cSqlServerEngine, as shown below: <pre>[ClassVersion("1.0.0.0"), FriendlyName("cSqlServerEngine")] class VSAR_cSqlServerEngine : OMI_BaseResource</pre> <ul style="list-style-type: none"> • Edit the properties to match the requirements of your resource. Try to use ValueMap and Values attributes for properties that take one of a set of fixed values. Use EmbeddedInstance("MSFT_Credential") where a credential is required. The parameters for cSqlServerEngine were defined as follows: <pre>[Key, Description("The name of sql instance")] string InstanceName; [Write, ValueMap("Present", "Absent"), Values{"Present", "Absent"}] string Ensure; [Write, Description("The service account under which the SQL Agent runs")] string AgentServiceAccount; [Write, Description("The service account under which the SQL Server service runs")] string SqlServiceAccount; [Write, Description("The account which is in the sysadmin role for SQL Server")] string SysAdminAccount; [Write, Description("The location of the TempDB data files")] string TempDBDataDirectory;</pre>

Config as Code – Walkthrough - Deploy TFS 2013 using DSC

Step	Instructions
	<pre>[Write, Description("The location of the TempDB log files")] string TempDBLogDirectory; [Write, Description("The location of the user database data files")] string UserDBDataDirectory; [Write, Description("The location of the user database log files")] string UserDBLogDirectory; [Write, ValueMap{"Present", "Absent"}, Values{"Present", "Absent"}] string FullText; [Write, Description("The path to the directory where log files are to be placed")] string LogPath; [Write, Description("The share path of sql server software")] string SourcePath; [Write, EmbeddedInstance("MSFT_Credential"), Description("The credential to be used to access net share of sql server software")]string SourcePathCredential;</pre> <ul style="list-style-type: none"> Verify that the schema is valid using the following command: <pre>Test-xDscSchema -Path \$env:ProgramFiles\WindowsPowerShell\Modules\cSql\DSCResources\VSAR_cSqlServerEngine\VSAR_cSqlServerEngine.schema.mof</pre> Fix any issues reported. The cmdlet will return "true" if the schema is valid. Note that you have to be running an elevated PowerShell environment to run this cmdlet.
3 Edit parameters to TargetResource functions <input type="checkbox"/> - Done	<p>In this step you set up the Get/Test/Set_TargetResource functions with the parameters that match the schema MOF you edited in the previous step, as follows:</p> <ul style="list-style-type: none"> Open the .psm1 file. Edit the parameters of Get_TargetResource to match the Key properties in the MOF file. Edit the parameters of Test_TargetResource and Set_TargetResource file to match the full set of properties in the MOF file. Verify that the resource is valid using the following command: <pre>Test-xDscResource cSqlServerEngine</pre> Fix any issues reported. Be sure that the parameter attributes in the PowerShell functions match the properties in the MOF file. In particular: <ul style="list-style-type: none"> Use PScredential for EmbeddedInstance ("MSFT_Credential") properties. "Key" properties must be declared: [parameter(Mandatory)] Non-Key parameters that are nonetheless mandatory should be declared [ValidateNotNullOrEmpty ()] or [ValidateNotNull ()].
4 Implement the resource <input type="checkbox"/> - Done	<p>Add the code to the Get, Test and Set functions for the Resource as appropriate. These functions must operate idempotently, which means that it should be possible to run them more than once and have the same result without getting any errors.</p>
5 Add a new resource <input type="checkbox"/> - Done	<p>If further resources are required in the module, repeat the steps above, using the new resource that we have just created as the starting point. Only create an entirely new module if the next resource to be authored, is not related to other resources in the same module.</p>

Table 10 – Building the cTfs resource

NOTE

In this walkthrough, we only covered cSql. The same steps apply to the cTfs resource referenced in this walkthrough.

Steps to Configure TFS on a Single Server

This section describes the steps to take to build TFS on a single server, from scratch, using the PowerShell DSC resources that we previously described. The result is an installation of Team Foundation Server 2013 and a Build Service, along with the SQL Server software, all on a single server running Windows Server 2012 R2.

The pre-requisite is a server running Windows Server 2012 R2.

Step	Instructions
1 Gather the installation media <input type="checkbox"/> - Done	<p>Place the installers for the following products on a share that is accessible from the server where TFS is going to be installed:</p> <ul style="list-style-type: none"> Windows Server 2012 R2 SQL Server 2012 Team Foundation Server 2013
2 Install the DSC resource <input type="checkbox"/> - Done	<ul style="list-style-type: none"> Copy the cSql and the cTfs resources to the following directory on the target server: %ProgramFiles%\WindowsPowerShell\Module Make sure that the scripts are not blocked by running the following command: <pre>\$resourcePath = [System.IO.Path]::Combine((Get-Item env:ProgramFiles).Value, "WindowsPowershell\Modules\") Get-ChildItem -Path \$resourcePath -Recurse -include (*.psd1, "*.psm1") Unblock-File</pre>
3 Configure the server for DSC <input type="checkbox"/> - Done	<ul style="list-style-type: none"> To be able to run DSC and the PowerShell scripts, on the server, carry out the following steps: Run the following command from a command prompt (do not use a PowerShell shell as this may stop responding): winrm quickconfig Enable processing of unsigned scripts using the following PowerShell command, from a shell that is running with elevated permissions: Set-ExecutionPolicy remotesigned If the server is part of a domain and domain accounts are going to be used then enable CredSSP using the following command: winrm set winrm/config/client/auth @{CredSSP="true"}
4 Set up the configuration <input type="checkbox"/> - Done	<ul style="list-style-type: none"> Place the PowerShell configuration into a .ps1 file on the server: <pre>Configuration Tfs { ... }</pre> <p>See <i>Walkthrough – Deploy TFS 2013 using DSC configuration script</i>, on page 5269 for the complete script.</p> At the top of the file just created above, add the configuration data, which must be set up along the following lines, replacing the marked tokens: <pre>\$ConfigData= @{ ... }</pre> At the end of the file just created above, add the following script, which actually invokes the DSC resources.
5 Run the configuration <input type="checkbox"/> - Done	<ul style="list-style-type: none"> Run the script that we created in the previous step. This will update Windows, install SQL Server, and TFS. The machine will reboot a couple of times. Check the log file location if you want to check on progress.


Table 11 – Configure the resource

Appendix – PowerShell 101

This section aims at giving a taste of PowerShell™ so the reader can more easily grasp the following chapters and be able to look at the proper reference material when needed.

First impact

We think that the easiest way to understand Windows PowerShell™ is to start with some quick examples.

Let's open a Windows PowerShell command prompt: press the  key, start typing "power" and select Windows PowerShell. Now, you should see a window like shown below.

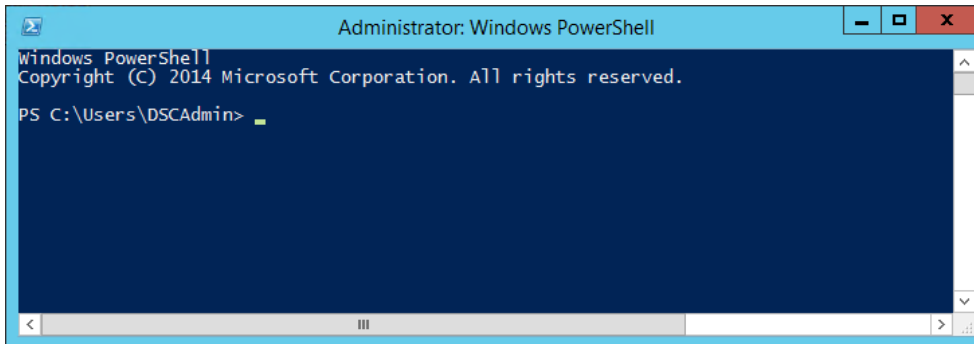


Figure 6 – Windows PowerShell™ interactive

Now you can type a command like `dir` (**bolded text** represents what you type).

```
PS C:\Users\Administrator> dir

Directory: C:\Users\Administrator

Mode                LastWriteTime         Length Name
----                -
d-r--             22-May-14   7:08 PM           Contacts
d-r--             03-Jun-14   7:32 PM           Desktop
...
```

While this output resembles the one from Command Prompt's native `dir` command, in reality it is quite different, because PowerShell commands⁴⁴ return objects, not strings.

Let demonstrate

```
PS C:\Users\Administrator> dir | sort LastWriteTime

Directory: C:\Users\Administrator

Mode                LastWriteTime         Length Name
----                -
-a---             21-Feb-14   6:02 PM          114 .gitconfig
d-r--             22-May-14   7:08 PM           Contacts
d-r--             22-May-14   7:08 PM           Favorites
...
```

The output of the `dir` command is piped into the `sort` command, which takes an argument that states, which object's property orders the result. Therefore, we discovered that commands may take arguments by position, which is not a big surprise; we can explicitly name the arguments, the name precedes the value and a dash (minus sign) marks it.

If you look carefully, the sort order is based on the property's *value*, in this case a date, not on the string representation. This is a critical difference from other environments, as you do not have to jump through hoops to parse strings and maybe the strings changes depending on locale.

⁴⁴ The term command is not quite right, as we will see soon. The exact word is *cmdlet*.

The use of the pipe sign is not original to PowerShell, as we use it in most command line shells: DOS, CMD, bash, csh, etc. It means to transfer the *objects* generated in output by the left command, to the next command on the right of the pipe. PowerShell is smart enough to optimize object generation through the pipeline: instead of waiting for the first command to complete and produce a bunch of objects; as soon as one object is ready, it is passed to the next command and so on⁴⁵.

A Language

Make no mistake, PowerShell is a full-blown programming language; see this example.

```
PS C:\Users\Administrator> $files = dir
PS C:\Users\Administrator> foreach ($f in $files) {
>> if ($f.LastWriteTime -gt ((Get-Date) - (New-TimeSpan -Days 7))) {
>> write $f.Name
>> }
>> }
>>
Desktop
Documents
Downloads
Videos
```

This code loops through the files in the current directory and outputs the name of files changed in the last seven days. Let us examine this code more carefully.

As you probably guessed, the dollar sign (\$) prefixes variables, so the first line stores the output from dir command into the variable "files."

The next line is more interesting, it states to loop through the content of the files variable, setting the f variable to the current item. Curly brackets ({}) groups commands⁴⁶ and PowerShell has the usual set of control statements: if/else for testing a condition, for and foreach to loop over a sequence of objects, while to conditionally loop, switch branch on a value, break, continue and return to control exit from deep nesting.

The loop body is an if statement with a complex condition

```
$f.LastWriteTime -gt (
(Get-Date) - (New-TimeSpan -Days 7)
)
```

It can be translated as pick current date, subtract seven days and compare with LastWriteProperty of the object stored in f variable, and return true if property value is more recent. In other words, gets the files changed in the last seven days.

The snippet has an interesting construct: -gt, in a place where a greater than symbol (>) is normally used. Surprising for a developer but not for an administrator: > and < are reserved to redirect output and input, respectively. That's the hardest habit to gain for programmers: stop typing angular brackets!

The minus (or dash) sign introduces a number of *operators*: -eq and -ne for equality and inequality, -gt, -ge, -lt, -le for comparison, -not, -and, -or, and -xor for logic (! is short for -not), -contains, -notcontains, -in, -notin for containment test, -like, -notlike and -match, -notmatch for string pattern matching. There are more operators, but this list a good starting point.

Get-Date and New-TimeSpan are two *cmdlets* that are two natural PowerShell commands. The first returns the current date and time, the latter a time interval⁴⁷.

The dot notation to access an object property should not be a surprise. Practically all modern languages use it.

You may use a different approach in your code, with identical result.

```
PS C:\Users\Administrator> dir | where {
>> $_.LastWriteTime -gt ((Get-Date) - (New-TimeSpan -Days 7))
>> } | select Name
>>
```

⁴⁵ Powershell commands employ a form of collaborative multi-tasking for this purpose; there is no real concurrency or Operating System threads involved.

⁴⁶ Not completely exact: curly braces build a *scriptblock*, that is packs a sequence of commands, to be run at a later time or in special way.

⁴⁷ Developers may suspect the presence of System.DateTime and System.TimeSpan classes and yes, they are right, beyond the scene, Powershell uses many primitive .Net framework classes.

```
Name
-----
Desktop
Documents
Downloads
Videos
```

This style leans heavily on pipelining commands, and we see this often in practice, so, our suggestion is to get used to reading both styles.

The automatic variable `$_` represent the current item in the loop, like `f` in the previous example. The `where` command is interesting: instead of testing the condition and the explicitly outputting an object, it filters which object passes the condition and move them downstream the pipeline. Last element is the `select` command, it projects some properties of the incoming objects in the output; you may say, it creates new objects on the fly based on the input. There is subtle difference, in fact, with the previous example: now the output is a series of objects with a single property `Name`; in the first case, we get a sequence of strings. You may easily spot the difference as the second result is formatted as a single-column table.

Exploring the field

We sloppily used the term command: the proper word is *cmdlet*. It is packed functionality written in some language (could be C#, VB.Net, or even PowerShell itself) that takes a number of parameters to realize some action. It is light in the sense that we run the cmdlets' code inside the PowerShell interpreter, without creating a new process.

Every new PowerShell version adds more cmdlets covering more Windows areas: PowerShell 4.0 in Windows 2012 R2 reaches more than 500 cmdlets. You can imagine the space it would take describing them all in details! A better approach is teaching you to fish, a proper metaphor, in the sea of cmdlets to find what you need, when you need it.

We organize Cmdlets in modules, each module covering a specific area; for example, the *WebAdministration* module permits to control most IIS objects like web sites or bindings.

To get information about modules use the `Get-Module` cmdlets. For example, a list of modules installed on the system:

```
PS C:\Users\Administrator> Get-Module -ListAvailable | select Name
```

The list of cmdlets contained in a module is easy to retrieve

```
PS C:\Users\Administrator> Get-Command -Module WebAdministration
```

The `Get-Command` is much more flexible; you can search names using wildcards

```
PS C:\Users\Administrator> Get-Command *apppool*
```

Notice the Verb-Object style of cmdlets: it does not happen by chance, instead is an important convention that helps finding and using commands. The range of verbs is controlled, large (around 100), but not arbitrary.

```
PS C:\Windows\system32> Get-Verb
```

This list goes on and on with seven contexts (groups): Common, Data, Lifecycle, Diagnostic, Communications, Security, and Other; this simply means that a verb like `Connect` will work in a communication context, e.g. `Connect-PSSession`⁴⁸, while a verb as `Add` can be found in any context, from `Add-AzureAccount` to `Add-WebConfigurationProperty`.

Recapping, you can navigate top-down and find a command that can be interesting. Now we need to discover how a command works and what data requires? The answer is easy: use the question mark option.

```
PS C:\Users\Administrator> dir -?
NAME
Get-ChildItem
```

⁴⁸ This command begins a Powershell session on a remote machine.

SYNTAX

```
Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [-Include <string[]>] [-Exclude <string[]>]
[-Recurse]
[-Force] [-Name] [-UseTransaction] [-Attributes <FlagsExpression[FileAttributes]> {ReadOnly |
Hidden | System |
Directory | Archive | Device | Normal | Temporary | SparseFile | ReparsePoint | Compressed |
Offline |
NotContentIndexed | Encrypted | IntegrityStream | NoScrubData}] [-Directory] [-File] [-Hidden] [-
ReadOnly]
[-System] [<CommonParameters>]

Get-ChildItem [[-Filter] <string>] -LiteralPath <string[]> [-Include <string[]>] [-Exclude
<string[]>] [-Recurse]
[-Force] [-Name] [-UseTransaction] [-Attributes <FlagsExpression[FileAttributes]> {ReadOnly |
Hidden | System |
Directory | Archive | Device | Normal | Temporary | SparseFile | ReparsePoint | Compressed |
Offline |
NotContentIndexed | Encrypted | IntegrityStream | NoScrubData}] [-Directory] [-File] [-Hidden] [-
ReadOnly]
[-System] [<CommonParameters>]
```

ALIASES

```
gci
ls
dir
```

REMARKS

Get-Help cannot find the Help files for this cmdlet on this computer. It is displaying only partial help.

-- To download and install Help files for the module that includes this cmdlet, use Update-Help.

-- To view the Help topic for this cmdlet online, type: "Get-Help Get-ChildItem -Online" or go to <http://go.microsoft.com/fwlink/?LinkID=113308>.

First, we discover that `dir` is not a primitive PowerShell cmdlet, but it is an alias for `Get-ChildItem`. This is a reinforcement of the verb-object rule. `dir`, `select`, where we disguise all cmdlets in a short, easy-to-type form. It is also a best practice to avoid using alias in source code, but writing the full cmdlet name.

The ending part is very interesting and shows how to get a more detailed help for the command.

The bulk of the output resolves in explaining which parameters the cmdlet accepts and the range of values admitted for each. You may have noticed that some arguments work as a switch, i.e. do not get any additional value but their sole presence alter the cmdlet behavior, `-Recurse` is an example.

PowerShell comes with a bare minimum help information you should refresh with latest additions and edits through this command.

```
PS C:\Users\Administrator> Update-Help
```

After this, we have more content for `dir/Get-ChildItem`.

```
PS C:\Users\Administrator> Get-Help dir
```

With this latest we have complete information to understand what the cmdlet does and the meaning of the options, but what about the output, the results flowing from it?

PowerShell is a **reflective** environment, which means you have PowerShell commands to get information on any object, a capability similar but more powerful than .Net reflection. Let us see this in action on the output of `dir/Get-ChildItem`.

```
PS C:\Users\Administrator> dir | Get-Member
```

Now, we know that `Get-ChildItem` returns objects of type `System.IO.DirectoryInfo` and of `System.IO.FileInfo` type, both well known to .Net developers. This reinforces what we said some pages above: PowerShell works with objects not basic strings.

`Get-Member` exposes all native properties and methods plus some. Yes, PowerShell adds supplemental information to ease object manipulation in scripts. They are easy to spot in the above listing: `NoteProperties` are

additional read-only data, ScriptProperties are dynamic properties that we write in PowerShell, and CodeProperties are dynamic code that we write in .Net, similar to C# extension methods. You can explore this and additional categories in the on-line documentation.

ISE

Suppose you are tired of all this typing, so it is time to learn a bit about PowerShell ISE (Integrated Scripting Environment, the tool that helps you writing and debugging code.

The most prominent feature for a newbie is the **Command Add-on** depicted in Figure 7. From this panel you can navigate the modules installed on the system, find cmdlets, understand their options, and so on

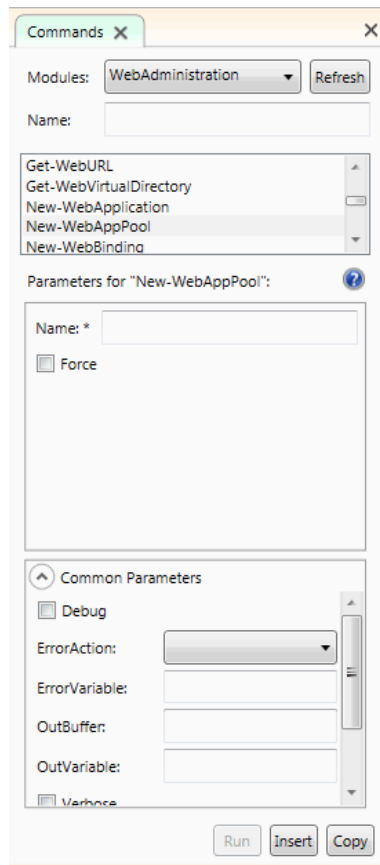


Figure 7 – Command panel

The three buttons at the bottom allows you to compose a command, run, or copy it. The command panel appears only when we enable the “Show Command Add-on” option, as in Figure 8.

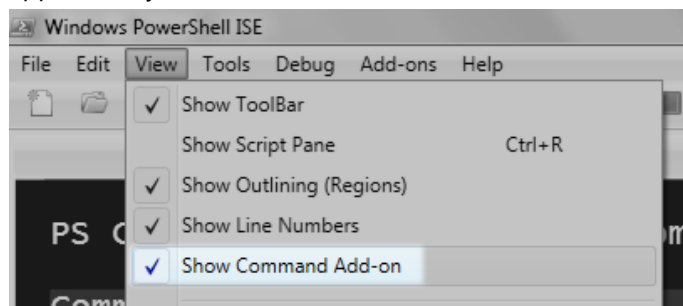


Figure 8 – Show Command Add-on Option

Packing up

So far, we worked interactively at the command line, now it is time to say something about scripting. You may avoid typing every time the same PowerShell code by saving it in a text file with the `.ps1` extension.

Take the following code:

```
Get-ChildItem "$env:SystemDrive\" -File -Recurse -ErrorAction SilentlyContinue | where { $_.Length -ge 100MB }
```

It scans the system drive to find any file bigger than 100 MB⁴⁹. Note the use of `$env:SystemDrive` which resolves in the `%SystemDrive%` environment variable, usually the C: drive.

Scripts

Save this code in a file named `findBigFiles.ps1` in your home directory, e.g. `C:\Users\Administrator`. To run it you can:

- Select "Run with PowerShell" from Windows Explorer's context menu (Figure 9)
- Type the path, relative or full, to the script from PowerShell interpreter
- Explicitly invoke the PowerShell interpreter with the `-File` argument and the full path to the script, from the Run prompt or a Command Line Prompt, e.g.
PowerShell `-File C:\Users\Administrator\findBigFiles.ps1`

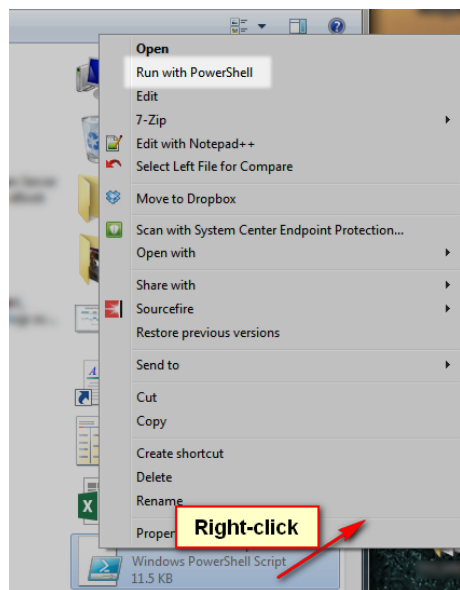


Figure 9 – Run a PowerShell script from Explorer

A script like this is useful to a point, as the disk and path to search are fixed, so let us make it more generic: the changed parts are in bold.

```
param (
    $pathToSearch = "$env:SystemDrive\",
    $sizeThreshold = 100MB
)
Get-ChildItem $pathToSearch -File -Recurse -ErrorAction SilentlyContinue | where { $_.Length -ge $sizeThreshold }
```

The `param` keyword introduces a set of parameters for the script, in this case the path where the search starts and the size of file we are interested in. Both parameters have a default value, used if the user does not specify a value.

To pass non-default values on a script invocation, just list them in the correct order separated with whitespace.

```
C:\Users\Administrator\findBigFiles.ps1 "C:\Program Files" 1MB
```

⁴⁹ Yes, Powershell is so kind to translate 100MB to 104857600.

You may also explicit the parameters names.

```
C:\Users\Administrator\findBigFiles.ps1 -sizeThreshold 1MB -pathToSearch C:\Builds
```

Surely, you have noticed that the syntax is homogeneous to invoking a cmdlet.

Functions

The basic packing mechanism to reuse code is *functions*. The code from our script becomes

```
function Search-BigFiles
{
    param(
        $pathToSearch = "$env:SystemDrive\",
        $sizeThreshold = 100MB
    )
    Get-ChildItem $pathToSearch -File -Recurse -ErrorAction SilentlyContinue |
        where { $_.Length -ge $sizeThreshold }
}
```

Invoking a function is quite similar to invoking a script.

```
Search-BigFiles C:\ProgramData 5MB
```

Functions permit you to decompose and organize your code; you can choose the names freely (no Verb-Object constraint).

Notable things

There are so many things to add, but we do not have space to cover them here. Two concepts are extremely important to automate distributed systems: PowerShell drives and Remoting, therefore we will briefly introduce them.

Drives

We are all used to DOS drives⁵⁰, C:, D:, and so on. PowerShell extends the concept to everything that we can represent as a hierarchy of containers and objects.

```
PS C:\Users\Administrator> Get-PSDrive
```

You have many real or virtual drives: disks, registry, and configuration. We manage each namespace by the respective provider and can be navigated and manipulated with the same cmdlets: Set-Location, Get-Item, Get-ChildItem, New-Item, Remove-Item, and so on.

The most used, apart from file system, are the registry and the certificate stores. A couple of example will help.

```
Get-ItemProperty -Path "HKCU:\Software\Microsoft\VisualStudio\12.0\TeamFoundation\SourceControl\Proxy"
-Name "URL"
```

Reads the Team Foundation Server proxy URL configured for the user.

```
Get-ChildItem Cert:\LocalMachine\My | where { $_.Issuer -like "*localhost*" }
```

Shows self-issued certificates.

Note: remember that most part of the Registry and Certificate Store requires administrative privileges even for reading.

Remoting and Serialization

PowerShell has powerful (pun not intended) built-in serialization mechanism. You can save on a file or read from it practically any object present in the PowerShell interpreter memory, script blocks included.

We use the same mechanism to send a command and its arguments on a remote machine so we run using a PowerShell instance on the remote machine with the result coming back to the caller.

⁵⁰ Few know that in Windows they are just a synonym to a specific path in the global OS namespace.

Checklist: Good PowerShell scripting practices

Category	Description	☒
Comments	<ul style="list-style-type: none"> Each script and function is self-documenting and well formatted. Minimum commenting keywords each function must include: .SYNOPSIS .DESCRIPTION .PARAMETER .EXAMPLE We document Dependencies, i.e. administrative access or potential dangers of script. We recommend Inline and closing bracket comments for clarity. Do not re-document self-documenting commands and parameters. The following is bad: # write a debug message Write-Debug "the command is self-explanatory and inline comment noisy" See about_Comment-Based_Help ⁵¹ for details or the base template on page 46. 	<input type="checkbox"/>
Output	<ul style="list-style-type: none"> Avoid use of Write-Host, which is visual UX metaphor bound, unless developing a Show* function. Instead use intelligent output <ul style="list-style-type: none"> Write-Verbose ... warm and fuzzy output Write_Error, Write-Warning, Write-Debug Use [CmdletBinding()] and refer to about_Functions_CmdletBindingAttribute ⁵² for details. 	<input type="checkbox"/>
Performance	<ul style="list-style-type: none"> Always test for performance and assume nothing. <ul style="list-style-type: none"> Piping tends to be slower, but uses less memory. Get-Process Select-Object -Property Name,@{n='Total';e={\$_.vm + \$_.m}} Structured tends to be faster, but uses more memory. \$procs = Get-Process foreach (\$proc in \$procs) { \$total = \$proc.vm + \$proc.pm Write-Output "\$(\$proc.name) \$(\$total)" } #foreach Use measure-command {} to measure performance of logic 	<input type="checkbox"/>
Formatted for readability	<ul style="list-style-type: none"> Indent code Avoid back ticks ` (escape character) as an effective line continuation <ul style="list-style-type: none"> Stick to one line or use "splatting" ... package parameters in hash tables to ensure a simple and one-line function, i.e. \$params = @{ 'ClassName' = 'Win32_LogicalDisk'; 'ComputerName' = 'localhost'; 'Filter' = 'DriveType=3' } Get-CimInstance @params See about_Splatting ⁵³ for details. 	<input type="checkbox"/>
Error Handling	<ul style="list-style-type: none"> Catch exceptions! try { # DO STUFF } #try catch { Write-Verbose -Message \$_ 	<input type="checkbox"/>

⁵¹ <http://technet.microsoft.com/en-us/library/hh847834.aspx>

⁵² <http://technet.microsoft.com/en-us/library/hh847872.aspx>

⁵³ <http://technet.microsoft.com/en-us/library/jj672955.aspx>

Category	Description	
	<pre> throw "..."</pre> <pre> } #catch</pre> <ul style="list-style-type: none"> Always test for positive ☺ and negative ☹ outcomes. 	
Tools vs Controller	<ul style="list-style-type: none"> Design and develop tools and controllers. Tools (hammer) <ul style="list-style-type: none"> Provide input for other tools ... Get/Input/ConvertFrom Provide processed output ... Out/Export/ConvertTo Processes one thing and outputs raw data Make modular and reusable Controllers (contractor) <ul style="list-style-type: none"> Use one or more tools to create an encapsulating single purpose Suitable to output formatted data 	<input type="checkbox"/>
Stay pure!	<ul style="list-style-type: none"> Use PowerShell commands (cmdlet) where possible. New-PSDrive -PSProvider FileSystem -Name Z -Root \\server\folder\path Devolve, i.e. use .NET objects, only when PowerShell does not provide feature. net use z: \\server\folder\path Keep it simple and maintainable. Document hacky code where needed. 	<input type="checkbox"/>

Table 12 – PowerShell scripting best practices checklist

Sample Template

Code 5 – Base template script snippet

```
<#####
# @Resource Overview@
#####>

function Get-Foo
<#####
.SYNOPSIS
    * A brief description of the function or script.

.DESCRIPTION
    * A detailed description of the function or script.

.PARAMETER <Parameter-Name>
    * The description of a parameter.

.EXAMPLE
    * A sample command that uses the function or script, optionally followed
    by sample output and a description. Repeat this keyword for each example.

.INPUTS
    The Microsoft .NET Framework types of objects that can be piped to the
    function or script.

.OUTPUTS
    The .NET Framework type of the objects that the cmdlet returns.

.NOTES
    Additional information about the function or script.

.LINK
    The name of a related topic. The URI must begin with "http" or "https".

.COMPONENT
    The technology or feature that the function or script uses, or to which
    it is related.
```

```
.ROLE
    The user role for the help topic.
#####>
function Foo
{
    param
    (
        [Parameter(Mandatory)]
        [ValidateNotNullOrEmpty()]
        [ValidateSet("X", "Y", "Z")]
        [String]$XYZ
    ) #param

    # do stuff
    measure-command {
        #DO STUFF HERE <<<<
    } #measure-command

    $returnValue = @{
        XYZ = [System.String]::Join("A", "B", "C", $XYZ)
    } #returnValue

    $returnValue
} #function Foo

# FUNCTIONS TO BE EXPORTED
Export-ModuleMember -function Foo
```

References and tooling ... where to find which gems

Videos

- TechEd 2014, Windows PowerShell Best Practices and Patterns: Time to Get Serious ⁵⁴
- TechEd 2012, Windows PowerShell Crash Course ⁵⁵
- Script Center: Windows PowerShell Scripting ⁵⁶

Podcasts

- PowerShell .org's PowerScripting podcast ⁵⁷

eBooks | Books

- PowerShell .org free eBooks ⁵⁸
- Windows PowerShell in Action, Second Edition ⁵⁹
- PowerShell in Depth: An administrator's guide ⁶⁰

Blogs

- The PowerShell Team Blog ⁶¹
- Hey, Scripting Guy! Blog ⁶²
- PowerShell.org Blog ⁶³
- PowerShell Magazine Blog ⁶⁴

Tutorials

- Using the PowerShell ISE ⁶⁵
- Microsoft Script Center ⁶⁶
- Scripting with PowerShell ⁶⁷
- Getting Started with PowerShell ⁶⁸

⁵⁴ <http://channel9.msdn.com/Events/TechEd/NorthAmerica/2014/DCIM-B418>

⁵⁵ <http://channel9.msdn.com/Events/TechEd/NorthAmerica/2012/WSV321-R>

⁵⁶ <http://technet.microsoft.com/en-us/scriptcenter/dd742419>

⁵⁷ <http://powershell.org/wp/powerscripting-podcast/>

⁵⁸ <http://1drv.ms/1eaLKiu>

⁵⁹ http://www.amazon.com/Windows-PowerShell-Action-Second-Edition/dp/1935182137/ref=sr_1_6?ie=UTF8&qid=1404739736&sr=8-6&keywords=powershell

⁶⁰ http://www.amazon.com/PowerShell-Depth-An-administrators-guide/dp/1617290556/ref=sr_1_3?ie=UTF8&qid=1404739736&sr=8-3&keywords=powershell

⁶¹ <http://blogs.msdn.com/b/powershell/>

⁶² <http://blogs.technet.com/b/heyscriptingguy/>

⁶³ <http://powershell.org/wp/>

⁶⁴ <http://www.powershellmagazine.com>

⁶⁵ <http://technet.microsoft.com/en-us/library/dd819474.aspx>

⁶⁶ <http://technet.microsoft.com/en-us/scriptcenter/bb410849.aspx>

⁶⁷ <http://technet.microsoft.com/library/bb978526.aspx>

⁶⁸ <http://technet.microsoft.com/en-us/library/hh857337.aspx>

Code Samples

Walkthrough - File Server & Share Custom Resource

Generate_cFileShare_Schema.ps1 script

Code 6 – Generate_cFileShare_Schema.ps1 script

#This creates the definition for the resource

#The share to create

```
$ShareName = New-xDscResourceProperty -Name ShareName -Type String -Attribute Key
```

#The path where the folder that maps to the share should be created

```
$Path = New-xDscResourceProperty -Name Path -Type String -Attribute Required
```

#Identify if the action is to create the share or remove the share (i.e. should it be present or absent)

```
$Ensure = New-xDscResourceProperty -Name Ensure -Type String -Attribute Write -ValidateSet "Present", "Absent"
```

#An array of users who should have full access

```
$FullAccessUsers = New-xDscResourceProperty -Name FullAccessUsers -Type String[] -Attribute Write
```

#An array of users who should have change access

```
$ChangeAccessUsers = New-xDscResourceProperty -Name ChangeAccessUsers -Type String[] -Attribute Write
```

#An array of users who should have read access

```
$ReadAccessUsers = New-xDscResourceProperty -Name ReadAccessUsers -Type String[] -Attribute Write
```

#Create the actual resource

```
New-xDscResource -Name VSAR_cCreateFileShare -Property $ShareName, $Path, $Ensure -Path 'C:\Program Files\WindowsPowerShell\Modules\cFileShare' -FriendlyName cCreateFileShare  
New-xDscResource -Name VSAR_cSetSharePermissions -Property $ShareName, $Ensure, $FullAccessUsers, $ChangeAccessUsers, $ReadAccessUsers -Path 'C:\Program Files\WindowsPowerShell\Modules\cFileShare' -FriendlyName cSetSharePermissions
```

VSAR_cCreateFileShare Get-TargetResource method

Code 7 – VSAR_cCreateFileShare Get-TargetResource method

```
function Get-TargetResource
{
    [CmdletBinding()]
    [OutputType([System.Collections.Hashtable])]
    param
    (
        [parameter(Mandatory = $true)]
        [System.String]
        $ShareName,
        [parameter(Mandatory = $true)]
        [System.String]
        $Path
    )

    #Temporary variable to store the results of the file share check
    $shareInfo = $null

    Write-Verbose -Message "Checking to see if the file share $ShareName exists..."

    #If the share exists, it will return an object
    $shareInfo = Get-SmbShare $ShareName -ErrorAction SilentlyContinue
    if ($shareInfo)
    {
        Write-Verbose -Message "File share exists."
        #If the Get-SmbShare does not throw an exception, then the share exists
        $ensureResult = "Present"
    }
    else
    {
        Write-Verbose -Message "File share does not exist."
        #The share doesn't exist
        $ensureResult = "Absent"
    }

    Write-Verbose -Message "Creating hashtable."

    $returnValue = @{
        ShareName = $ShareName
        Path = $Path
        Ensure = $ensureResult
    }

    $returnValue
}
```

VSAR_cCreateFileShare Test-TargetResource method

Code 8 – VSAR_cCreateFileShare Test-TargetResource

```

function Test-TargetResource
{
    [CmdletBinding()]
    [OutputType([System.Boolean])]
    param
    (
        [parameter(Mandatory = $true)]
        [System.String]
        $ShareName,

        [parameter(Mandatory = $true)]
        [System.String]
        $Path,

        [ValidateSet("Present", "Absent")]
        [System.String]
        $Ensure
    )

    Write-Verbose -Message "Configuration has requested that the share be $Ensure"

    #Initialize the return value variable
    $returnValue = $false

    if($Ensure -eq "Present")
    {
        <#
            The user wants to make sure that the share exists
            If it does, return true, if not false
        #>
        Write-Verbose -Message "Checking to see if the file share $ShareName exists..."
        $ShareInfo = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
        if ($ShareInfo)
        {
            Write-Verbose -Message "Share exists."
            $returnValue = $true
        }
        else
        {
            Write-Verbose -Message "Share does not exist."
            $returnValue = $false
        }
    }
    else
    {
        <#
            The user wants to make sure that the share does NOT exist
            If it does, return false, if not true
        #>
        Write-Verbose -Message "Checking to see if the file share $ShareName exists..."
        $ShareInfo = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
        if ($ShareInfo)
        {
            Write-Verbose -Message "Share exists."
            $returnValue = $false
        }
        else
        {
            Write-Verbose -Message "Share does not exist."
            $returnValue = $true
        }
    }

    $returnValue
}

```

VSAR_cCreateFileShare Set-TargetResource method

Code 9 – VSAR_cCreateFileShare Set-TargetResource

```
function Set-TargetResource
{
    [CmdletBinding()]
    param
    (
        [parameter(Mandatory = $true)]
        [System.String]
        $ShareName,

        [parameter(Mandatory = $true)]
        [System.String]
        $Path,

        [ValidateSet("Present", "Absent")]
        [System.String]
        $Ensure
    )

    if($Ensure -eq "Present")
    {
        #Add the share
        Write-Verbose -Message "Creating the file share"

        Write-Verbose -Message "Checking to see if the path exists"
        #Check to see if the path exists
        $PathExists = Test-Path -Path $Path

        if (!$PathExists)
        {
            Write-Verbose "The path does not exist"
            Write-Verbose "Creating the path..."
            #Create the folder because it doesn't exist
            New-Item -Path $Path -ItemType directory
        }

        Write-Verbose -Message "Creating the share"
        #Create the share
        New-SmbShare -Name $ShareName -Path $Path
    }
    else
    {
        Write-Verbose -Message "Removing the file share"
        #Remove the share
        Remove-SmbShare -Name $ShareName -Force
    }
}
```


VSAR_cSetSharePermission Get-TargetResource method

Code 10 – VSAR_cSetSharePermission Get-TargetResource method

```
function Get-TargetResource
{
    [CmdletBinding()]
    [OutputType([System.Collections.Hashtable])]
    param
    (
        [parameter(Mandatory = $true)]
        [System.String]
        $ShareName
    )

    #Write-Verbose "Use this cmdlet to deliver information about command processing."

    #Write-Debug "Use this cmdlet to write debug information while troubleshooting."

    #For this situation, this method will always return Ensure = false because we aren't
    #going to check the permissions every time in this method.

    $returnValue = @{
        ShareName = $ShareName
        Ensure = "Absent"
    }

    $returnValue
}
```

VSAR_cSetSharePermission Test-TargetResource method

Code 11 – VSAR_cSetSharePermission Test-TargetResource method

```
function Test-TargetResource
{
    [CmdletBinding()]
    [OutputType([System.Boolean])]
    param
    (
        [parameter(Mandatory = $true)]
        [System.String]
        $ShareName,

        [ValidateSet("Present", "Absent")]
        [System.String]
        $Ensure,

        [System.String[]]
        $FullAccessUsers,

        [System.String[]]
        $ChangeAccessUsers,

        [System.String[]]
        $ReadAccessUsers
    )

    <#
    If the users have the minimum permissions required by this resource, then return true.
    Read access users can be in Read, Change or full groups
    Change access users can be in Change or full groups
    Full access users must be in the Full group
    #>

    Write-Verbose -Message "Retrieving share permissions"

    #Get the members who have access to the share
    $results = Get-SmbShareAccess -Name $ShareName

    if ($Ensure -eq "Present")
    {
        Write-Verbose -Message "Checking for users to add"

        #before starting these checks, check to see if the user is attempting to add any users to the
        #share at all, if not this is not a required check
        if ($FullAccessUsers -ne $null)
        {
            Write-Verbose -Message "Checking full access users"

            #Loop through the list of full access users to be added
            for($i = 0; $i -lt $FullAccessUsers.Count; $i++)
            {
                #Search the list of returned users where the account name has been provided and the
                #current access right is full
                $found = $results | Where-Object { ($_.AccountName -eq $FullAccessUsers[$i])
                    -and ($_.AccessRight -eq "Full")}

                #If any user in this loop is not found return false to indicate that the state is not
                # as desired
                if ($found -eq $null)
                {
                    Write-Verbose -Message "At least one user was not found in the full access group"
                    return $false
                }
            }
        }

        if ($ChangeAccessUsers -ne $null)
        {
            Write-Verbose -Message "Checking change access users"

            for($i = 0; $i -lt $ChangeAccessUsers.Count; $i++)
            {
```

```
#For the change access user, check the change and full rights
$found = $results | Where-Object { ($_.AccountName -eq $ChangeAccessUsers[$i])
    -and (($_ .AccessRight -eq "Full") -or ($_ .AccessRight -eq "Change"))}
if ($found -eq $null)
{
    Write-Verbose -Message "At least one user was not found in the change access group"
    return $false
}
}

if ($ReadAccessUsers -ne $null)
{
    Write-Verbose -Message "Checking read access users"

    for($i = 0; $i -lt $ReadAccessUsers.Count; $i++)
    {
        #For the Read access users check the Full, Change and Read rights
        $found = $results | Where-Object { ($_.AccountName -eq $ReadAccessUsers[$i])
            -and (($_ .AccessRight -eq "Full")
                -or ($_ .AccessRight -eq "Change")
                -or ($_ .AccessRight -eq "Read"))}
        if ($found -eq $null)
        {
            Write-Verbose -Message "At least one user was not found in the read access group"
            return $false
        }
    }
}
}
else
{
    #The resource is to remove the users from the specified groups if they exist.
    #The removal is an exact remove whereas the add is a minimum set
    #before starting these checks, check to see if the user is attempting to add any users to the
    #share at all, if not this is not a required check
    Write-Verbose -Message "Checking for users to remove"

    if ($FullAccessUsers -ne $null)
    {
        Write-Verbose -Message "Checking full access users"

        #Loop through the list of full access users to be added
        for($i = 0; $i -lt $FullAccessUsers.Count; $i++)
        {
            #Search the list of returned users where the account name has been provided
            #and the current access right is full
            $found = $results | Where-Object { ($_.AccountName -eq $FullAccessUsers[$i])
                -and ($_ .AccessRight -eq "Full") }
            #If any user in this loop is found return false to indicate that the state is not as
            #desired
            if ($found -ne $null)
            {
                Write-Verbose -Message "At least one user was found in the full access group"
                return $false
            }
        }
    }

    if ($ChangeAccessUsers -ne $null)
    {
        Write-Verbose -Message "Checking change access users"

        for($i = 0; $i -lt $ChangeAccessUsers.Count; $i++)
        {
            $found = $results | Where-Object { ($_.AccountName -eq $ChangeAccessUsers[$i])
                -and ($_ .AccessRight -eq "Change") }
            if ($found -ne $null)
            {
                Write-Verbose -Message "At least one user was found in the change access group"
                return $false
            }
        }
    }
}
}
```

```
if ($ReadAccessUsers -ne $null)
{
    Write-Verbose -Message "Checking read access users"

    for($i = 0; $i -lt $ReadAccessUsers.Count; $i++)
    {
        $found = $results | Where-Object { ($_.AccountName -eq $ReadAccessUsers[$i]) -and
($_.AccessRight -eq "Read") }
        if ($found -ne $null)
        {
            Write-Verbose -Message "At least one user was found in the read access group"
            return $false
        }
    }
}

#If this is called, then all users are in an acceptable group (or not as the case may be)
return $true
}
```

VSAR_cSetSharePermission Set-TargetResource method

Code 12 – VSAR_cSetSharePermission Set-TargetResource method

```

function Set-TargetResource
{
    [CmdletBinding()]
    param
    (
        [parameter(Mandatory = $true)]
        [System.String]
        $ShareName,

        [ValidateSet("Present", "Absent")]
        [System.String]
        $Ensure,

        [System.String[]]
        $FullAccessUsers,

        [System.String[]]
        $ChangeAccessUsers,

        [System.String[]]
        $ReadAccessUsers
    )

    Write-Verbose -Message "Retrieving share permissions"

    #Get the members who have access to the share
    $results = Get-SmbShareAccess -Name $ShareName

    if($Ensure -eq 'Present')
    {
        if ($FullAccessUsers -ne $null)
        {
            #Loop through the list of full access users to be added
            for($i = 0; $i -lt $FullAccessUsers.Count; $i++)
            {
                #Search the list of returned users where the account name has been provided
                #and the current access right is full
                $found = $results | Where-Object { ($_.AccountName -eq $FullAccessUsers[$i]) -
                    -and ($_.AccessRight -eq "Full") }
                #If any user in this loop is not found add the user to the group
                if ($found -eq $null)
                {
                    Write-Verbose -Message "Adding user $FullAccessUsers[$i] to the Full access group"
                    Grant-SmbShareAccess -Name $ShareName -AccountName $FullAccessUsers[$i]
                        -AccessRight Full -Force
                }
            }
        }

        if ($ChangeAccessUsers -ne $null)
        {
            for($i = 0; $i -lt $ChangeAccessUsers.Count; $i++)
            {
                $found = $results | Where-Object { ($_.AccountName -eq $ChangeAccessUsers[$i]) -
                    and ($_.AccessRight -eq "Change") }
                if ($found -eq $null)
                {
                    Write-Verbose -Message "Adding user $ChangeAccessUsers[$i] to Change access group"
                    Grant-SmbShareAccess -Name $ShareName -AccountName $ChangeAccessUsers[$i]
                        -AccessRight Change -Force
                }
            }
        }

        if ($ReadAccessUsers -ne $null)
        {
            for($i = 0; $i -lt $ReadAccessUsers.Count; $i++)
            {

```

```

    $found = $results | Where-Object { ($_.AccountName -eq $ReadAccessUsers[$i])
                                     -and ($_.AccessRight -eq "Read") }

    if ($found -eq $null)
    {
        Write-Verbose -Message "Adding user $ReadAccessUsers[$i] to the Read access group"
        Grant-SmbShareAccess -Name $ShareName -AccountName $ReadAccessUsers[$i]
                               -AccessRight Read -Force
    }
}
}
else
{
    if ($FullAccessUsers -ne $null)
    {
        #Loop through the list of full access users to be added
        for($i = 0; $i -lt $FullAccessUsers.Count; $i++)
        {
            #Search the list of returned users where the account name has been provided
            #and the current access right is full
            $found = $results | Where-Object { ($_.AccountName -eq $FullAccessUsers[$i])
                                             -and ($_.AccessRight -eq "Full") }

            #If any user in this loop is not found add the user to the group
            if ($found -eq $null)
            {
                Write-Verbose -Message "Removing user $FullAccessUsers[$i] from Full access group"
                Remove-SmbShareAccess -Name $ShareName -AccountName $FullAccessUsers[$i]
                                       -AccessRight Full -Force
            }
        }
    }

    if ($ChangeAccessUsers -ne $null)
    {
        for($i = 0; $i -lt $ChangeAccessUsers.Count; $i++)
        {
            $found = $results | Where-Object { ($_.AccountName -eq $ChangeAccessUsers[$i])
                                             -and ($_.AccessRight -eq "Change") }

            if ($found -eq $null)
            {
                Write-Verbose -Message "Remoe user $ChangeAccessUsers[$i] from Change access grp"
                Remove-SmbShareAccess -Name $ShareName -AccountName $ChangeAccessUsers[$i]
                                       -AccessRight Change -Force
            }
        }
    }

    if ($ReadAccessUsers -ne $null)
    {
        for($i = 0; $i -lt $ReadAccessUsers.Count; $i++)
        {
            $found = $results | Where-Object { ($_.AccountName -eq $ReadAccessUsers[$i])
                                             -and ($_.AccessRight -eq "Read") }

            if ($found -eq $null)
            {
                Write-Verbose -Message "Removing user $ReadAccessUsers[$i] from Read access group"
                Remove-SmbShareAccess -Name $ShareName -AccountName $ReadAccessUsers[$i]
                                       -AccessRight Read -Force
            }
        }
    }
}
}
}

```

VSAR_cCreateFileShare_UnitTests.ps1

Code 13 – VSAR_cCreateFileShare_UnitTests.ps1

```
#Unit tests for VSAR_cCreateFileShare
Import-Module "C:\Program
Files\WindowsPowerShell\Modules\cFileShare\DSCResources\VSAR_cCreateFileShare"

#Variable Declarations
$ShareName = "TestShare"
$Path = "C:\Test"

$PassCounter = 0
$FailCounter = 0

#####
#
# Tests for Get-TargetResource
#
#####

#####
# Test #1 - If share exists, Ensure returns "Present"
#####

#Setup for Test #1
$SetupResult = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
if (!$SetupResult)
{
    New-SmbShare -Path $Path -Name $ShareName
}

$Result = Get-TargetResource -ShareName $ShareName -Path $Path

if ($Result.Ensure -ne "Present")
{
    $FailCounter += 1
    "Test 1 Failed"
}
else
{
    $PassCounter += 1
    "Test 1 Passed"
}

#####
# Test #2 - If share does not exist, Ensure returns "Absent"
#####

#Setup for Test #2
$SetupResult = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
if ($SetupResult)
{
    Remove-SmbShare -Name $ShareName -Force
}

$Result = Get-TargetResource -ShareName $ShareName -Path $Path

if ($Result.Ensure -eq "Present")
{
    $FailCounter += 1
    "Test 2 Failed"
}
else
{
    $PassCounter += 1
    "Test 2 Passed"
}

#####
#
# Tests for Test-TargetResource
#
```

```
#####

#####
# Test #3 - Configuration: Ensure Present
#       Share does exist
#       Returns True
#####

#Setup for Test #3
$SetupResult = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
if (!$SetupResult)
{
    $NewResult = New-SmbShare -Path $Path -Name $ShareName
}

$Result = Test-TargetResource -ShareName $ShareName -Path $Path -Ensure Present

if ($Result -eq $true)
{
    $PassCounter += 1
    "Test 3 Passed"
}
else
{
    $FailCounter += 1
    "Test 3 Failed"
}

#####
# Test #4 - Configuration: Ensure Present
#       Share does not exist
#       Returns False
#####

#Setup for Test #4
$SetupResult = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
if ($SetupResult)
{
    Remove-SmbShare -Name $ShareName -Force
}

$Result = Test-TargetResource -ShareName $ShareName -Path $Path -Ensure Present

if ($Result -eq $true)
{
    $FailCounter += 1
    "Test 4 Failed"
}
else
{
    $PassCounter += 1
    "Test 4 Passed"
}

#####
# Test #5 - Configuration: Ensure Absent
#       Share does not exist
#       Returns True
#####

#Setup for Test #5
$SetupResult = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
if ($SetupResult)
{
    Remove-SmbShare -Name $ShareName -Force
}

$Result = Test-TargetResource -ShareName $ShareName -Path $Path -Ensure Absent

if ($Result -eq $true)
{
    $PassCounter += 1
    "Test 5 Passed"
}
}
```



```

else
{
    $FailCounter += 1
    "Test 5 Failed"
}

#####
# Test #6 - Configuration: Ensure Absent
#     Share does exists
#     Returns False
#####

#Setup for Test #6
$SetupResult = Get-SmbShare -Name $ShareName -ErrorAction SilentlyContinue
if (!$SetupResult)
{
    $NewResult = New-SmbShare -Path $Path -Name $ShareName
}

$Result = Test-TargetResource -ShareName $ShareName -Path $Path -Ensure Absent

if ($Result -eq $true)
{
    $FailCounter += 1
    "Test 6 Failed"
}
else
{
    $PassCounter += 1
    "Test 6 Passed"
}

#####
#
# Tests for Set-TargetResource
#
#####

#####
# Test #7 - Configuration: Ensure Absent
#     Share does exists
#     Result: Share no longer exists
#####

#####
# Test #8 - Configuration: Ensure Absent
#     Share does not exists
#     Result: Share still does not exists
#####

#####
# Test #9 - Configuration: Ensure Present
#     Share does exists
#     Result: Share still exists
#####

#####
# Test #10 - Configuration: Ensure Present
#     Path exists but share does not exist
#     Result: Share exists
#####

#####
# Test #11 - Configuration: Ensure Present
#     Path does not exist and share does not exist
#     Result: Share exists
#####

#Add in the unit tests for the set share permissions

""
"Passed: $PassCounter, Failed: $FailCounter"

```

VSAR_cSetSharePermission_UnitTests.ps1

Code 14 - VSAR_cSetSharePermission_UnitTests.ps1

#Unit tests for VSAR_cSetSharePermissions

```
Import-Module "C:\Program
Files\WindowsPowerShell\Modules\cFileShare\DSCResources\VSAR_cSetSharePermissions"
```

#Variable Declarations

\$ShareName = "TestShare"

\$Users = @("[domain]\[user1]", "[domain]\[user2]")

\$Path = "C:\Test"

\$User1 = "[domain]\[user1]"

\$User2 = "[domain]\[user2]"

\$PassCounter = 0

\$FailCounter = 0

#####

#

Tests for Get-TargetResource

#

#####

#####

Test #1: Share does not exist

Ensure is Absent

#####

#Setup for Test #1

\$SetupResult = Get-SmbShare -Name \$ShareName -ErrorAction SilentlyContinue

if (\$SetupResult)

{

\$SetupResult = Remove-SmbShare -Name \$ShareName -Force

}

\$result = Get-TargetResource -ShareName "TestShare"

if (\$result.Ensure -eq "Present")

{

\$FailCounter += 1

"Test 1 Failed"

}

else

{

\$PassCounter += 1

"Test 1 Passed"

}

#####

#

#This setup is valid for the rest of the tests. Ensures the share exists before beginning these tests

#

#####

\$SetupResult = Get-SmbShare -Name \$ShareName -ErrorAction SilentlyContinue

if (!\$SetupResult)

{

\$SetupResult = New-SmbShare -Path \$Path -Name \$ShareName

}

#####

#

Tests for Test-TargetResource

#

#####

#####

Test #2: Ensure is Present

AccessLevel is Read

One user is not in the Read group

Result False

#####

#setup - add a single user to the Read group

```
RemoveUsers($ShareName, $User1, $User2);
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Read -AccountName $User1 -Force

#test
$TestResult = Test-TargetResource -ShareName $ShareName -ReadAccessUsers $Users -Ensure Present
if ($TestResult)
{
    $FailCounter += 1
    "Test 2 Failed"
}
else
{
    $PassCounter += 1
    "Test 2 Passed"
}

#####
# Test #3: Ensure is Present
#     AccessLevel is Read
#     One user is in the Full group
#     One user is in the Change group
#     Result True
#####

#setup
RemoveUsers($ShareName, $User1, $User2);
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Change -AccountName $User1 -Force
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Full -AccountName $User2 -Force

#test
$TestResult = Test-TargetResource -ShareName $ShareName -ReadAccessUsers $Users -Ensure Present
if (!$TestResult)
{
    $FailCounter += 1
    "Test 3 Failed"
}
else
{
    $PassCounter += 1
    "Test 3 Passed"
}

#####
# Test #4: Ensure is Present
#     AccessLevel is Change
#     One user is in the Full group
#     One user is in the Change group
#     Result True
#####

#setup
RemoveUsers($ShareName, $User1, $User2);
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Change -AccountName $User1 -Force
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Full -AccountName $User2 -Force

#test
$TestResult = Test-TargetResource -ShareName $ShareName -ChangeAccessUsers $Users -Ensure Present
if (!$TestResult)
{
    $FailCounter += 1
    "Test 4 Failed"
}
else
{
    $PassCounter += 1
    "Test 4 Passed"
}

#####
# Test #5: Ensure is Present
#     AccessLevel is Change
#     One user is in the Full group
#     One user is in the Read group
#     Result False
```

```
#####

#setup
RemoveUsers($ShareName, $User1, $User2);
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Read -AccountName $User1 -Force
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Full -AccountName $User2 -Force

#test
$TestResult = Test-TargetResource -ShareName $ShareName -ChangeAccessUsers $Users -Ensure Present
if ($TestResult)
{
    $FailCounter += 1
    "Test 5 Failed"
}
else
{
    $PassCounter += 1
    "Test 5 Passed"
}

#####
# Test #6: Ensure is Present
#     AccessLevel is Full
#     One user is in the Full group
#     One user is in the Change group
#     Result False
#####

#setup
RemoveUsers($ShareName, $User1, $User2);
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Change -AccountName $User1 -Force
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Full -AccountName $User2 -Force

#test
$TestResult = Test-TargetResource -ShareName $ShareName -FullAccessUsers $Users -Ensure Present
if ($TestResult)
{
    $FailCounter += 1
    "Test 6 Failed"
}
else
{
    $PassCounter += 1
    "Test 6 Passed"
}

#####
# Test #7: Ensure is Absent
#     AccessLevel is Read
#     One user is in the Full group
#     One user is in the Change group
#     Result True
#####

#setup
RemoveUsers($ShareName, $User1, $User2);
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Change -AccountName $User1 -Force
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Full -AccountName $User2 -Force

#test
$TestResult = Test-TargetResource -ShareName $ShareName -ReadAccessUsers $Users -Ensure Absent
if (!$TestResult)
{
    $FailCounter += 1
    "Test 8 Failed"
}
else
{
    $PassCounter += 1
    "Test 8 Passed"
}

#####
# Test #8: Ensure is Absent
```

```
#           AccessLevel is Change
#           One user is in the Full group
#           One user is in the Change group
#           Result False
#####

#setup
RemoveUsers($ShareName, $User1, $User2);
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Change -AccountName $User1 -Force
$SetupResult = Grant-SmbShareAccess -Name $ShareName -AccessRight Full -AccountName $User2 -Force

#test
$TestResult = Test-TargetResource -ShareName $ShareName -ChangeAccessUsers $Users -Ensure Absent
if ($TestResult)
{
    $FailCounter += 1
    "Test 9 Failed"
}
else
{
    $PassCounter += 1
    "Test 9 Passed"
}

""
"Passed: $PassCounter, Failed: $FailCounter"

function RemoveUsers($a)
{
    $revokeResult = Revoke-SmbShareAccess -Name $a[0] -AccountName $a[1] -force
    $revokeResult = Revoke-SmbShareAccess -Name $a[0] -AccountName $a[2] -force
}
```

cFileShare.psd1 Manifest File

Code 15 –cFileShare.psd1 Manifest File

```
#
# Module manifest for module 'cFileShare'
#
# Generated by: ALM Rangers
#
# Generated on: 8/14/2014
#

@{

# Script module or binary module file associated with this manifest.
# RootModule = ''

# Version number of this module.
ModuleVersion = '1.0'

# ID used to uniquely identify this module
GUID = '24e2b672-5c37-45d1-b292-2b45dcba8f19'

# Author of this module
Author = 'ALM Rangers'

# Company or vendor of this module
CompanyName = 'Microsoft Corporation'

# Copyright statement for this module
Copyright = '(c) 2014 ALM Rangers. All rights reserved.'

# Description of the functionality provided by this module
# Description = ''

# Minimum version of the Windows PowerShell engine required by this module
# PowerShellVersion = ''

# Name of the Windows PowerShell host required by this module
# PowerShellHostName = ''

# Minimum version of the Windows PowerShell host required by this module
# PowerShellHostVersion = ''

# Minimum version of Microsoft .NET Framework required by this module
# DotNetFrameworkVersion = ''

# Minimum version of the common language runtime (CLR) required by this module
# CLRVersion = ''

# Processor architecture (None, X86, Amd64) required by this module
# ProcessorArchitecture = ''

# Modules that must be imported into the global environment prior to importing this module
# RequiredModules = @()

# Assemblies that must be loaded prior to importing this module
# RequiredAssemblies = @()

# Script files (.ps1) that are run in the caller's environment prior to importing this module.
# ScriptsToProcess = @()

# Type files (.ps1xml) to be loaded when importing this module
# TypesToProcess = @()

# Format files (.ps1xml) to be loaded when importing this module
# FormatsToProcess = @()

# Modules to import as nested modules of the module specified in RootModule/ModuleToProcess
# NestedModules = @()

# Functions to export from this module
FunctionsToExport = '*'
}
```

```
# Cmdlets to export from this module
CmdletsToExport = '*'

# Variables to export from this module
VariablesToExport = '*'

# Aliases to export from this module
AliasesToExport = '*'

# List of all modules packaged with this module
# ModuleList = @()

# List of all files packaged with this module
# FileList = @()

# Private data to pass to the module specified in RootModule/ModuleToProcess
# PrivateData = ''

# HelpInfo URI of this module
# HelpInfoURI = ''

# Default prefix for commands exported from this module. Override the default prefix using Import-
# Module -Prefix.
# DefaultCommandPrefix = ''

}
```

VSAR_cCreateFileShare.schema.mof Schema File

Code 16 – VSAR_cCreateFileShare.schema.mof Schema File

```
[ClassVersion("1.0.0.0"), FriendlyName("cCreateFileShare")]
class VSAR_cCreateFileShare : OMI_BaseResource
{
    [Key] String ShareName;
    [Required] String Path;
    [Write, ValueMap{"Present","Absent"}, Values{"Present","Absent"}] String Ensure;
};
```

VSAR_cSetSharePermissions.schema.mof Schema File

Code 17 – VSAR_cSetSharePermissions.schema.mof Schema File

```
[ClassVersion("1.0.0.0"), FriendlyName("cSetSharePermissions")]
class VSAR_cSetSharePermissions : OMI_BaseResource
{
    [Key] String ShareName;
    [Write, ValueMap{"Present","Absent"}, Values{"Present","Absent"}] String Ensure;
    [Write] String FullAccessUsers[];
    [Write] String ChangeAccessUsers[];
    [Write] String ReadAccessUsers[];
};
```


Walkthrough – Deploy TFS 2013 using PowerShell DSC

Configuration script

Code 18 – Deploy TFS 2013 using DSC configuration script

```
$ConfigData=
@{
    AllNodes = @(
        @{
            NodeName="*"
            PSDscAllowPlainTextPassword=$true
        }
        @{
            NodeName = "<name of TFS server>"
        }
    )
}
```

Configuration Tfs

```
{
    param
    (
        [Parameter(Mandatory)]
        [string] $WindowsMediaPath,
        [Parameter(Mandatory)]
        [string] $SqlServerMediaPath,
        [Parameter(Mandatory)]
        [string] $TfsMediaPath,
        [Parameter(Mandatory)]
        [string] $LogPath,
        [Parameter(Mandatory=$true)]
        [ValidateNotNullorEmpty()]
        [PSCredential] $TfsAdministratorCredential
    )

    Import-DscResource -Module cSql
    Import-DscResource -Module cTfs

    Node <name of TFS server>
    {
        WindowsFeature InstallDotNet35
        {
            Ensure = "Present"
            Name = "Net-Framework-Core"
            Source = (Join-Path $WindowsMediaPath -ChildPath "\Sources\SxS")
        }

        WindowsFeature InstallDotNet40
        {
            Ensure = "Present"
            Name = "AS-NET-Framework"
        }

        cSqlServerEngine InstallSqlServer2012Engine
        {
            InstanceName = "MSSQLSERVER"
            Ensure = "Present"
            AgentServiceAccount = "NT Authority\Network Service"
            SqlServiceAccount = "NT Authority\Network Service"
            SysAdminAccount = $TfsAdministratorCredential.UserName
            TempDBDataDirectory = "C:\SQL\Databases\TempDB\"
            TempDBLogDirectory = "C:\SQL\Databases\TempDB\"
            UserDBDataDirectory = "C:\SQL\Databases\UserDBs\"
            UserDBLogDirectory = "C:\SQL\Databases\UserDBs\"
            FullText = "Present"
            LogPath = $LogPath
            SourcePath = $SqlServerMediaPath
            DependsOn = "[WindowsFeature]InstallDotNet35","[WindowsFeature]InstallDotNet40"
        }
    }
}
```

```

}

cSqlServerAnalysisServices InstallSqlServer2012AnalysisServices
{
    InstanceName = "MSSQLSERVER"
    Ensure = "Present"
    ServiceAccount = "NT Authority\Network Service"
    SysAdminAccount = $TfsAdministratorCredential.UserName
    TempDataDirectory = "C:\SQL\AnalysisServices\Temp\"
    LogPath = $LogPath
    SourcePath = $SqlServerMediaPath
    DependsOn = "[WindowsFeature]InstallDotNet35","[WindowsFeature]InstallDotNet40"
}

cSqlServerManagementStudio InstallSqlServer2012ManagementStudio
{
    Name = "SSMS"
    Ensure = "Present"
    InstanceDirectory = ""
    Advanced = "Present"
    LogPath = $LogPath
    SourcePath = $SqlServerMediaPath
    DependsOn = "[cSqlServerEngine]InstallSqlServer2012Engine",
                "[cSqlServerAnalysisServices]InstallSqlServer2012AnalysisServices"
}

cTfsApplicationTier InstallTfs2013
{
    Name = $Node.NodeName
    Ensure = "Present"
    TfsAdminCredential = $TfsAdministratorCredential
    TfsServiceAccount = "NT AUTHORITY\Local Service"
    SqlServerInstance = $Node.NodeName
    FileCacheDirectory = "C:\TFS\FileCache"
    TeamProjectCollectionName = "DefaultCollection"
    LogPath = $LogPath
    SourcePath = $TfsMediaPath
    DependsOn = "[cSqlServerEngine]InstallSqlServer2012Engine",
                "[cSqlServerAnalysisServices]InstallSqlServer2012AnalysisServices"
}

cTfsBuildServer InstallTfs2013BuildServer
{
    Name = $Node.NodeName
    Ensure = "Present"
    ConfigurationCredential = $TfsAdministratorCredential
    BuildServiceCredential = $TfsAdministratorCredential
    Port = 9191
    AgentCount = 2
    TeamProjectCollectionUri = "http://localhost:8080/tfs/DefaultCollection/"
    LogPath = $LogPath
    SourcePath = $TfsMediaPath
    DependsOn = "[cTfsApplicationTier]InstallTfs2013"
}

LocalConfigurationManager
{
    RebootNodeIfNeeded = $true
}

}

}

$MofPath = ".\Mof"
$LogPath = "c:\DSCLogsTFS"
$WindowsMediaPath = "D:\"
$SqlServerMediaPath = "E:\"
$TfsMediaPath = "F:\"
$domainName = "<domain/workgroup>"
$domainAdminAccount = New-Object System.Management.Automation.PSCredential("$domainName\Administrator",
(ConvertTo-SecureString "<password>" -AsPlainText -Force))

if (!(Test-Path $MofPath))
{
    New-Item $MofPath -ItemType Directory

```

```
}

if (!(Test-Path $LogPath))
{
    New-Item $LogPath -ItemType Directory
}

Tfs -ConfigurationData $ConfigData -OutputPath .\Mof -WindowsMediaPath $WindowsMediaPath -
SqlServerMediaPath $SqlServerMediaPath -TfsMediaPath $TfsMediaPath -LogPath $LogPath -
TfsAdministratorCredential $domainAdminAccount

Set-DscLocalConfigurationManager .\Mof

Start-DscConfiguration -Path .\Mof -ComputerName $env:COMPUTERNAME -Wait -Debug
```

In Conclusion

This concludes our adventure of **PowerShell Desired State Configuration** during which we briefly explored the essentials of PowerShell and DSC resources, practical walkthroughs of creating a custom resource and deploying TFS 2013, quick reference cheat sheets / posters, and checklists.

We hope you find it a valuable technology to invest in and that you have found this guide useful.

Sincerely

The Microsoft Visual Studio ALM Rangers




Quick Reference Sheets / Posters

PowerShell Desired State Configuration (DSC) Resource Overview

This cheat sheet is available separately in high-quality JPG and PDF format as part of the guidance.

Desired State Configuration (DSC) Resource Overview


Visual Studio
 ALM Rangers

1 Getting Ready

- DSC Resource Kit aka.ms/dscRK
- DSC Guidance aka.ms/dscGuide
- PowerShell.org location aka.ms/dscPso
- PowerShell.org eBook aka.ms/dscPsoBook

2 Architecture

- Custom module folder
- Manifest (.psd1)
- DSCResource sub-folder
- Resource sub folder
- Script module (.psm1)
- Schema (.mof)

Resource Designer creates the structure for you!

3 Dev process

- Planning the resource
 - Define self-config information
- Define properties
 - Use Resource Designer aka.ms/dscRD
- Program resource
 - Get-TargetResource
 - Set-TargetResource
 - Test-TargetResource
- Test & Deploy aka.ms/dscTP


DSC Resource Module

- Manifest.ps1
- DSCResource
 - Resource ... 1
 - Script.psm1
 - Schema.mof
 - Resource ... n

Example: xNetworking.psd1

Example: MSFT_xFirewall.psm1

Example: MSFT_xFirewall.Schema.mof



Visual Studio ALM Rangers Solutions – <http://aka.ms/vsarsolutions>

2014-09-26 v1

DSC – Custom Resource Walkthrough

This cheat sheet is available separately in high-quality JPG and PDF format as part of the guidance.

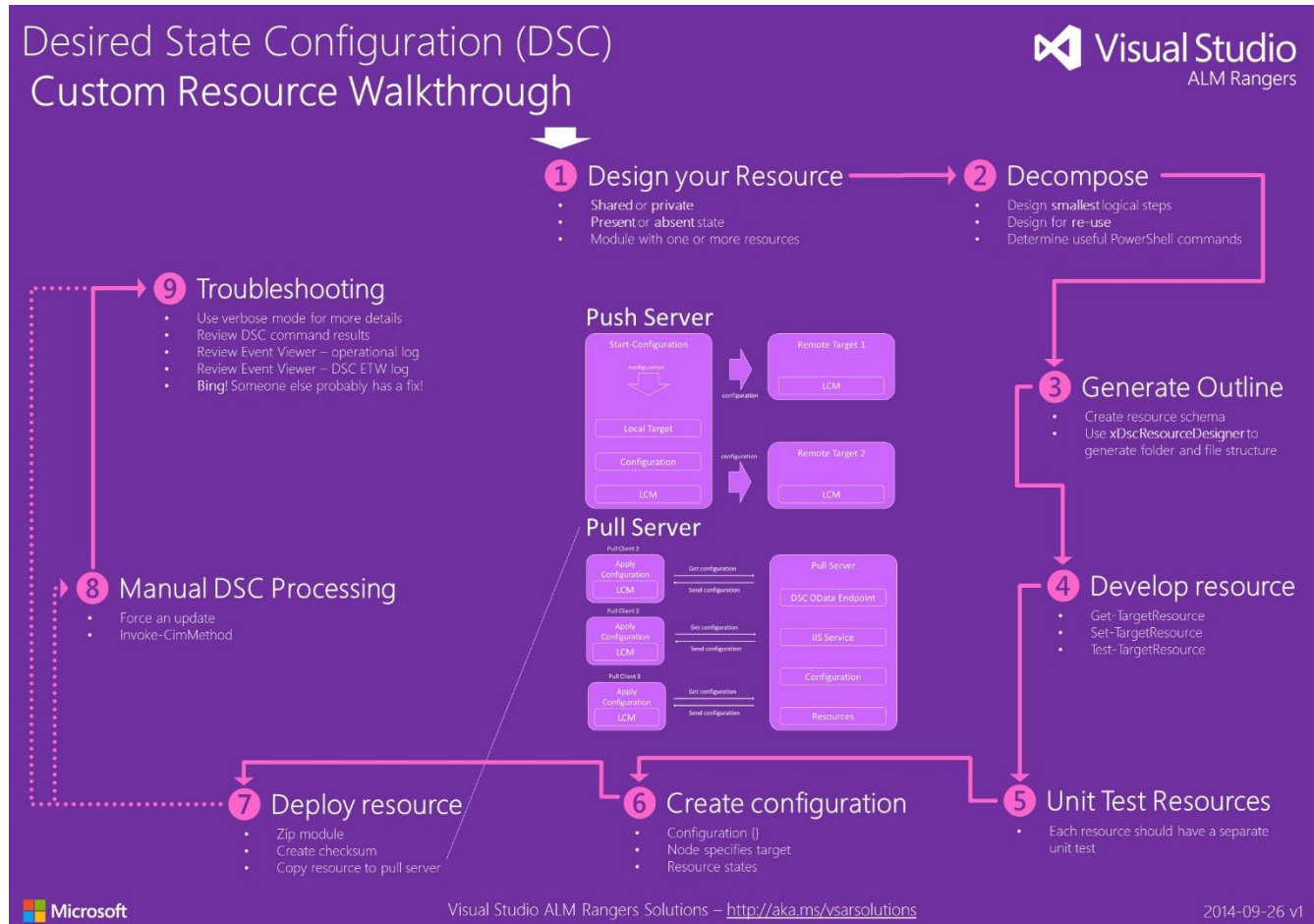


Figure 10 - DSC - Custom Resource Walkthrough

DSC – Implementation Walkthrough

This cheat sheet is available separately in high-quality JPG and PDF format as part of the guidance.

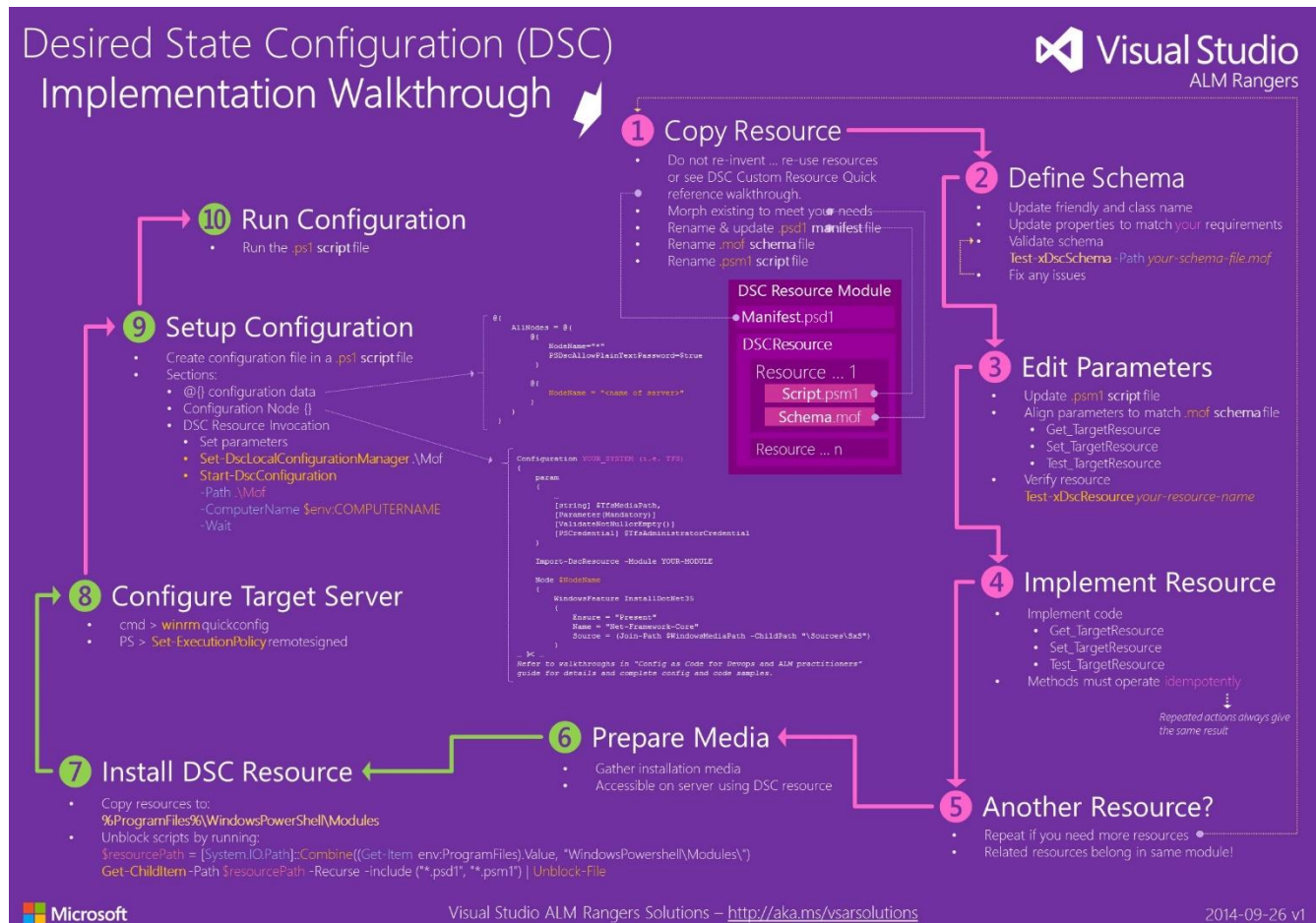


Figure 11 - DSC - Implementation Walkthrough

DSC – Composite Walkthrough

This cheat sheet is available separately in high-quality JPG and PDF format as part of the guidance.

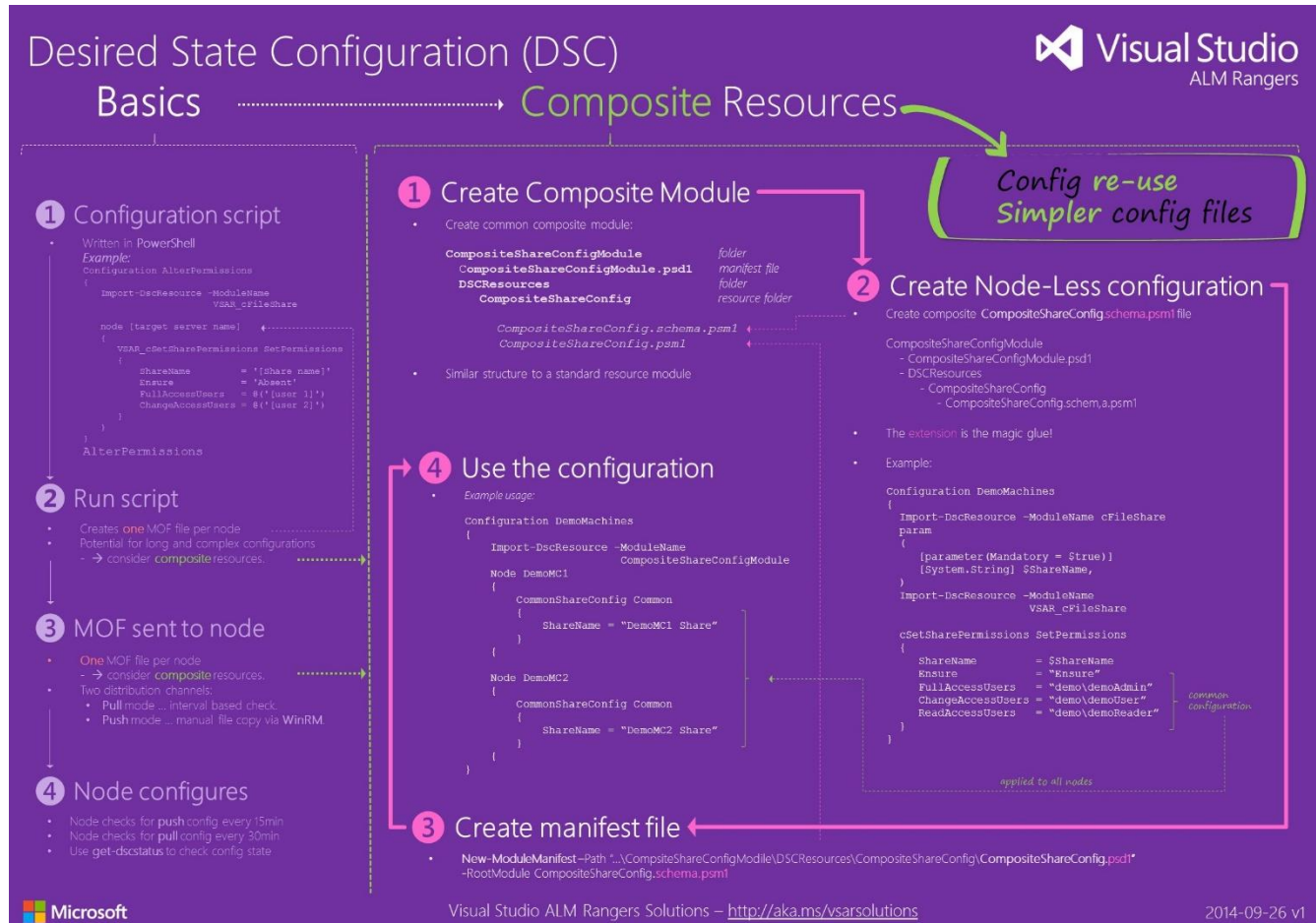


Figure 12 - DSC – Composite Walkthrough