# Getting started with Windows PowerShell Desired State Configuration

1. Introduction
2. Why DSC?
3. Key concepts
   1. Declarative, expression of end state, idempotent
4. Define (Language)
   1. Configuration keyword
   2. Node keyword
   3. Resources
   4. DependsOn
   5. Configuration Data
5. Compile
6. Deploy
   1. Push
   2. Pull
7. Maintain
   1. Logs
   2. Cmdlets (test, compare, etc.)
8. Try it {link to Lab}
9. Need More Resources
   1. Finding
      1. TechNet
      2. PowerShellGet
      3. Get- DscResource
   2. Developing
      1. Script
      2. Native
      3. Classes
   3. Import-DscResource
10. Components
11. More information {link to Don/Steve’s book: http://powershell.org/wp/ebooks/ }
12. Definitions

# Introduction

With the proliferation of the cloud, a new breed of configuration management tools has been created to manage the platforms, applications, and infrastructure. The need for these new tools and infrastructure comes from the increase in scale, rapid rate of change, and complexity of the cloud. In order for Windows Server to deliver on its promise as a cloud OS for the modern datacenter, it must include tools that enable and deployment and manageability at large scale.

Customers need to be able to deploy and manage the configuration of Windows Server, its applications, roles, features, and infrastructure for the cloud throughout the lifecycle in an easy and maintainable way. Windows PowerShell Desired State Configuration (DSC) provides a configuration platform that is based in open standards and is flexible enough to function reliably in each stage of the deployment lifecycle (development, test, pre-production, production) as well as during scale-out.

In order for DSC to make Windows Server truly cloud-ready, all components and applications must be fully supported. One or more DSC resources must be created for each Windows Server component and application. Since DSC is implemented with open standards, these resources can then be utilized by DSC or any other tool that needs to manage Windows Servers.

# Why DSC?

# Key Concepts

Windows PowerShell Desired State Configuration (DSC) is a declarative platform used for configuration, deployment, and management of systems. It consists of four primary components:

* **Resources** are the imperative building blocks of DSC which are written to model the various components of a sub-system and implement the control flow of their changing states. They reside within PowerShell modules and can be written to model something as generic as a file or a Windows process or as specific as an IIS server or a VM running in Azure.
* A number of additions to the PowerShell language, including a **Configuration** keyword (which can contain **Node** blocks and calls to the keyword **Import-DscResource**). A Configuration block is a special kind of declarative PowerShell function which constructs a desired state out of DSC resources.
* The **Local Configuration Manager** **(LCM)** is the engine by which DSC facilitates the interaction between resources and configurations. The LCM regularly polls the system using the control flow implemented by resources to ensure that the state laid out by a Configuration is maintained. If the system is out of state, the LCM uses more logic inside of the resources to “make it so” according to the Configuration declaration.
* A number of cmdlets and tools to help build DSC resources, invoke configurations, and manage the LCM. Many of these cmdlets can be found in Windows 8.1 as part of the **PsDesiredStateConfig** module (including **Start-DscConfiguration**, **Set-DscLocalConfigurationManager**, and **Get-DscResource**). The **xDscResourceDesigner** (found on TechNet)is a collection of cmdlets that simplify the development of DSC resources.

In other words, resources define the logic of *how* to configure something, and the configuration declares in *what* state instances of that thing should exist. This means that, once a resource is written properly, configuration writers do not have to deal with convoluted control flow in order to deploy incremental changes to their environments. With DSC, there is no risk of configuration drift putting systems into a bad state.

# Resources (should come after Configurations)

**Resources** can model something as generic as a file or a Windows process or as specific as an IIS server or a VM running in Azure. Resources are special PowerShell modules in that they must include the following functions:

* Get-TargetResource: retrieves the configurable unit modeled by the resource
* Test-TargetResource: determines whether the configurable unit modeled by the resource is in the correct state
* Set-TargetResource: makes changes to the configurable unit modeled by the resource when Test-TargetResource returns false

When a Configuration is executed

# Configurations

DSC **configurations** are declarative PowerShell scripts which define and configure instances of resources. Upon running the configuration, DSC (and the resources being called by the configuration) will simply “make it so”, ensuring that the system exists in the state laid out by the configuration. DSC configurations are also idempotent: the LCM will continue to ensure that machines are configured in whatever state the configuration declares.

To define these configurations, DSC introduces a new PowerShell keyword called **Configuration**. To use DSC resources to configure your environment, first define a PowerShell script block using the Configuration keyword, followed by an identifier and curly braces **{}** to delimit the block. This creates a function that, when executed, generates a MOF file that can be passed to the LCM. You may have other PowerShell commands and variable definitions outside of this Configuration script block, but it should contain the entirety of your DSC configuration. ([You can read more about the Configuration keyword on the Windows PowerShell blog](http://blogs.msdn.com/b/powershell/archive/2013/11/05/understanding-configuration-keyword-in-desired-state-configuration.aspx).)

Inside the configuration block you can define **Node** blocks that specify the desired configuration for each node (computer or VM) within your environment. A node block starts with the **Node** keyword, followed by an identifier for the target computer. This identifier can be a hostname, computer name, or IP address, and it may be represented as a variable. After the computer name, delimit the node block like so:

Configuration MyDscConfiguration

{

Node “ComputerName1”

{

…

}

Node “ComputerName2”

{

…

}

}

Note: in the absence of a node block, the entire configuration will simply be applied to localhost.

Within these Node blocks are **resource instances**. These blocks define a single instance of a resource and its configuration properties. For instance, the following Configuration would Ensure that Remote Server Administration Tools (RSAT) and Bitlocker are installed on a machine called Test-PC1:

Configuration MyDscConfiguration {

Node “TEST-PC1” {

WindowsFeature MyFeatureInstance {

Ensure = “Present”

Name = “RSAT”

}

WindowsFeature My2ndFeatureInstance {

Ensure = “Present”

Name = “Bitlocker”

}

}

}

# Running the configuration function here will generate a MOF file for the LCM

MyDscConfiguration

**Best practice:** Always execute the Configuration function outside the Configuration block at the bottom of your configuration scripts to generate the MOF file that must be passed to the DSC engine.

As you can see, we defined a Configuration function called MyDscConfiguration which will deploy to a Node with the name Test-PC1. As you can see, Node blocks can contain multiple instances of the same resource which can be configured independently. Within these instance blocks, we set properties using key/value pairs. **Ensure** is a commonly used property with the possible values “Present” and “Absent” that ensures whether or not a resource instance exists. In this case, the **Name** property also defines which Windows feature should be ensured to exist (i.e. “RSAT” and “PowerShell”).

Later, if a user decides that Test-PC1 should not have RSAT installed, they can edit the configuration script to ensure that RSAT is absent and re-run the Configuration function. This will generate a new MOF file which, when passed to the LCM, will perform the necessary logic to ensure that RSAT is now absent from the node.

## DependsOn

Another useful keyword is **DependsOn**. Typically (though not necessarily always), DSC will instantiate resource instances in the order that they appear within the configuration. However, DependsOn specifies which resource instances depend on other resource instances, and the LCM will ensure that the dependency graph is traversed properly, regardless of the order in which resource instances are defined. For example, a configuration might specify that a User instance depends on the existence of a Group instance:

Configuration DependsOnExample {

Node Test-PC1 {

Group GroupExample {

Ensure = “Present”

GroupName = “TestGroup”

}

User UserExample {

Ensure = “Present”

FullName = “TestUser”

DependsOn = “GroupExample”

}

}

}

DependsOnExample

## Using New Resources in Your Configuration

Today, DSC ships with 12 resources as part of the **PSDesiredStateConfiguration** module. Other resources in external modules must be placed in $env:PSModulePath in order to be recognized by the LCM. A new cmdlet, **Get-DscResource**, can be used to determine what resources are installed on the system and available for use by the LCM.

([You can read more about consuming resources on the PowerShell blog](http://blogs.msdn.com/b/powershell/archive/2013/12/05/how-to-deploy-and-discover-windows-powershell-desired-state-configuration-resources.aspx).)

Once these modules have been placed in $env:PSModulePath and are properly recognized by Get-DscResource, they still need to be loaded within your Configuration. **Import-DscResource** is a dynamic keyword that can *only* be recognized within a Configuration block (i.e. it is *not* a cmdlet). Import-DscResource supports two parameters:

* **-ModuleName** is the recommended way of using Import-DscResource. It accepts the name of the module that contains the resources to be imported (as well as a string array of module names).
* **-Name** is the name of the resource to import. This is not the friendly name returned as “Name” by Get-DscResource, but the class name used when defining the resource schema (returned as ResourceType by Get-DscResource).

([You can learn more about Import-DscResource on the Windows PowerShell blog](http://blogs.msdn.com/b/powershell/archive/2014/04/25/understanding-import-dscresource-keyword-in-desired-state-configuration.aspx).)

## Configuration Data

With DSC, it is possible to separate configuration data from the logic of your configuration. Consider the structural configuration (for example, a configuration might require that IIS be installed) as separate from the environmental configuration (whether the situation is a test environment vs. a production one—the node names would be different, but the configuration applied to them would be the same). This has the following advantages:

* It allows you to reuse your configuration data for different resources, nodes, and configurations.
* Configuration logic is more reusable if it does not contain hard-coded data. This is similar to good scripting guidelines for functions.
* If some of the data needs to change, you can make the changes in one location, thereby saving time and reducing errors.

**Best Practice:** Write your configuration data in an external .psd1 and pass it to your configuration function as a file using the **–ConfigurationData** parameter. For example, the last line of your configuration script might read:

myDscConfiguration –ConfigurationData myDscConfiguration-ConfigData.psd1

([You can read more about DSC Configuration Data on TechNet](http://technet.microsoft.com/en-us/library/dn249925.aspx).)

## Securing the MOF File

When writing DSC configurations that employ credentials, special care must be taken to ensure the MOF file generated does not contain unencrypted credentials or plaintext passwords. For more information on how to do this, see [this TechNet article on Securing the MOF File.](http://technet.microsoft.com/en-us/library/dn781430.aspx)

# Local Configuration Manager

The engine powering DSC is known as the Local Configuration Manager (LCM). It runs on all target nodes, and it is responsible for calling the configuration resources that are included in a DSC configuration script.

To manage the LCM, two new cmdlets have been introduced:

When running a DSC configuration function (typically on the last line of a configuration script, just outside the Configuration block), Windows PowerShell generates a folder containing a MOF file. This MOF file is understood by the LCM and can be used in conjunction with the LCM using a number of different cmdlets:

* **Get-DscLocalConfigurationManager** retrieves the current LCM properties and their associated values. For example, if ConfigurationModeFrequencyMins is set to 30, we know that the LCM ensures a Configuration is in compliance every 30 minutes.
* **Set-DscLocalConfigurationManager** is used to change the current LCM properties. It accepts the path of the generated
* Get-DscConfiguration
* Get-DscResource
* New-DSCCheckSum
* Restore-DscConfiguration
* Test-DscConfiguration
* Start-DscConfiguration

To see a list of the properties supported by the LCM, and how you can modify the Local Configuration Manager settings on a target node, see [this TechNet article on the LCM](http://technet.microsoft.com/en-us/library/dn249922.aspx).

# Consuming DSC Resources as Modules

Resources are building blocks that you can use to write a DSC script. They must exist within PowerShell modules (even if that module only includes DSC resources) and must include a MOF schema (.schema.mof), a script module (.psm1), and a module manifest (.psd1). A typical directory structure might look something like this:

$env:PSModulePath (folder)

|- <ModuleName> (folder)  
 |- <ModuleName>.psd1 (file, required)

|- DSCResources (folder)

|- <ResourceName1> (folder)

|- <ResourceName>.psm1 (file, required)

|- <ResourceName>.schema.mof (file, required)

|- <ResourceName2> (folder)

|- <ResourceName2>.psm1 (file, required)

|- <ResourceName2>.schema.mof (file, required)

With this module structure, existing \*-Module cmdlets can recognize DSC resources at the ModuleName level just as they would with any PowerShell module. However, the introduction of the DSCResources folder within the module requires a new cmdlet (**Get-DscResource**) and a new dynamic keyword (**Import-DscResource**).

**Get-DscResource** is analogous to a Get-Module for DSC resources: it returns a list of available DSC resources within $env:PSModulePath and their associated parent modules.

# Need more resources

Customers need to be able to deploy and manage the configuration of Windows Server, its applications, roles, features, and infrastructure for the cloud throughout the lifecycle in an easy and maintainable way. Windows PowerShell Desired State Configuration (DSC) provides a configuration platform based on open standards and is flexible enough to function reliably in each stage of the deployment lifecycle (development, test, pre-production, production) as well as during scale-out.

In order for DSC to make Windows Server truly cloud-ready, all components and applications must be fully supported. One or more DSC resources must be created for each Windows Server component and application. Since DSC is implemented with open standards, these resources can then be utilized by Windows PowerShell Desired State Configuration or any other tool that needs to manage Windows Servers.

## Finding More Resources

DSC comes with a set of built-in functionality for configuring resources such as files and folders, server features and roles, registry settings, environment variables, and services and processes. To learn about the full list of built-in DSC resources and how to use them, see [Built-In Windows PowerShell Desired State Configuration Resources](http://technet.microsoft.com/en-us/library/dn249921.aspx).

You can also find a collection of more experimental DSC resources in the [DSC Resource Kit on the TechNet Gallery](http://gallery.technet.microsoft.com/DSC-Resource-Kit-All-c449312d).

In addition, any users interested in experimenting with the [Windows Management Framework 5.0 Preview](http://www.microsoft.com/en-us/download/details.aspx?id=44070), the [Windows 10 Technical Preview for Enterprise](http://www.microsoft.com/en-us/evalcenter/evaluate-windows-technical-preview-for-enterprise), or the [Windows Server Technical Preview](http://www.microsoft.com/en-us/evalcenter/evaluate-windows-server-vnext-technical-preview) can use the PowerShellGet module to download and install new PowerShell modules and DSC resources.

## Creating

You can also create resources to extend the set of built-in DSC resources. To read more about this subject and to learn how to create a custom resource, read the TechNet article on how to [Build Custom Windows PowerShell Desired State Configuration Resources](http://technet.microsoft.com/en-us/library/dn249927.aspx).

# Learn More

A fantastic collection of official and community blog posts on DSC (located on Michael Greene’s blog, “Building Clouds”): <http://blogs.technet.com/b/privatecloud/archive/2014/04/25/desired-state-configuration-blog-series-part-1-learning-about-dsc.aspx>

More of Michael Greene’s DSC-related blog posts: http://blogs.technet.com/b/privatecloud/archive/tags/dsc/

Using DSC on Amazon Web Services: <http://aws.amazon.com/blogs/aws/powershell-dsc-setup-quick-ref>

Windows PowerShell Desired State Configuration Revealed by Ravikanth Chaganti: <http://www.amazon.com/Windows-PowerShell-Desired-Configuration-Revealed/dp/1484200179/ref=sr_1_1?ie=UTF8&qid=1412001567&sr=8-1&keywords=Windows+PowerShell+Desired+State+Configuration+Revealed>

ALM Ranger posters

Azure Extension stuff

<http://powershell.org/wp/ebooks/>