✓ 100 XP

Introduction

2 minutes

When you deploy a set of virtual machines (VMs) to host a production system, it's essential that all the VMs have the same state. They must have the same services installed. VMs should be configured in the same way, and the software on each machine must be the same version. The configuration of each VM can *drift*. You can end up with systems that are prone to failure because of incompatibilities in the setup of each machine. Azure Automation State Configuration addresses many of the problems associated with deploying at scale and managing configuration drift.

Imagine you're an administrator for a retail company that uses Azure Virtual Machines to host its website. Recently one of your VMs was redeployed without the Windows IIS web server feature installed. This issue made your website behave unpredictably.

To reduce the time you spend troubleshooting and maintaining the configuration consistency across your VMs, you decide to set up Azure Automation State Configuration.

Learning objectives

In this module, you'll:

- Identify the capabilities of Azure Automation State Configuration.
- Learn how to onboard VMs for management by Azure Automation.
- Automatically update VMs to maintain a desired state configuration.

Prerequisites

- Familiarity with Azure Virtual Machines
- Ability to run scripts in Azure CLI and PowerShell

Next unit: What is Azure Automation State Configuration?

Continue >

✓ 200 XP

What is Azure Automation State Configuration?

10 minutes

You use Azure Automation State Configuration to make sure that the virtual machines (VMs) in a cluster are in a consistent state. The VMs should have the same software installed and the same configurations.

In this unit, you'll learn about the features and capabilities of Azure Automation. You'll review the declarative model of PowerShell Desired State Configuration (DSC), and you'll explore its benefits.

What is Azure Automation State Configuration?

Azure Automation State Configuration is an Azure service built on PowerShell. It allows you to consistently deploy, reliably monitor, and automatically update the desired state of all your resources. Azure Automation provides tools to define configurations and apply them to real and virtual machines.

Why use Azure Automation State Configuration?

Manually maintaining a correct and consistent configuration for the servers that run your services can be difficult and error prone. Azure Automation State Configuration uses PowerShell DSC to help address these challenges. It centrally manages your DSC artifacts and the DSC process.

Azure Automation State Configuration has a built-in pull server. You can target nodes to automatically receive configurations from this pull server, conform to the desired state, and report back on their compliance. Target virtual or physical Windows or Linux machines, in the cloud or on-premises.

You can use Azure Monitor logs to review the compliance of your nodes by configuring Azure Automation State Configuration to send this data.

What is PowerShell DSC?

PowerShell DSC is a declarative management platform that Azure Automation State Configuration uses to configure, deploy, and control systems. A declarative programming language separates intent (what you want to do) from execution (how do you want to do it). You specify the desired state and let DSC do the work to get there. You don't have to know how to implement or deploy a feature when a DSC resource is available. Instead, you focus on the structure of your deployment.

If you're already using PowerShell, you might wonder why you need DSC. Consider the following example.

When you want to create a share on a Windows server, you might use this PowerShell command:

PowerShell

Copy

```powershell
# Create a file share
New-SmbShare -Name MyFileShare -Path C:\Shared -FullAccess User1 -ReadAccess
User2
```

The script is straightforward and easy to understand. However, if you use this script in production, you'll come across several problems. Consider what might happen if the script runs multiple times or if the user `User2` already has full access rather than only read access.

This approach isn't *idempotent*. Idempotence describes an operation that has the same effect whether you run it once or 10,001 times. To achieve idempotence in PowerShell, you need to add logic and error handling. If the share doesn't exist, you create it. If the share does exist, there's no need to create it. If `User2` exists but doesn't have read access, you add read access.

Your PowerShell script would look something like:

PowerShell

Copy

```
$shareExists = $false
$smbShare = Get-SmbShare -Name $Name -ErrorAction SilentlyContinue
if($smbShare -ne $null)
{
    Write-Verbose -Message "Share with name $Name exists"
    $shareExists = $true
}

if ($shareExists -eq $false)
{
    Write-Verbose "Creating share $Name to ensure it is Present"
    New-SmbShare @psboundparameters
}
else
{
    # Need to call either Set-SmbShare or *ShareAccess cmdlets
    if ($psboundparameters.ContainsKey("ChangeAccess"))
    {
       #...etc., etc., etc
    }
}
```

Other special cases you haven't considered might come to light only when problems arise. DSC handles unexpected cases automatically. With DSC, you describe the result rather than the process to achieve the result.

The following DSC code snippet shows an example:

PowerShell

Copy

```
Configuration Create_Share
{
   Import-DscResource -Module xSmbShare
   # A node describes the VM to be configured

   Node $NodeName
   {
      # A node definition contains one or more resource blocks
      # A resource block describes the resource to be configured on the node
      xSmbShare MySMBShare
      {
          Ensure      = "Present"
          Name        = "MyFileShare"
          Path        = "C:\Shared"
```

```
            ReadAccess  = "User1"
            FullAccess  = "User2"
            Description = "This is an updated description for this share"
        }
    }
}
```

This example uses the `xSmbShare` module. The module tells DSC *how* to check the state for a file share. The DSC Resource Kit currently contains more than 80 resource modules, including one for installing an IIS site. You'll find a link to this resource at the end of the module.

You'll learn more about the structure of the PowerShell DSC code in the next unit.

What is the LCM?

The local configuration manager (LCM) is a component of the Windows Management Framework (WMF) that's on a Windows operating system. The LCM is responsible for updating the state of a node, like a VM, to match the desired state. Every time the LCM runs, it completes the following steps:

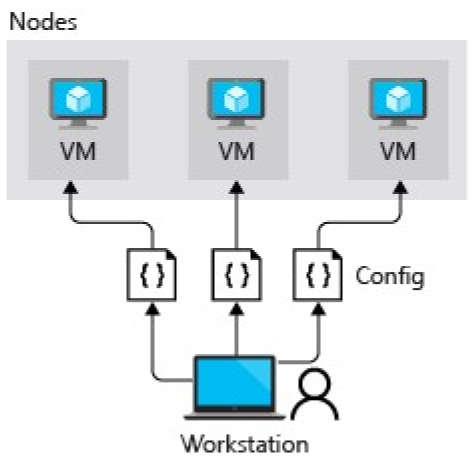
1. **Get**: Get the current state of the node.
2. **Test**: Compare the current state of a node against the desired state by using a compiled DSC script (.mof file).
3. **Set**: Update the node to match the desired state described in the .mof file.

You configure the LCM when you register a VM with Azure Automation.

Push and pull architectures in DSC

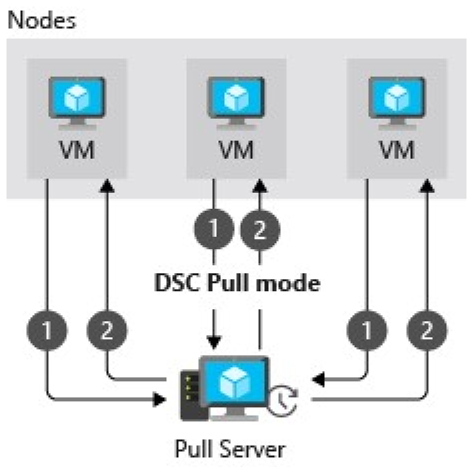
The LCM on each node can operate in two modes.

- **Push mode**: An administrator manually sends, or *pushes*, the configurations toward one or more nodes. The LCM makes sure that the state on each node matches what the configuration specifies.



- **Pull mode**: A *pull server* holds the configuration information. The LCM on each node polls the pull server at regular intervals, by default every 15 minutes, to get the latest configuration details. These requests are denoted as step 1 in the following diagram. In step 2, the pull server sends the details about any configuration changes back to each node.

  In pull mode, each node has to be registered with the pull service.



Both modes have advantages:

- Push mode is easy to set up. It doesn't need its own dedicated infrastructure, and it can run on a laptop. Push mode is helpful to test the functionality of DSC. You could also use push mode to get a newly imaged machine to the baseline desired state.
- Pull mode is useful in an enterprise deployment that spans a large number of machines. The LCM regularly polls the pull server and makes sure the nodes are in the desired state. If an external tool or team applies hotfixes that result in

configuration drift on individual machines, those machines are quickly brought back in line with the configuration you've set. This process can help you achieve a state of continuous compliance for your security and regulatory obligations.

Supported platforms and operating systems

Azure Automation DSC is supported by Azure Cloud Services and other cloud providers, your on-premises infrastructure, or a hybrid of all of these environments.

Azure Automation DSC supports the following operating systems:

- Windows
  - Server 2019
  - Server 2016
  - Server 2012 R2
  - Server 2012
  - Server 2008 R2 SP1
  - 10
  - 8.1
  - 7
- Linux
  - Most variants, but not Debian or Ubuntu 18.04

PowerShell DSC is installed on all Linux machines supported by Azure Automation DSC.

DSC requirements for Windows

For Windows machines, the Azure Desired State Configuration (DSC) VM extension uses WMF to manage the versions of Windows features like Windows PowerShell DSC and Windows Remote Management (WinRM). Azure DSC supports WMF 4.0 and later. So Windows machines must run Windows Server 2012, Windows 7, or later.

The first time the Azure DSC extension is called, it installs an OS-compatible version of WMF on all Windows versions except Windows Server 2016 and later. Windows Server 2016 and later versions already have the latest version of WMF installed. After WMF is installed, the machine requires a restart.

WinRM is enabled on machine nodes that run Windows Server 2012, Windows 7, or later versions of Windows.

Proxy support for the DSC agent is available in Windows builds 1809 and later. Proxy support is unavailable in DSC for previous versions of Windows.

Other DSC requirements

If your nodes are located in a private network, DSC needs the following port and URLs to communicate with Azure Automation:

- **Port**: Only TCP 443 is required for outbound internet access.
- **Global URL**: *.azure-automation.net
- **Global URL of US Gov Virginia**: *.azure-automation.us
- **Agent service**: https://.agentsvc.azure-automation.net

Check your knowledge

1. What is Azure Automation State Configuration?

○ A declarative management platform to configure, deploy, and control systems.

[✓ A service used to write, manage, and compile PowerShell Desired State Configuration (DSC) configurations, import DSC resources, and assign configurations to target nodes.] ✓

Azure Automation State Configuration enables you to ensure that all virtual machines in a collection are in the same consistent state.

○ A service that manages the state configuration on each destination, or node.

2. A PowerShell DSC script _____________.

○ Contains the steps required to configure a virtual machine to get it into a specified state.

Is idempotent.

[✓ Describes the desired state.] ✓

A PowerShell DSC script is declarative. It describes the desired state but doesn't include the steps necessary to achieve that state.

3. Why should you use pull mode instead of push mode for DSC?

[✓ Pull mode is best for complex environments that need redundancy and scale.] ✓

The local configuration manager (LCM) on each node automatically polls the pull server at regular intervals to get the latest configuration details. In push mode, an administrator manually sends the configurations toward the nodes.

Pull mode is easy to set up and doesn't need its own dedicated infrastructure.

Pull mode uses the local configuration manager (LCM) to make sure that the state on each node matches the state specified by the configuration.

Next unit: Use PowerShell DSC to achieve a desired state

Continue >

✓ 100 XP

Use PowerShell DSC to achieve a desired state

10 minutes

You use PowerShell DSC to specify the desired state for a VM. In this unit, you'll learn more about PowerShell DSC and how to use it to control the state of your VMs. In the example scenario, you use PowerShell DSC to make sure that IIS for Windows Server is installed and configured consistently across all of your web servers.

By the end of this unit, you'll:

- Understand node and configuration blocks.
- Understand credential assets.
- Write PowerShell DSC code to install Microsoft IIS idempotently.

DSC resources

You've seen that PowerShell DSC is a declarative task-oriented scripting language. When you need to configure and deploy an Azure resource in a consistent way across a set of VMs, PowerShell DSC can help. You can use PowerShell DSC even when you're not familiar with the technical steps to install and configure the software and services.

Windows Server has a set of built-in PowerShell DSC resources. You can see these resources by running the `Get-DSCResource` PowerShell cmdlet.

PowerShell Copy

```
Get-DscResource | select Name,Module,Properties
```

The following table lists some of the built-in PowerShell DSC resources.

Resource	Description

Resource	Description
File	Manages files and folders on a node
Archive	Decompresses an archive in the .zip format
Environment	Manages system environment variables
Log	Writes a message in the DSC event log
Package	Installs or removes a package
Registry	Manages a node's registry key (except HKEY Users)
Script	Executes PowerShell commands on a node
Service	Manages Windows services
User	Manages local users on a node
WindowsFeature	Adds or removes a role or feature on a node
WindowsOptionalFeature	Adds or removes an optional role or feature on a node
WindowsProcess	Manages a Windows process

For more complex resources, like Active Directory integration, use the DSC Resource Kit, which is updated monthly. You'll find the link to this resource at the end of the module.

The resource you want to configure must already be part of the VM or part of the VM image. Otherwise, the job will fail to compile and run.

Anatomy of a DSC code block

A DSC code block contains four sections. Use the following example to take a closer look. In the example, the numbers aren't part of the syntax. They refer to sections in the discussion that follows.

PowerShell Copy

```
Configuration MyDscConfiguration {               ##1
    Node "localhost" {                            ##2
        WindowsFeature MyFeatureInstance {       ##3
            Ensure = 'Present'
            Name = 'Web-Server'
        }
    }
}
MyDscConfiguration -OutputPath C:\temp\                              ##4
```

The configuration syntax includes these sections:

1. **Configuration**: The configuration block is the outermost script block. It starts with the `Configuration` keyword, and you provide a name. Here, the name of the configuration is `MyDscConfiguration`.

   The configuration block describes the desired configuration. Think of a configuration block like a function, except that it contains a description of the resources to install rather than the code to install them.

   Like a PowerShell function, a configuration block can take parameters. For example, you could parameterize the node name.

   PowerShell Copy

   ```
   Configuration MyDscConfiguration {
   param
   (
       [string] $ComputerName='localhost'
   )

   Node $ComputerName {
       ...
   }
   ```

2. **Node**: You can have one or more node blocks. The node block determines the names of .mof files that are generated when you compile the configuration. For example, the node name `localhost` generates only one *localhost.mof* file. But you

can send that .mof file to any server. You generate multiple .mof files when you use multiple node names.

Use the array notation in the node block to target multiple hosts. For example:

PowerShell

```
Node @('WEBSERVER1', 'WEBSERVER2', 'WEBSERVER3')
```

3. **Resource**: One or more resource blocks can specify the resources to configure. In this case, a single resource block references the `WindowsFeature` resource. The `WindowsFeature` resource here ensures that the Windows feature `Web-Server` is installed.

4. **MyDscConfiguration**: This call invokes the `MyDscConfiguration` block. It's like running a function. When you run a configuration block, it's compiled into a Managed Object Format (MOF) document. MOF is a compiled language created by Desktop Management Task Force, and it's based on interface definition language.

   For every node listed in the DSC script, a .mof file is created in the folder you specified with the `-OutputPath` parameter.

Configuration data in a DSC script

In a configuration data block, you can provide data that the configuration process might need. You apply this data to named nodes, or you apply it globally across all nodes.

A configuration data block is a named block that contains an array of nodes. The array must be named `AllNodes`. Inside the `AllNodes` array, you specify the data for a node by using the `NodeName` variable.

Using the previous scenario, let's say that on the web server that's installed on each node, you want to set the `SiteName` property to different values. You could define a configuration data block like this:

PowerShell

```
$datablock =
@{
    AllNodes =
    @(
        @{
            NodeName = "WEBSERVER1"
            SiteName = "WEBSERVER1-Site"
        },
        @{
            NodeName = "WEBSERVER2"
            SiteName = "WEBSERVER2-Site"
        },
        @{
            NodeName = "WEBSERVER3"
            SiteName = "WEBSERVER3-Site"
        }
    );
}
```

If you want to set a property to the same value in each node, in the `AllNodes` array, specify `NodeName = "*"`.

Secure credentials in a DSC script

A DSC script might require credential information for the configuration process. Avoid putting a credential in plaintext in your source code management tool. Instead, DSC configurations in Azure Automation can reference credentials stored in a `PSCredential` object. You define a parameter for the DSC script by using the `PSCredential` type. Before running the script, get the credentials for the user, use the credentials to create a new `PSCredential` object, and pass this object into the script as a parameter.

Credentials aren't encrypted in .mof files by default. They're exposed as plaintext. To encrypt credentials, use a certificate in your configuration data. The certificate's private key needs to be on the node on which you want to apply the configuration. Certificates are configured through the node's LCM.

Starting in PowerShell 5.1, .mof files on the node are encrypted at rest. In transit, all credentials are encrypted through WinRM.

Push the configuration to a node

After you create a compiled .mof file for a configuration, you can push it to a node by running the `Start-DscConfiguration` cmdlet. If you add the path to the directory, it applies any .mof file it finds in that directory to the node:

PowerShell

Copy

```powershell
Start-DscConfiguration -path D:\
```

This step corresponds to *push mode*, which you learned about in the previous unit.

Pull the configuration for nodes

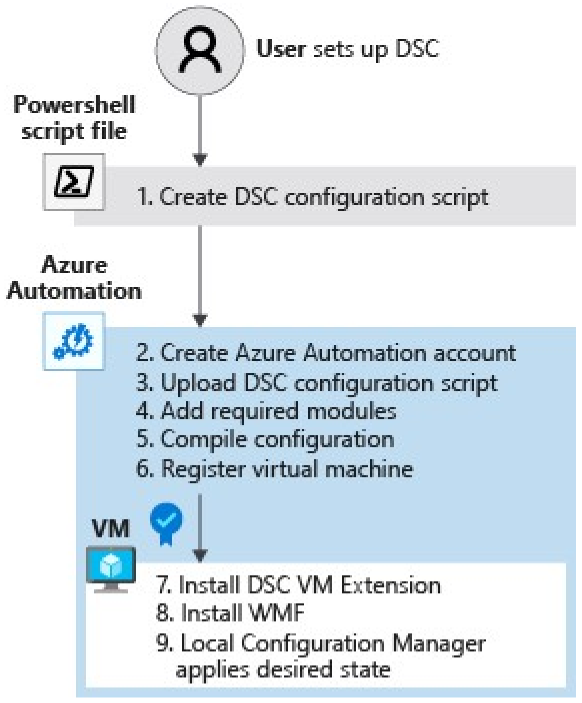
If you have hundreds of VMs on Azure, pull mode is more appropriate than push mode.

You can configure an Azure Automation account to act as a pull service. Just upload the configuration to the Automation account. Then register your VMs with this account.

Before you compile your configuration, import into your automation account any PowerShell modules the DSC process needs. These modules define how to complete the task to achieve the desired state.

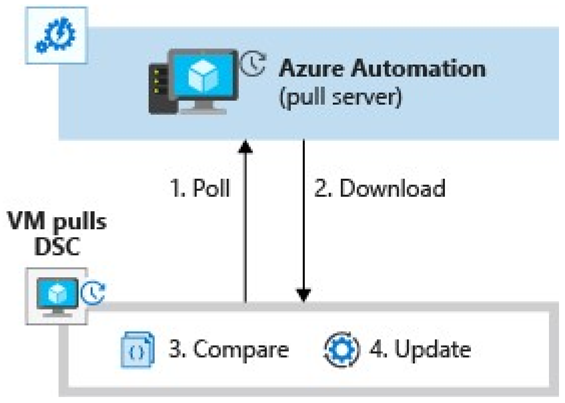
For example, a DSC script in the previous unit used the `xSmbShare` PowerShell module to tell DSC *how* to check the state for a file share. DSC automatically pulls modules from the automation account to the node.

The following diagram shows how to set up Azure Automation State Configuration. We'll explore these steps more in the next unit.


User sets up DSC
Powershell script file
1. Create DSC configuration script
Azure Automation
2. Create Azure Automation account
3. Upload DSC configuration script
4. Add required modules
5. Compile configuration
6. Register virtual machine
VM
7. Install DSC VM Extension
8. Install WMF
9. Local Configuration Manager applies desired state

By default, after 15 minutes, the LCM on the VM polls Azure Automation for any changes to the DSC configuration file. Any changes in the VMs are recorded in the desired state configuration. If you change a configuration, you can upload it to the Automation account to automatically reconfigure the VMs.

The following diagram shows the LCM's process to manage the desired state on the VM.



Credentials are handled natively by your Automation account. This management reduces the complexity of securing and working with credentials.

Next unit: Exercise - Set up a DSC and configure a desired state

Continue >

✓ 100 XP

Summary

2 minutes

You've learned about the capabilities of Azure Automation State Configuration. You've deployed a desired state to a server so that the Windows IIS feature is consistently deployed across a set of nodes.

In particular, you've seen how to:

- Identify the capabilities of Azure Automation State Configuration.
- Enable and register virtual machines for management by Azure Automation.
- Automatically update Virtual Machines to maintain a desired state configuration.

Clean up

The sandbox automatically cleans up your resources when you're finished with this module.

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

Learn more

- Install and configure WMF 5.1
- Troubleshoot DSC - WinRM dependency
- PowerShell DSC for Linux
- DSC Resource Kit

Module incomplete:

Go back to finish