Windows Management Framework 5.0

Production Preview Release Notes

# Overview

Windows Management Framework (WMF) 5.0 Production Preview contains functionality that has been updated from WMF 4.0. WMF 5.0 Production Preview is available for installation on Windows 8.1, Windows Server 2012 R2, Windows Server 2012, Windows 2008 R2 and Windows 7, and contains updated versions of the following features:

* Windows PowerShell
* Windows PowerShell Integrated Scripting Environment (ISE)
* Windows PowerShell Web Services (Management OData IIS Extension)
* Windows PowerShell Desired State Configuration (DSC)
* Windows Remote Management (WinRM)
* Windows Management Instrumentation (WMI)

You should be sure to back up your environments when you run WMF 5.0 Production Preview in production.

WMF 5.0 Production Preview differs from older previews of WMF 5.0 in the following ways:

* It is fully supported until three months after WMF 5.0 RTM. By *fully supported*, we mean we will investigate production-blocking issues, and provide workarounds or updates as necessary.
* All features in the Production Preview are production-ready. No experimental features are in this package.
* You must uninstall all older releases of WMF before installing the Production Preview.

# Requirements

You can install WMF 5.0 Preview Production Preview only on the following operating systems:

| Operating System | Service Pack Level | Editions |
| --- | --- | --- |
| Windows 8.1 |  | All |
| Windows Server 2012 R2 |  | All except IA64 |
| Windows Server 2012 |  | All except IA64 |
| Windows Server 2008 R2 | SP1 | All except IA64 |
| Windows 7 | SP1 |  |

Earlier releases of WMF 5.0 Preview (such as the February 2015 or April 2015 releases) are not required to install WMF 5.0 Production Preview.

Install the latest Windows updates before installing WMF 5.0 Production Preview.

# How to install and uninstall WMF 5.0 Production Preview

## Installation instructions

Download the correct package for your operating system and architecture. The following architectures are supported.

* + Windows Server 2012 R2, Windows 8.1 Pro, and Windows 8.1 Enterprise
    1. x64: Win8.1AndW2K12R2-KB3066437-x64.msu
    2. x86: Win8.1AndW2K12R2-KB3066437-x86.msu
  + Windows Server 2012
    1. x64: Win2K12-KB3066438-x64.msu
  + Windows 7 SP1 and Windows Server 2008 R2 SP1
    1. x64: Win7AndW2K8R2-KB3066439-x64.msu
    2. x86: Win7AndW2K8R2-KB3066439-x86.msu

### Windows 8.1, Windows Server 2012 R2, and Windows Server 2012

Ensure all of the following prerequisites have been met:

* **Windows 8.1**, **Windows Server 2012 R2**,or **Windows Server 2012**
* **Windows PowerShell 2.0 (if applicable)**
* Windows PowerShell 2.0 is disabled by default on Windows Server 2012 R2 Server Core. It can be enabled by running the following two commands:

dism /online /enable-feature:MicrosoftWindowsPowerShellV2

dism /online /enable-feature:MicrosoftWindowsPowerShellV2-WOW64

Installation:

* **Double-click** the appropriate MSU file to start installation, or run the MSU file directly from **Command Prompt**.

### Windows 7 SP1 and Windows Server 2008 r2 sp1

Ensure all of the following prerequisites have been met:

* Ensure that the latest service pack is installed.
* Ensure that WMF 4.0 is installed, or install WMF 4.0 by following the instructions [here](http://www.microsoft.com/en-us/download/details.aspx?id=40855).

Installation:

* **Double-click** the appropriate MSU file to start installation, or run the MSU file directly from Command Prompt.

### Desired State Configuration on Windows Server 2012, Windows Server 2008 R2 SP1, Windows 7 SP1

Windows PowerShell Desired State Configuration (DSC) depends on WinRM. WinRM is enabled by default on Windows Server 2012, but it is not enabled by default on Windows 8, Windows 7, or Windows Server 2008 R2. To enable WinRM, in a Windows PowerShell elevated session, run **Set-WSManQuickConfig**.

## Uninstallation instructions

### By using Control Panel

1. Open **Control Panel**.
2. Open **Programs**, then open **Uninstall a program**.
3. Click **View installed updates**.
4. Select **Windows Management Framework 5.0 Production Preview April 2015** from the list of installed updates. This corresponds to KB3066437, KB3066438, or KB3066439. Click **Uninstall.**

### By using Command Prompt

1. Open **Command Prompt**.
2. Run the following command:

wusa /uninstall /kb:3066437

or

wusa /uninstall /kb:3066438

or

wusa /uninstall /kb:3066439

# New scenarios enabled by WMF 5.0 Production Preview

The following table provides an overview of the scenarios that are now in the **Stable** state in WMF 5.0 Production Preview.

|  |  |
| --- | --- |
| **Scenario** | **Design Status** |
| Side-by-side installation of DSC resources and PowerShell Modules | Stable |
| Separation of node and configuration IDs | Stable |
| PSDesiredStateConfiguration Module version updated to 1.1 | Stable |
| PowerShell Information Stream | Stable |

# Scenarios enabled by WMF 5.0 Production Preview

The following table provides an overview of all scenarios enabled by WMF 5.0 Production Preview.

| **Scenario** |
| --- |
| New and updated cmdlets based on community feedback |
| Generate Windows PowerShell cmdlets based on an OData endpoint with ODataUtils |
| Manage .ZIP archives through new cmdlets |
| Interact with symbolic links using improved Item cmdlets |
| Network Switch management with Windows PowerShell |
| DSC authoring improvements in Windows PowerShell ISE |
| 32-bit support for the configuration keyword in DSC |
| Audit Windows PowerShell usage by transcription and logging |
| Extract and parse structured object out of string content |
| PowerShell Information Stream |
| Configure DSC’s Local Configuration Manager with the meta-configuration attribute |
| Configure piece by piece with partial configurations in DSC |
| Manage with cross-computer dependencies in DSC |
| More control over configurations in DSC |
| Find more detail about configuration status in DSC |
| Support for -? during DSC configuration compilation |
| Support for DSC RunAsCredential |
| Rich information for DSC LCM State |
| Side-by-Side installation of DSC Resources and PowerShell Modules |
| PSDesiredStateConfiguration Module version updated to 1.1 |
| Discover and install software with PackageManagement |
| Discover modules and DSC resources with PowerShellGet |
| Develop with classes in Windows PowerShell |
| More control and remoting in Windows PowerShell Debugging |
| Report configuration status from DSC to a central location |
| Software Inventory Logging. |

## New and updated cmdlets based on community feedback

We have added some new Utility cmdlets as well as updated existing cmdlets based on feedback from the community via [Connect](https://connect.microsoft.com/powershell). We continue to go through the feedback, and address as many of your requests as possible.

**Format-Hex** lets you view text or binary data in hexadecimal format. Here is an example, looking at the hexadecimal representation of a string, as well as the binary contents of a Word document.

PS C:\> "hello world" | Format-Hex

0 1 2 3 4 5 6 7 8 9 A B C D E F

00000000 68 65 6C 6C 6F 20 77 6F 72 6C 64 hello world

PS C:\> Format-Hex -Path 'C:\Users\slee\Downloads\DSP1009\_1 1 1.doc'

Path: C:\Users\slee\Downloads\DSP1009\_1 1 1.doc

0 1 2 3 4 5 6 7 8 9 A B C D E F

00000000 D0 CF 11 E0 A1 B1 1A E1 00 00 00 00 00 00 00 00 ÐÏ.à¡±.á........

00000010 00 00 00 00 00 00 00 00 3E 00 03 00 FE FF 09 00 ........>...þ...

00000020 06 00 00 00 00 00 00 00 00 00 00 00 08 00 00 00 ................

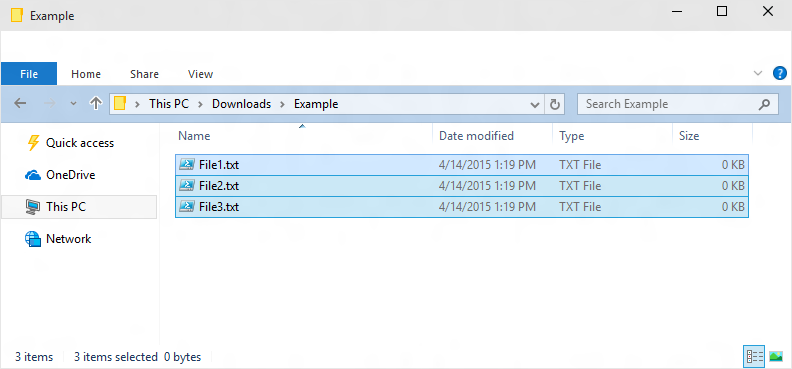
00000030 DF 01 00 00 00 00 00 00 00 10 00 00 E1 01 00 00 ß...........á...

00000040 01 00 00 00 FE FF FF FF 00 00 00 00 DB 01 00 00 ....þ.......Û...

00000050 DC 01 00 00 DD 01 00 00 DE 01 00 00 E3 01 00 00 Ü...Ý...Þ...ã...

00000060 4F 02 00 00 D8 02 00 00 79 03 00 00 FF FF FF FF O...Ø...y.......

**Get-Clipboard** and **Set-Clipboard** make it easier for you to transfer content to and from a Windows PowerShell session. The following example uses File Explorer to copy three files:



Now, you can easily access the contents of the clipboard as a list of files:

PS C:\> Get-Clipboard -Format FileDropList

Directory: C:\Users\slee\Downloads\Example

Mode LastWriteTime Length Name

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

-a---- 4/14/2015 1:19 PM 0 File2.txt

-a---- 4/14/2015 1:19 PM 0 File3.txt

-a---- 4/14/2015 1:19 PM 0 File1.txt

The Clipboard cmdlets support images, audio files, file lists, and text.

Sometimes in your scripts, you must create a temporary file. You can easily do this with the **New-TemporaryFile** cmdlet:

PS C:\> $tempFile = New-TemporaryFile

PS C:\> $tempFile.FullName

C:\Users\slee\AppData\Local\Temp\tmp375.tmp

**Out-File**, **Add-Content**, and **Set-Content** now have a new **–NoNewline** switch which simply omits a new line after the output.

PS C:\Users\slee> "This is " | Out-File -FilePath Example.txt -NoNewline

PS C:\Users\slee> "a single " | Add-Content -Path Example.txt -NoNewline

PS C:\Users\slee> "sentence." | Add-Content -Path Example.txt -NoNewline

PS C:\Users\slee> Get-Content .\Example.txt

This is a single sentence.

Without –NoNewline specified, each fragment would be on a separate line:

PS C:\Users\slee> "This is " | Out-File -FilePath Example.txt

PS C:\Users\slee> "a single " | Add-Content -Path Example.txt

PS C:\Users\slee> "sentence." | Add-Content -Path Example.txt

PS C:\Users\slee> Get-Content .\Example.txt

This is

a single

sentence.

**Get-ChildItem** now has a **–Depth** parameter you use with **–Recurse** to limit the recursion:

PS C:\Users\slee\Downloads\Example> Get-ChildItem -Recurse -Depth 0

Directory: C:\Users\slee\Downloads\Example

Mode LastWriteTime Length Name

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

d----- 4/14/2015 5:36 PM Depth0

-a---- 4/14/2015 1:19 PM 0 File1.txt

-a---- 4/14/2015 1:19 PM 0 File2.txt

-a---- 4/14/2015 1:19 PM 0 File3.txt

PS C:\Users\slee\Downloads\Example> Get-ChildItem -Recurse -Depth 1

Directory: C:\Users\slee\Downloads\Example

Mode LastWriteTime Length Name

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

d----- 4/14/2015 5:36 PM Depth0

-a---- 4/14/2015 1:19 PM 0 File1.txt

-a---- 4/14/2015 1:19 PM 0 File2.txt

-a---- 4/14/2015 1:19 PM 0 File3.txt

Directory: C:\Users\slee\Downloads\Example\Depth0

Mode LastWriteTime Length Name

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

d----- 4/14/2015 5:33 PM Depth1

Often in scripts (or perhaps writing a DSC resource), you have the need for a unique identifier. GUIDs work well, and it is easy to call the .NET Framework Guid class to generate one, but having a cmdlet makes this more discoverable for end users who are not already familiar with the .NET Framework class:

PS C:\> New-Guid

Guid

----

e19d6ea5-3cc2-4db9-8095-0cdaed5a703d

File version information can be misleading, particularly in cases where the file was patched. This release of WMF Production Preview adds new **FileVersionRaw** and **ProductVersionRaw** script properties to FileInfo objects. Here are the properties as displayed for powershell.exe:

PS C:\> Get-Process -Id $pid -FileVersionInfo | fl \*version\* -Force

FileVersionRaw : 10.0.10055.0

ProductVersionRaw : 10.0.10055.0

FileVersion : 10.0.10055.0 (fbl\_srv2.150402-1826)

ProductVersion : 10.0.10055.0

**Convert-String** exposes "replace by magic" functionality. Provide before and after examples of how you want text to look, and Convert-String formats your text automatically. Here's a demo - taking somebody's first and last name, and replacing it with their last name, a comma, the first initial of their last name, and a dot. Try it with a regex, and see how long it takes you.

PS C:\> "Lee Holmes", "Steve Lee", "Jeffrey Snover" | Convert-String -Example "Bill Gates = Gates, B.","John Smith = Smith, J."

Holmes, L.

Lee, S.

Snover, J.

This also introduces some additional functionality for the ConvertFrom-String cmdlet:

* Removes the extent text property by default. You can include it with the -IncludeExtent parameter.
* Many learning algorithm bug fixes from MVP and community feedback.
* A new -UpdateTemplate parameter to save the results of the learning algorithm into a comment in the template file. This makes the learning process (the slowest stage) a one-time cost. Running Convert-String with a template that contains the encoded learning algorithm is now nearly instantaneous.

## Generate Windows PowerShell cmdlets based on an OData endpoint with ODataUtils

The ODataUtils module allows generation of Windows PowerShell cmdlets from REST endpoints that support OData. The following incremental enhancements are in the Microsoft.PowerShell.ODataUtils Windows PowerShell module.

* Channel additional information from server-side endpoint to client side.
* Client-side paging support
* Server-side filtering by using the -Select property
* Support for web request headers

The proxy cmdlets generated by the Export-ODataEndPointProxy cmdlet provide additional information (not mentioned in the $metadata used during the client-side proxy generation) from the server side OData endpoint on the information stream (a new Windows PowerShell 5.0 feature). Here is an example of how to get that information.

Import-Module Microsoft.PowerShell.ODataUtils -Force

$generatedProxyModuleDir = Join-Path -Path $env:SystemDrive -ChildPath 'ODataDemoProxy'

$uri = "http://services.odata.org/V3/(S(fhleiief23wrm5a5nhf542q5))/OData/OData.svc/"

Export-ODataEndpointProxy -Uri $uri -OutputModule $generatedProxyModuleDir -Force -AllowUnSecureConnection -Verbose -AllowClobber

Import-Module $generatedProxyModuleDir -Force

# In the below command, we are retrieving top 1 product.

# By specifying -IncludeTotalResponseCount parameter,

# we are getting the total count of all the Product records

# available on the server side. This information

# is surfaced on the client side through the Information stream.

$product = Get-Product -Top 1 -AllowUnsecureConnection -AllowAdditionalData -IncludeTotalResponseCount -InformationVariable infoStream

# The information stream contains the additional

# infomration sent from the server side.

$additionalInfo = $infoStream.GetEnumerator() | % MessageData

# 'Odata.Count' indicates the total product records

# available on the server side Odata endpoint.

$additionalInfo['odata.count']

You can get the records from the server side in batches, by using client-side paging support. This is useful when you must get a large amount of data from the server over the network.

$skipCount = 0

$batchSize = 3

# Client Side Paging Support: The records from the server side

# are retrieved in batches of $batchSize

while($skipCount -le $additionalInfo['odata.count'])

{

Get-Product -AllowUnsecureConnection -AllowAdditionalData -Top $batchSize -Skip $skipCount

$skipCount += $batchSize

}

The generated proxy cmdlets support a –Select property, which you can use as a filter to receive only the record properties that the client needs. This reduces the amount of data that is transferred over the network, because the filtering occurs on the server side.

# In the below example only the Name property of the

# Product record is retrieved from the server side.

Get-Product -Top 2 -AllowUnsecureConnection -AllowAdditionalData -Select Name

The Export-ODataEndpointProxy cmdlet, and the proxy cmdlets generated by it, now support a Headers property (supply values as a hash table), which you can use to channel any additional information that is expected by the server-side OData endpoint. In the following example, you can channel a Subscription key through Headers for services that are expecting a Subscription key for authentication.

# As an example, In the below command 'XXX' is the authentication used by the

# Export-ODataEndpointProxy cmdlet to interact with the server side

# OData endpoint accessed through $endPointUri.

Export-ODataEndpointProxy -Uri $endPointUri -OutputModule $generatedProxyModuleDir -Force -AllowUnSecureConnection -Verbose -Headers @{'subscription-key'='XXXX'}

## Manage .ZIP archives with new cmdlets

Two new cmdlets, Compress-Archive and Expand-Archive, let you compress and expand ZIP files.

### Compress-Archive

The Compress-Archive cmdlet creates a new archive file from specified files. An archive file allows multiple files to be packaged and optionally compressed into a single file for easier handling and storage. An archive file can be compressed by using a compression algorithm specified in the -CompressionLevel parameter.

|  |
| --- |
| Compress-Archive -LiteralPath <String[]> [-DestinationPath] <String> [-Update] [-CompressionLevel <Microsoft.PowerShell.Commands.CompressionLevel>] |
| Compress-Archive [-Path] <String[]> [-DestinationPath] <String> [-Update] [-CompressionLevel <Microsoft.PowerShell.Commands.CompressionLevel>] |

### Expand-Archive

The Expand-Archive cmdlet extracts files from a specified archive file. An archive file allows multiple files to be packaged and optionally compressed into a single file for easier handling and storage.

|  |
| --- |
| Expand-Archive -LiteralPath <String> [-DestinationPath] <String> |
| Expand-Archive [-Path] <String> [-DestinationPath] <String> |

## Interact with Symbolic Links using improved Item cmdlets

To support symbolic links, \*-Item and a few related cmdlets have been extended. Now you can create symbolic links in a single, simple line with New-Item.

You’ll notice that the Item related cmdlets (Remove-Item, Get-ChildItem) behave very similarly to before.

The following shows some use cases of the new capabilities:

### SYMBOLIC LINK FILES

# Create a new symbolic link file named MySymLinkFile.txt in C:\Temp which links to $pshome\profile.ps1

cd C:\Temp

New-Item -ItemType SymbolicLink -Name MySymLinkFile.txt -Target $pshome\profile.ps1 # File

# Target is an alias to the Value parameter

# Equivalent to above

New-Item -ItemType SymbolicLink -Path C:\Temp -Name MySymLinkFile.txt -Value $pshome\profile.ps1

# Equivalent to above

New-Item -ItemType SymbolicLink -Path C:\Temp\MySymLinkFile.txt -Value $pshome\profile.ps1

# Equivalent to above

New-Item -ItemType SymbolicLink -Name C:\Temp\MySymLinkFile.txt -Value $pshome\profile.ps1

### SYMBOLIC LINK DIRECTORIES

# Create a new symbolic link directory named MySymLinkDir in C:\Temp which links to the $pshome folder

# ItemType is the same for files and directories - autodetect based on specified target

cd C:\Temp

New-Item -ItemType SymbolicLink -Name MySymLinkDir -Target $pshome # Directory

# Target is an alias to the Value parameter

# Similar to above, any combination of Path and Name also works

New-Item -ItemType SymbolicLink -Path C:\Temp -Name MySymLinkDir -Value $pshome

New-Item -ItemType SymbolicLink -Path C:\Temp\MySymLinkDir -Value $pshome

New-Item -ItemType SymbolicLink -Name C:\Temp\MySymLinkDir -Value $pshome

### HARD LINKS

New-Item -ItemType HardLink -Path C:\Temp -Name MyHardLinkFile.txt -Value $pshome\profile.ps1

# Same combinations of Path and Name allowed as described above

### DIRECTORY JUNCTIONS

New-Item -ItemType Junction -Path C:\Temp\MyJunctionDir -Value $pshome

# Same combinations of Path and Name allowed as described above

# GET-CHILDITEM

# Append link type column to Mode property and display with Get-ChildItem

# Use 'l' for all link types

# Increase the width of the Length column by 4 (from 10 to 14)

Get-ChildItem C:\Temp | sort LastWriteTime -Descending

Directory: C:\Temp

Mode LastWriteTime Length Name

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

-a---- 6/13/2014 3:00 PM 16 File.txt

-a---- 6/13/2014 3:00 PM 98956046499840 My90TB.vhd

d----- 6/13/2014 3:00 PM Directory

-a---l 6/13/2014 3:21 PM 0 MySymLinkFile.txt

d----l 6/13/2014 3:22 PM MySymLinkDir

-a---l 6/13/2014 3:23 PM 23304 MyHardLinkFile.txt

d----l 6/13/2014 3:24 PM MyJunctionDir

# New Target property

# Works with any link type

# Not displayed in the default table view

# Displayed in the default list view

# New LinkType property with values: SymbolicLink

Get-ChildItem C:\Temp\MySymLinkFile.txt | Format-List

Directory: C:\Temp

Name : MySymLinkFile.txt

Length : 0

Mode : -a---l

LinkType : SymbolicLink

Target : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1

CreationTime : 6/16/2014 3:21:01 PM

LastWriteTime : 6/16/2014 3:21:01 PM

LastAccessTime : 6/16/2014 3:21:01 PM

VersionInfo : File: C:\Temp\MySymLinkFile.txt

InternalName:

OriginalFilename:

FileVersion:

FileDescription:

Product:

ProductVersion:

Debug: False

Patched: False

PreRelease: False

PrivateBuild: False

SpecialBuild: False

Language:

# REMOVE-ITEM

# Works like any other item type

# Removes MySymLinkFile.txt

Remove-Item C:\Temp\MySymLinkFile.txt

# Returns an error as this is a reparse point.

Remove-Item C:\Temp\MySymLinkDir

# Removes the files in the target directory and MySymLinkDir

Remove-Item C:\Temp\MySymLinkDir -Force

## Network Switch management with Windows PowerShell (improvements)

The Get-NetworkSwitchEthernetPort cmdlet now returns the following additional information with instances:

* IPAddress – the IP address associated with the port
* PortMode – the port mode: access, route or trunk
* AccessVLAN – the ID of the VLAN associated with this port in access mode
* TrunkedVLANList – a list of IDs of VLANs associated with this port in trunk mode

### Fundamental network switch management with Windows PowerShell

The Network Switch cmdlets, introduced in the first WMF 5.0 Preview, enable you to apply switch, virtual LAN (VLAN), and basic Layer 2 network switch port configuration to Windows Server 2012 R2 logo-certified network switches. Microsoft remains committed to supporting the [Datacenter Abstraction](http://technet.microsoft.com/en-us/cloud/dal.aspx) Layer (DAL) vision, and to show value for our customers and partners in this space. Using these cmdlets you can perform:

* + - Global switch configuration, such as:
      * Sett host name
      * Set switch banner
      * Persist configuration
      * Enable or disable feature
    - VLAN configuration:
      * Create or remove VLAN
      * Enable or disable VLAN
      * Enumerate VLAN
      * Set friendly name to a VLAN
    - Layer 2 port configuration:
      * Enumerate ports
      * Enable or disable ports
      * Set port modes and properties
      * Add or associate VLAN to Trunk or Access on the port

Start exploring by looking for all of the NetworkSwitch cmdlets!

PS> Get-Command \*-NetworkSwitch\*

|  |  |  |
| --- | --- | --- |
| CommandType | Name | Source |
| ----------- | ---- | ------ |
| Function | Disable-NetworkSwitchEthernetPort | NetworkSwitch |
| Function | Disable-NetworkSwitchFeature | NetworkSwitch |
| Function | Disable-NetworkSwitchVlan | NetworkSwitch |
| Function | Enable-NetworkSwitchEthernetPort | NetworkSwitch |
| Function | Enable-NetworkSwitchFeature | NetworkSwitch |
| Function | Enable-NetworkSwitchVlan | NetworkSwitch |
| Function | Get-NetworkSwitchEthernetPort | NetworkSwitch |
| Function | Get-NetworkSwitchFeature | NetworkSwitch |
| Function | Get-NetworkSwitchGlobalData | NetworkSwitch |
| Function | Get-NetworkSwitchVlan | NetworkSwitch |
| Function | New-NetworkSwitchVlan | NetworkSwitch |
| Function | Remove-NetworkSwitchEthernetPortIPAddress | NetworkSwitch |
| Function | Remove-NetworkSwitchVlan | NetworkSwitch |
| Function | Restore-NetworkSwitchConfiguration | NetworkSwitch |
| Function | Save-NetworkSwitchConfiguration | NetworkSwitch |
| Function | Set-NetworkSwitchEthernetPortIPAddress | NetworkSwitch |
| Function | Set-NetworkSwitchPortMode | NetworkSwitch |
| Function | Set-NetworkSwitchPortProperty | NetworkSwitch |
| Function | Set-NetworkSwitchVlanProperty | NetworkSwitch |

More information is available in Jeffrey Snover’s WMF 5.0 Preview announcement blog post: <http://blogs.technet.com/b/windowsserver/archive/2014/04/03/windows-management-framework-v5-preview.aspx>

## DSC authoring improvements in Windows PowerShell ISE

Authoring DSC configurations in Windows PowerShell ISE is much easier, thanks to the following improvements:

* List all DSC resources within a **configuration** block or **node** block by entering **Ctrl+Space** on a blank line within it.
* Automatic completion on resource properties that are of the **enumeration** type.
* Automatic completion on the **DependsOn** property of DSC resources, based on other resource instances in the configuration.
* Better tab completion of resource property values.

**Note:** You must have an empty string for resource property values before you can use Ctrl+Space to list the options. Pressing **Tab** cycles through options.

## 32-bit support for the Configuration keyword in DSC

The Configuration keyword is now supported in WOW64 on a 64-bit computer. This means that a DSC configuration can be defined and compiled within a 32-bit process such as Windows PowerShell ISE (x86) running on a 64-bit computer.

## Audit Windows PowerShell usage by using transcription and logging

### Enhanced Windows PowerShell transcription options

Windows PowerShell transcription has been improved to apply to all hosting applications (such as Windows PowerShell ISE) rather than just the console host (powershell.exe).

In addition to extending for transcripting, the transcripting functionality itself has been updated to support arbitrary nesting of transcripts, additional metadata in the resulting transcript header, and the ability to set a transcription output directory (to support centralized log collection).

Transcription options (including enabling a system-wide transcript) can be configured by configuring the **Turn on PowerShell Transcription** Group Policy setting (in Administrative Templates -> Windows Components -> Windows PowerShell).

### Script tracing and logging

While Windows PowerShell already has the **LogPipelineExecutionDetails** Group Policy setting to log the invocation of cmdlets, Windows PowerShell’s scripting language has plenty of features that you might want to log and/or audit. The new Detailed Script Tracing feature lets you enable detailed tracking and analysis of Windows PowerShell scripting use on a system. After you enable detailed script tracing, Windows PowerShell logs all script blocks to the ETW event log, **Microsoft-Windows-PowerShell/Operational**. If a script block creates another script block (for example, a script that calls the Invoke-Expression cmdlet on a string), that resulting script block is logged as well.

Logging of these events can be enabled through the **Turn on PowerShell Script Block Logging** Group Policy setting (in Administrative Templates -> Windows Components -> Windows PowerShell).

The events are:

|  |  |
| --- | --- |
| Channel | Operational |
| Level | Verbose |
| Opcode | Create |
| Task | CommandStart |
| Keyword | Runspace |
| EventId | Engine\_ScriptBlockCompiled (0x1008 = 4104) |
| Message | Creating Scriptblock text (%1 of %2):  %3  ScriptBlock ID: %4 |

The text embedded in the message is the extent of the script block compiled. The ID is a GUID that is retained for the life of the script block.

When you enable verbose logging, the feature writes begin and end markers:

|  |  |
| --- | --- |
| Channel | Operational |
| Level | Verbose |
| Opcode | Open (/ Close) |
| Task | CommandStart (/ CommandStop) |
| Keyword | Runspace |
| EventId | ScriptBlock\_Invoke\_Start\_Detail (0x1009 = 4105) /  ScriptBlock\_Invoke\_Complete\_Detail (0x100A = 4106) |
| Message | Started (/ Completed) invocation of ScriptBlock ID: %1  Runspace ID: %2 |

The ID is the GUID representing the script block (that can be correlated with event ID 0x1008), and the Runspace ID represents the runspace in which this script block was run.

Percent signs in the invocation message represent structured ETW properties. While they are replaced with the actual values in the message text, a more robust way to access them is to retrieve the message with the Get-WinEvent cmdlet, and then use the **Properties** array of the message.

Here's an example of how this functionality can help unwrap a malicious attempt to encrypt and obfuscate a script:

## Malware

function SuperDecrypt

{

    param($script)

    $bytes = [Convert]::FromBase64String($script)

    ## XOR “encryption”

    $xorKey = 0x42

    for($counter = 0; $counter -lt $bytes.Length; $counter++)

    {

        $bytes[$counter] = $bytes[$counter] -bxor $xorKey

    }

    [System.Text.Encoding]::Unicode.GetString($bytes)

}

$decrypted = SuperDecrypt "FUIwQitCNkInQm9CCkItQjFCNkJiQmVCEkI1QixCJkJlQg=="

Invoke-Expression $decrypted

Running this generates the following log entries:

Compiling Scriptblock text (1 of 1):

function SuperDecrypt

{

param($script)

$bytes = [Convert]::FromBase64String($script)

## XOR "encryption"

$xorKey = 0x42

for($counter = 0; $counter -lt $bytes.Length; $counter++)

{

$bytes[$counter] = $bytes[$counter] -bxor $xorKey

}

[System.Text.Encoding]::Unicode.GetString($bytes)

}

ScriptBlock ID: ad8ae740-1f33-42aa-8dfc-1314411877e3

Compiling Scriptblock text (1 of 1):

$decrypted = SuperDecrypt "FUIwQitCNkInQm9CCkItQjFCNkJiQmVCEkI1QixCJkJlQg=="

ScriptBlock ID: ba11c155-d34c-4004-88e3-6502ecb50f52

Compiling Scriptblock text (1 of 1):

Invoke-Expression $decrypted

ScriptBlock ID: 856c01ca-85d7-4989-b47f-e6a09ee4eeb3

Compiling Scriptblock text (1 of 1):

Write-Host 'Pwnd'

ScriptBlock ID: 5e618414-4e77-48e3-8f65-9a863f54b4c8

If the script block length exceeds what ETW is capable of holding in a single event, Windows PowerShell breaks the script into multiple parts. Here is sample code to recombine a script from its log messages:

    $created = Get-WinEvent -FilterHashtable @{ ProviderName="Microsoft-Windows-PowerShell"; Id = 4104 } |

        Where-Object { $\_.<...> }

    $sortedScripts = $created | sort { $\_.Properties[0].Value }

    $mergedScript = -join ($sortedScripts | % { $\_.Properties[2].Value })

As with all logging systems that have a limited retention buffer (i.e., ETW logs), one attack against this infrastructure is to flood the log with spurious events to hide earlier evidence. To protect yourself from this attack, ensure that you have some form of event log collection set up (i.e., Windows Event Forwarding, <http://www.nsa.gov/ia/_files/app/Spotting_the_Adversary_with_Windows_Event_Log_Monitoring.pdf>) to move event logs off of the computer as soon as possible.

### CMS (Cryptographic Message Syntax) cmdlets

The Cryptographic Message Syntax cmdlets support encryption and decryption of content using the IETF standard format for cryptographically protecting messages as documented by [RFC5652](http://tools.ietf.org/html/rfc5652).

  Get-CmsMessage [-Content] <string>

  Get-CmsMessage [-Path] <string>

  Get-CmsMessage [-LiteralPath] <string>

  Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-Content] <string> [[-OutFile] <string>]

  Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-Path] <string> [[-OutFile] <string>]

  Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-LiteralPath] <string> [[-OutFile] <string>]

  Unprotect-CmsMessage [-EventLogRecord] <EventLogRecord> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

  Unprotect-CmsMessage [-Content] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

  Unprotect-CmsMessage [-Path] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

  Unprotect-CmsMessage [-LiteralPath] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

The CMS encryption standard implements public key cryptography, where the keys used to encrypt content (the *public key*) and the keys used to decrypt content (the *private key*) are separate.

Your public key can be shared widely, and is not sensitive data. If any content is encrypted with this public key, only your private key can decrypt it. For more information about Public Key Cryptography, see: <http://en.wikipedia.org/wiki/Public-key_cryptography>.

To be recognized in Windows PowerShell, encryption certificates require a unique key usage identifier (EKU) to identify them as data encryption certificates (like the identifiers for 'Code Signing', 'Encrypted Mail').

Here is an example of creating a certificate that is good for Document Encryption:

(Change the text in **Subject** to your name, email, or other identifier), and put in a file (i.e.: DocumentEncryption.inf):

  [Version]

  Signature = "$Windows NT$"

  [Strings]

  szOID\_ENHANCED\_KEY\_USAGE = "2.5.29.37"

  szOID\_DOCUMENT\_ENCRYPTION = "1.3.6.1.4.1.311.80.1"

  [NewRequest]

  Subject = "[cn=me@somewhere.com](mailto:cn=me@somewhere.com)"

  MachineKeySet = false

  KeyLength = 2048

  KeySpec = AT\_KEYEXCHANGE

  HashAlgorithm = Sha1

  Exportable = true

  RequestType = Cert

  KeyUsage = "CERT\_KEY\_ENCIPHERMENT\_KEY\_USAGE | CERT\_DATA\_ENCIPHERMENT\_KEY\_USAGE"

  ValidityPeriod = "Years"

  ValidityPeriodUnits = "1000"

  [Extensions]

  %szOID\_ENHANCED\_KEY\_USAGE% = "{text}%szOID\_DOCUMENT\_ENCRYPTION%"

Then run:

  certreq -new DocumentEncryption.inf DocumentEncryption.cer

And you can now encrypt and decrypt content:

106 [C:\temp]  
>> $protected = "Hello World" | Protect-CmsMessage -To "\*me@somewhere.com\*"

107 [C:\temp]  
>> $protected  
  
-----BEGIN CMS-----  
MIIBqAYJKoZIhvcNAQcDoIIBmTCCAZUCAQAxggFQMIIBTAIBADA0MCAxHjAcBgNVBAMMFWxlZWhv  
bG1AbWljcm9zb2Z0LmNvbQIQQYHsbcXnjIJCtH+OhGmc1DANBgkqhkiG9w0BAQcwAASCAQAnkFHM  
proJnFy4geFGfyNmxH3yeoPvwEYzdnsoVqqDPAd8D3wao77z7OhJEXwz9GeFLnxD6djKV/tF4PxR  
E27aduKSLbnxfpf/sepZ4fUkuGibnwWFrxGE3B1G26MCenHWjYQiqv+Nq32Gc97qEAERrhLv6S4R  
G+2dJEnesW8A+z9QPo+DwYU5FzD0Td0ExrkswVckpLNR6j17Yaags3ltNVmbdEXekhi6Psf2MLMP  
TSO79lv2L0KeXFGuPOrdzPAwCkV0vNEqTEBeDnZGrjv/5766bM3GW34FXApod9u+VSFpBnqVOCBA  
DVDraA6k+xwBt66cV84OHLkh0kT02SIHMDwGCSqGSIb3DQEHATAdBglghkgBZQMEASoEEJbJaiRl  
KMnBoD1dkb/FzSWAEBaL8xkFwCu0e1ZtDj7nSJc=  
-----END CMS-----

108 [C:\temp]  
>> $protected | Unprotect-CmsMessage  
Hello World

Any parameter of type **CMSMessageRecipient** supports identifiers in the following formats:

* An actual certificate (as retrieved from the certificate provider)
* Path to the a file containing the certificate
* Path to a directory containing the certificate
* Thumbprint of the certificate (used to look in the certificate store)
* Subject name of the certificate (used to look in the certificate store)

To view document encryption certificates in the certificate provider, you can use the -**DocumentEncryptionCert** dynamic parameter:

58 [Cert:\currentuser\my]

>> dir -DocumentEncryptionCert

## Extract and parse structured objects out of string content

In collaboration with [Microsoft Research](http://research.microsoft.com/), a new **ConvertFrom-String** cmdlet has been added.

This cmdlet supports two modes: basic delimited parsing, and auto generated example-driven parsing.

Delimited parsing, by default, splits the input at white space, and assigns property names to the resulting groups. You can customize the delimiter:

1 [C:\temp]  
>> "Hello World" | ConvertFrom-String | Format-Table -Auto

P1    P2  
--    --

The cmdlet also supports auto-generated example-driven parsing based on the [FlashExtract](http://research.microsoft.com/en-us/um/people/sumitg/flashextract.html) research work in [Microsoft Research](http://research.microsoft.com).

To get started, consider a text-based address book:

    Ana Trujillo

    Redmond, WA

    Antonio Moreno

    Renton, WA

    Thomas Hardy

    Seattle, WA

    Christina Berglund

    Redmond, WA

    Hanna Moos

    Puyallup, WA

Copy a few examples into a file, which you will use as your template:

    Ana Trujillo

    Redmond, WA

    Antonio Moreno

    Renton, WA

Put curly braces around data that you want to extract, giving it a name as you do so. Because the **Name** property (and its associated other properties) can appear multiple times, use an asterisk (\*) to indicate that this results in multiple records (rather than extracting a bunch of properties into one record):

    {Name\*:Ana Trujillo}

    {City:Redmond}, {State:WA}

    {Name\*:Antonio Moreno}

    {City:Renton}, {State:WA}

From this set of examples, **ConvertFrom-String** can now automatically extract object-based output from input files with similar structure.

2 [C:\temp]

>> Get-Content .\addresses.output.txt | ConvertFrom-String -TemplateFile .\addresses.template.txt |  
>>> Format-Table -Auto

ExtentText                     Name               City     State  
----------                     ----               ----     -----  
Ana Trujillo...                Ana Trujillo       Redmond  WA  
Antonio Moreno...              Antonio Moreno     Renton   WA  
Thomas Hardy...                Thomas Hardy       Seattle  WA  
Christina Berglund...          Christina Berglund Redmond  WA  
Hanna Moos...                  Hanna Moos         Puyallup WA

To do additional data manipulation on extracted text, the **ExtentText** property captures the raw text from which the record was extracted. To provide feedback on this feature, or to share content for which you are having difficulty writing examples, please email [psdmfb@microsoft.com](mailto:psdmfb@microsoft.com).

## Powershell information stream

A new structured Information stream can now be used to transmit structured data between a script and its host. Write-Host has also been updated to emit its output to the Information stream where you can now capture or silence it. The new Write-Information cmdlet (along with a corresponding WriteInformation API) used with InformationVariable and InformationAction common parameters enables more flexibility and capability.

PS C:\Users\leeholm> ## Here's a typical PowerShell script  
PS C:\Users\leeholm> Show-ColorizedContent c:\temp\OutputGusher.ps1

001 | [CmdletBinding()]  
002 | param()  
003 |   
004 | Write-Host -ForegroundColor Green "Preparing to give you output!"  
005 | Write-Host "============================="  
006 | Write-Host "I " -NoNewLine -ForegroundColor White  
007 | Write-Host "<3" -ForegroundColor Red –NoNewLine  
008 | Write-Host " Output" -ForegroundColor White  
009 | Write-Host "============================="  
010 |   
011 | $p = Get-Process -id $pid  
012 | $p  
013 |   
014 | Write-Information $p -Tag Process  
015 | Write-Information "Some spammy logging information" -Tag LogLow  
016 | Write-Information "Some important logging information" -Tag LogHigh  
017 |   
018 | Write-Host  
019 | Write-Host -ForegroundColor Green "SCRIPT COMPLETE!One!Eleven!"  
020 | Write-Host "============================="

PS C:\Users\leeholm> ## And a typical problem  
PS C:\Users\leeholm> $r = c:\temp\OutputGusher  
Preparing to give you output!  
=============================  
I <3 Output  
=============================  
SCRIPT COMPLETE!One!Eleven!  
=============================  
PS C:\Users\leeholm>  
PS C:\Users\leeholm> ## Output still has some captured data  
PS C:\Users\leeholm> $r.Id  
4008  
PS C:\Users\leeholm>  
PS C:\Users\leeholm> ## With the -InformationVariable common variable, you can capture   
PS C:\Users\leeholm> ## the Write-Host data now.  
PS C:\Users\leeholm> $r = c:\temp\OutputGusher -InformationVariable iv  
Preparing to give you output!  
=============================  
I <3 Output  
=============================

SCRIPT COMPLETE!One!Eleven!

=============================

PS C:\Users\leeholm> $ivOutput = $iv | Group-Object -AsHash { $\_.Tags[0] } -AsString

PS C:\Users\leeholm> $ivOutput.PSHOST | Format-Table

MessageData

-----------

{[Message, Preparing to give you output!], [NoNewLine, False], [ForegroundColor, Green], [BackgroundColor, DarkMagenta]}

{[Message, =============================], [NoNewLine, False], [ForegroundColor, DarkYellow], [BackgroundColor, DarkMa...

{[Message, I ], [NoNewLine, True], [ForegroundColor, White], [BackgroundColor, DarkMagenta]}

{[Message, <3], [NoNewLine, True], [ForegroundColor, Red], [BackgroundColor, DarkMagenta]}

{[Message,  Output], [NoNewLine, False], [ForegroundColor, White], [BackgroundColor, DarkMagenta]}

{[Message, =============================], [NoNewLine, False], [ForegroundColor, DarkYellow], [BackgroundColor, DarkMa...

{[Message, ], [NoNewLine, False], [ForegroundColor, DarkYellow], [BackgroundColor, DarkMagenta]}

{[Message, SCRIPT COMPLETE!One!Eleven!], [NoNewLine, False], [ForegroundColor, Green], [BackgroundColor, DarkMagenta]}

{[Message, =============================], [NoNewLine, False], [ForegroundColor, DarkYellow], [BackgroundColor, DarkMa...

PS C:\Users\leeholm>

PS C:\Users\leeholm> ## And even ignore spammy output altogether

PS C:\Users\leeholm> $r = c:\temp\OutputGusher -InformationAction "SilentlyContinue"

PS C:\Users\leeholm>

PS C:\Users\leeholm> ## As we saw in lines 14..16 of the original script, though,

PS C:\Users\leeholm> ## the Write-Information cmdlet lets a script emit more than

PS C:\Users\leeholm> ## just one stream of output.  
PS C:\Users\leeholm> $ivOutput.LogHigh | % { Write-Warning $\_.MessageData }  
WARNING: Some important logging information  
PS C:\Users\leeholm>  
PS C:\Users\leeholm> ## All output includes useful properties that you would expect  
PS C:\Users\leeholm> ## of a generic event stream  
PS C:\Users\leeholm> $ivOutput.Process   
MessageData     : System.Diagnostics.Process (powershell)  
Source          : c:\temp\OutputGusher.ps1  
TimeGenerated   : 2/9/2015 5:08:52 PM  
Tags            : {Process}  
User            : leeholm  
Computer        : srv2  
ProcessId       : 4008  
NativeThreadId  : 7980  
ManagedThreadId : 8

PS C:\Users\leeholm> ## And of course, this works from hosting applications.  
PS C:\Users\leeholm> $ps = [PowerShell]::Create()  
PS C:\Users\leeholm> $ps.AddCommand('c:\temp\OutputGusher.ps1').Invoke()

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName  
-------  ------    -----      ----- -----   ------     -- -----------  
    559      41   119704     130968 ...46     7.77   4008 powershell

PS C:\Users\leeholm> $information = $ps.Streams.Information  
PS C:\Users\leeholm> $information | ? { $\_.Tags -contains 'LogLow'}

MessageData     : Some spammy logging information  
Source          : C:\temp\OutputGusher.ps1  
TimeGenerated   : 2/9/2015 5:08:53 PM  
Tags            : {LogLow}  
User            : leeholm  
Computer        : srv2  
ProcessId       : 4008  
NativeThreadId  : 2276  
ManagedThreadId : 14

## Develop DSC resources with classes in Windows PowerShell

We’ve improved the Windows PowerShell language for defining classes and other user-defined types. The goal is to enable developers and IT professionals to embrace Windows PowerShell for a wider range of use cases, simplify development of Windows PowerShell artifacts (such as DSC resources), and accelerate coverage of management surfaces.

### Supported scenarios in this release

* Define DSC resources and their associated types by using the Windows PowerShell language.
* Define custom types in Windows PowerShell by using familiar object-oriented programming constructs, such as classes, properties, methods, etc.
* Inheritance support with class in Windows PowerShell and class base DSC resource.
* Debug types by using the Windows PowerShell language.
* Generate and handle exceptions by using formal mechanisms, and at the right level.

### Define DSC resources with classes

Based on feedback, we’ve made authoring class-defined DSC resources simpler and easier to understand. The major differences between a class-defined DSC resource and a cmdlet DSC resource provider are:

* A MOF file for the schema is not required.
* A **DSCResource** subfolder in the module folder is not required.
* A Windows PowerShell module file can contain multiple DSC resource classes.

The following is an example of a class-defined DSC resource provider that extends the other class DSC resource in the same file. This is saved as a module, **MyDSCResource.psm1**. Note that you must always include at least one key property and Get, Set, Test method in a class-defined DSC resource or its base classes.

enum Ensure

{

Absent

Present

}

<#

This resource manages the file in a specific path.

[DscResource()] indicates the class is a DSC resource

#>

[DscResource()]

class BaseFileResource

{

<#

This property is the fully qualified path to the file that is expected to be present or absent.

The [DscProperty(Key)] attribute indicates the property is a key and its value uniquely identifies a resource instance. Defining this attribute also means the property is required and DSC will ensure a value is set before calling the resource.

A DSC resource must define at least one key property.

#>

[DscProperty(Key)]

[string]$Path

<#

This property indicates if the settings should be present or absent on the system.

For present, the resource ensures the file pointed to by $Path exists. For absent, it ensures the file that $Path points to does not exist.

The [DscProperty(Mandatory)] attribute indicates the property is required and DSC will guarantee it is set.

If Mandatory is not specified or if it is defined as Mandatory=$false, the value is not guaranteed to be set when DSC calls the resource. This is appropriate for optional properties.

#>

[DscProperty(Mandatory)]

[Ensure] $Ensure

<#

This property defines the fully qualified path to a file that will be placed on the system if $Ensure = Present and $Path does not exist.

NOTE: This property is required because [DscProperty(Mandatory)] is set.

#>

[DscProperty(Mandatory)]

[string] $SourcePath

<#

This property reports the file's creation timestamp.

[DscProperty(NotConfigurable))] attribute indicates the property is not configurable in a DSC configuration. Properties marked this way are populated by the Get() method to report additional details about the resource when it is present.

#>

[DscProperty(NotConfigurable)]

[Nullable[datetime]] $CreationTime

<#

This method is equivalent of the Set-TargetResource script function.

It sets the resource to the desired state.

#>

[void] Set()

{

$fileExists = $this.TestFilePath($this.Path)

if($this.ensure -eq [Ensure]::Present)

{

if(-not $fileExists)

{

$this.CopyFile()

}

}

else

{

if($fileExists)

{

Write-Verbose -Message "Deleting the file $($this.Path)"

Remove-Item -LiteralPath $this.Path -Force

}

}

}

<#

This method is equivalent of the Test-TargetResource script function.

It should return True or False, showing whether the resource

is in a desired state.

#>

[bool] Test()

{

$present = $this.TestFilePath($this.Path)

if($this.Ensure -eq [Ensure]::Present)

{

return $present

}

else

{

return -not $present

}

}

<#

This method is equal to the Get-TargetResource script function.

The implementation should use the keys to find appropriate resources.

This method returns an instance of this class with the updated key properties.

#>

[BaseFileResource] Get()

{

$present = $this.TestFilePath($this.Path)

if ($present)

{

$file = Get-ChildItem -LiteralPath $this.Path

$this.CreationTime = $file.CreationTime

$this.Ensure = [Ensure]::Present

}

else

{

$this.CreationTime = $null

$this.Ensure = [Ensure]::Absent

}

return $this

}

<#

Helper method to check if the file exists and it is the right file

#>

[bool] TestFilePath([string] $location)

{

$present = $true

$item = Get-ChildItem -LiteralPath $location -ea Ignore

if ($item -eq $null)

{

$present = $false

}

elseif( $item.PSProvider.Name -ne "FileSystem")

{

throw "Path $($location) is not a file path."

}

elseif($item.PSIsContainer)

{

throw "Path $($location) is a directory path."

}

return $present

}

<#

Helper method to copy the file from source to path

#>

[void] CopyFile()

{

if(-not $this.TestFilePath($this.SourcePath))

{

throw "SourcePath $($this.SourcePath) is not found."

}

[System.IO.FileInfo] $destFileInfo = new-object System.IO.FileInfo($this.Path)

if (-not $destFileInfo.Directory.Exists)

{

Write-Verbose -Message "Creating directory $($destFileInfo.Directory.FullName)"

#use CreateDirectory instead of New-Item to avoid code

# to handle the non-terminating error

[System.IO.Directory]::CreateDirectory($destFileInfo.Directory.FullName)

}

if(Test-Path -LiteralPath $this.Path -PathType Container)

{

throw "Path $($this.Path) is a directory path"

}

Write-Verbose -Message "Copying $($this.SourcePath) to $($this.Path)"

#DSC engine catches and reports any error that occurs

Copy-Item -LiteralPath $this.SourcePath -Destination $this.Path -Force

}

}

<#

This resource inherits from the [BaseFileResource]

It reports additional information in Get method

#>

[DscResource()]

class FileResource : BaseFileResource

{

<#

This property reports if it is a readonly file

#>

[DscProperty(NotConfigurable)]

[bool] $IsReadOnly

<#

This property reports the file's LastAccessTime timestamp.

#>

[DscProperty(NotConfigurable)]

[Nullable[datetime]] $LastAccessTime

<#

This property reports the file's LastWriteTime timestamp.

#>

[DscProperty(NotConfigurable)]

[Nullable[datetime]] $LastWriteTime

<#

This method overrides the Get method in the base class.

#>

[FileResource] Get()

{

$present = $this.TestFilePath($this.Path)

if ($present)

{

$file = Get-ChildItem -LiteralPath $this.Path

$this.CreationTime = $file.CreationTime

$this.IsReadOnly = $file.IsReadOnly

$this.LastAccessTime = $file.LastAccessTime

$this.LastWriteTime = $file.LastWriteTime

$this.Ensure = [Ensure]::Present

}

else

{

$this.CreationTime = $null

$this.LastAccessTime = $null

$this.LastWriteTime = $null

$this.Ensure = [Ensure]::Absent

}

return $this

}

}

After creating the class-defined DSC resource provider, and saving it as a module, create a module manifest for the module. In this example, the following module manifest is saved as **MyDscResource.psd1**.

@{

# Script module or binary module file associated with this manifest.

RootModule = 'MyDscResource.psm1'

# Version number of this module.

ModuleVersion = '1.0'

# ID used to identify this module uniquely

GUID = '81624038-5e71-40f8-8905-b1a87afe22d7'

# Author of this module

Author = 'User01'

# Company or vendor of this module

CompanyName = 'Unknown'

# Copyright statement for this module

Copyright = '(c) 2015 User01. All rights reserved.'

# Description of the functionality provided by this module

Description = 'DSC resource provider for FileResource.'

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

PowerShellVersion = '5.0'

# Name of the Windows PowerShell host required by this module

# PowerShellHostName = ''

# Required for DSC to detect PS class-based resources.

DscResourcesToExport = @('BaseFileResource','FileResource')

}

Deploy the new DSC resource provider by creating a **MyDscResource** folder for it under $env:SystemDrive\Program Files\WindowsPowerShell\Modules. You do not need to create a DSCResource subfolder. Copy the module and module manifest files (**MyDscResource.psm1** and **MyDscResource.psd1**) to the **MyDscResource** folder.

From this point, you create and run a configuration script as you would with any DSC resource. The following is a configuration that references the MyDSCResource module. Save this as a script, **MyResource.ps1**.

Configuration MyConfig

{

Import-Dscresource -ModuleName MyDscResource

BaseFileResource file

{

Path = "C:\test\baseFile.txt"

SourcePath = "c:\\test.txt"

Ensure = "Present"

}

FileResource file

{

Path = "C:\test\File.txt"

SourcePath = "c:\\test.txt"

Ensure = "Present"

}

}

MyConfig

Run this as you would any DSC configuration script. To start the configuration, in an elevated Windows PowerShell console, run the following cmdlet. You will see the output of Get-DscConfiguration from FileResource contains more information than BaseFileResource.

PS C:\test> .\MyResource.ps1

PS C:\test> Start-DscConfiguration c:\test\MyConfig –wait –verbose

PS C:\test> Get-DscConfiguration

### Known issues

In this release, the following are known issues with class-defined DSC resource providers.

* Complex types are not supported in a class-defined resource.
* Custom types are not available outside the module yet.
* A module containing both class-defined and script resources is not supported.
* Classes cannot yet define composite resources.
* Remove-Module does not unload a module that contains a class, which can cause a failure to reload the class-defined DSC resource when the LCM is set to ForceModuleImport debug mode.

## Inheritance with classes in Windows PowerShell

### Declare base classes for Windows PowerShell classes

You can declare a Windows PowerShell class as a base type for another Windows PowerShell class.

class bar

{

[int]foo() {return 100500}

}

class baz : bar {}

[baz]::new().foo() # return 100500

You can also use existing .NET Framework types as base classes.

class MyIntList : system.collections.generic.list[int]

{

}

$list = [MyIntList]::new()

$list.Add(100)

$list[0] # return 100

### Declare implemented interfaces for Windows PowerShell classes

You can declare implemented interfaces after base types, or immediately after a colon (:), if there is no base type specified. Separate all type names by using commas. It’s very similar to C# syntax.

class MyComparable : system.IComparable

{

[int] CompareTo([object] $obj)

{

return 0;

}

}

class MyComparableBar : bar, system.IComparable

{

[int] CompareTo([object] $obj)

{

return 0;

}

}

### Call base class constructor

To call a base class constructor from a subclass, use the keyword **base**.

class A {

[int]$a

A([int]$a)

{

$this.a = $a

}

}

class B : A

{

B() : base(103) {}

}

[B]::new().a # return 103

If a base class has a default (no parameter) constructor, you can omit an explicit constructor call.

class C : B

{

C([int]$c) {}

}

### Call base class method

You can override existing methods in subclasses. To do this, declare methods by using the same name and signature.

class baseClass

{

[int]foo() {return 100500}

}

class childClass1 : baseClass

{

[int]foo() {return 200600}

}

[childClass1]::new().foo() # return 200600

To call base class methods from overridden implementations, cast to the base class ([baseclass]$this) on invocation.

class childClass2 : baseClass

{

[int]foo()

{

return 3 \* ([baseClass]$this).foo()

}

}

[childClass2]::new().foo() # return 301500

All Windows PowerShell methods are virtual. You can hide non-virtual .NET methods in a subclass by using the same syntax as you do for an override: just declare methods with same name and signature.

class MyIntList : system.collections.generic.list[int]

{

# Add is final in system.collections.generic.list

[void] Add([int]$arg)

{

([system.collections.generic.list[int]]$this).Add($arg \* 2)

}

}

$list = [MyIntList]::new()

$list.Add(100)

$list[0] # return 200

## DSC resource script debugging

WMF 5.0 Production Preview includes support for debugging DSC resource scripts while they run on target nodes. In older WMF 5.0 releases, we added advanced script debugging features with the ability to attach to local processes (Get-PSHostProcessInfo, Enter-PSHostProcessInfo, Exit-PSHostProcessInfo), enumerate all runspaces in a process, and debug an arbitrary runspace in a process (Get-Runspace, Debug-Runspace).

DSC resource script debugging builds on this work by adding two new cmdlets.

Enable-DscDebug

SYNTAX

Enable-DscDebug [-BreakAll] [-CimSession <CimSession[]>] [-ThrottleLimit <int>] [-AsJob] [-WhatIf] [-Confirm] [<CommonParameters>]

Disable-DscDebug

SYNTAX

Disable-DscDebug [-CimSession <CimSession[]>] [-ThrottleLimit <int>] [-AsJob] [-WhatIf] [-Confirm] [<CommonParameters>]

.Enable-DscDebug cmdlet with BreakAll configures the DSC LCM to run Get-TargetResource, Set-TargetResource, and Test-TargetResource scripts in Break All mode, meaning that the script stops at the first script statement and waits for a debugger to be attached. You can then connect to the target computer by using Windows PowerShell remoting, and attach the Windows PowerShell debugger to the LCM process and runspace to debug the script. After you do this, you can set breakpoints and debug the script in the normal Windows PowerShell way.

A typical DSC resource script debugging session goes as follows:

PS C:\Test> Enable-DscDebug –BreakAll

PS C:\Test> Start-DscConfiguration -path .\TestConfig2 -Wait -Verbose

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, ''namespaceName' = root/Microsoft/Windows/DesiredStateConfiguration,'className' = MSFT\_DSCLocalConfigurationManager,'methodName' = SendConfigurationApply'.

VERBOSE: An LCM method call arrived from computer MGMT-10001-827 with user sid S-1-5-21-397955417-626881126-188441444-3860663.

VERBOSE: [MGMT-10001-827]: LCM: [ Start Set ]

WARNING: [MGMT-10001-827]: [DSCEngine] Warning LCM is in Debug 'ResourceScriptBreakAll' mode. Resource script processing will be stopped to wait for PowerShell script debugger to attach.

VERBOSE: [MGMT-10001-827]: [DSCEngine] Importing the module C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\PSDesiredStateConfiguration\DscResources\MSFT\_EnvironmentResource\MSFT\_EnvironmentResource.psm1 i

n force mode.

VERBOSE: [MGMT-10001-827]: LCM: [ Start Resource ] [[Environment]e1]

VERBOSE: [MGMT-10001-827]: LCM: [ Start Test ] [[Environment]e1]

VERBOSE: [MGMT-10001-827]: [[Environment]e1] Importing the module MSFT\_EnvironmentResource in force mode.

WARNING: [MGMT-10001-827]: [[Environment]e1] Resource is waiting for PowerShell script debugger to attach. Use the following commands to begin debugging this resource script:

Enter-PSSession -ComputerName MGMT-10001-827 -Credential <credentials>

Enter-PSHostProcess -Id 2640 -AppDomainName DscPsPluginWkr\_AppDomain

Debug-Runspace -Id 3

PS C:\Test> Disable-DscDebug

By running the commands shown in the warning message after running Start-DscConfiguration, you attach your client Windows PowerShell script debugger to the appropriate computer, process, application domain, and runspace.

# Run this command to optionally use Windows PowerShell remoting to connect to the target computer. Skip this if you already have a remote desktop connection.

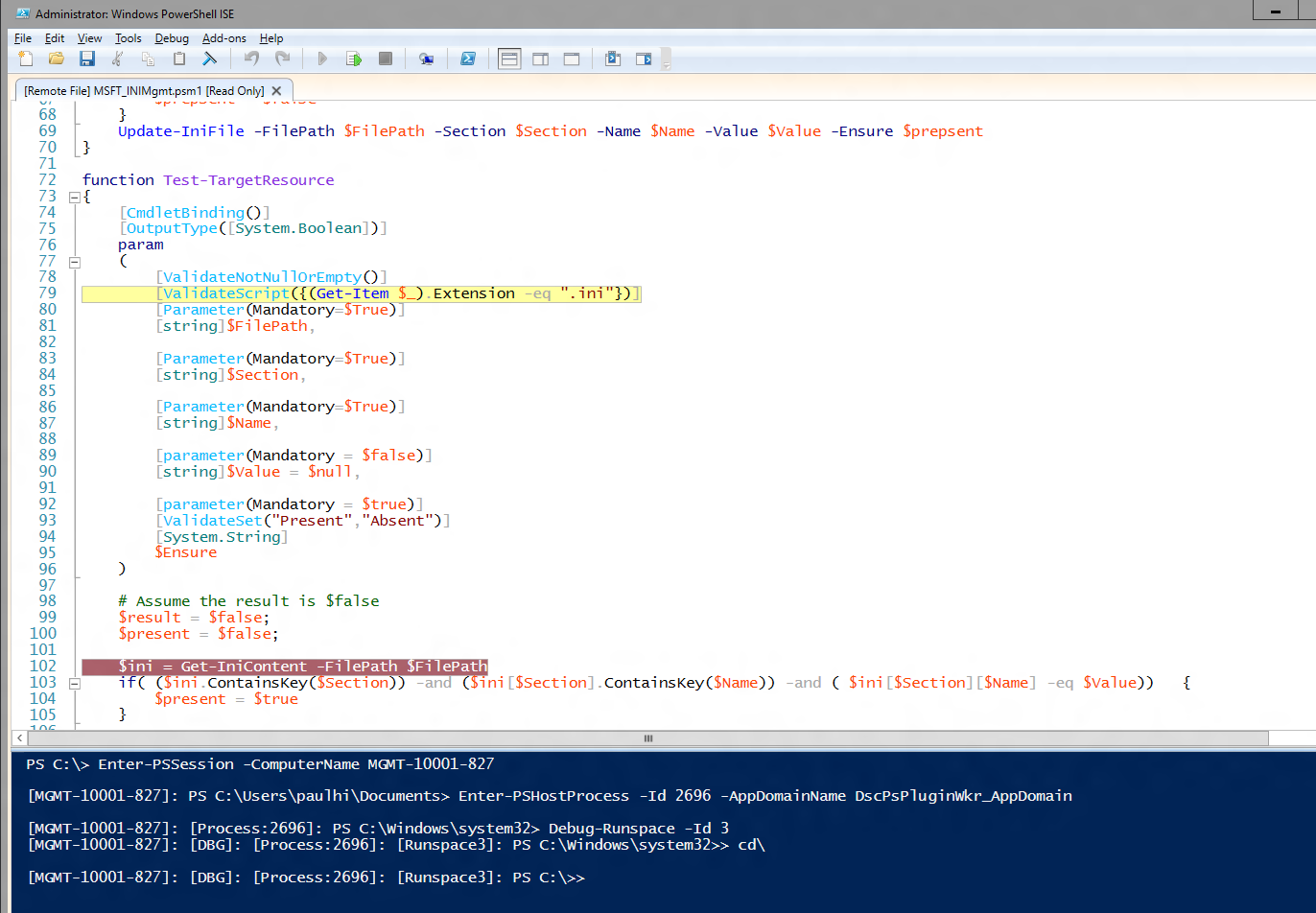
Enter-PSSession -ComputerName MGMT-10001-827

# This command attaches to the DSC LCM host process and app domain where the resource script is running.

Enter-PSHostProcess -Id 2640 -AppDomainName DscPsPluginWkr\_AppDomain

# This last command lets you debug the DSC runspace running the script.

Debug-Runspace -Id 3



After you have the debugger attached, set line breakpoints where you want to investigate script execution and run the **continue** debugger command to let the script run in the debugger. When you are done debugging, you can stop the script from running by typing the **quit** debugger command, or let the script continue to run without the debugger by running the **detach** debugger command.

Notice that ***all*** resource scripts are stopped in the debugger. This means that Test-TargetResoruce, Set-TargetResource, and Get-TargetResource scripts each in turn stop in the debugger. If you don't want to debug a resource script, you can exit debug mode in the runspace by running this command.

Disable-RunspaceDebug -RunspaceId 3

Or you can attach the debugger by running Debug-Runspace, and then immediately run the **detach** command.

After you are done debugging your resource script, you should stop the DSC configuration by running the following command.

Stop-DscConfiguration –Force

Finally, you must reconfigure the target computer LCM to disable debug mode using Disable-DscDebug cmdlet..

## Support for new RefreshMode value

This release introduces a new RefreshMode value, **Disabled**. When this mode is set, LCM does not do document management, and any third party can invoke DSC resources directly by running the Invoke-DscResource cmdlet.

Configuration LCMSettings

{

    Node localhost

    {

        LocalConfigurationManager

        {

           RefreshMode = 'Disabled'

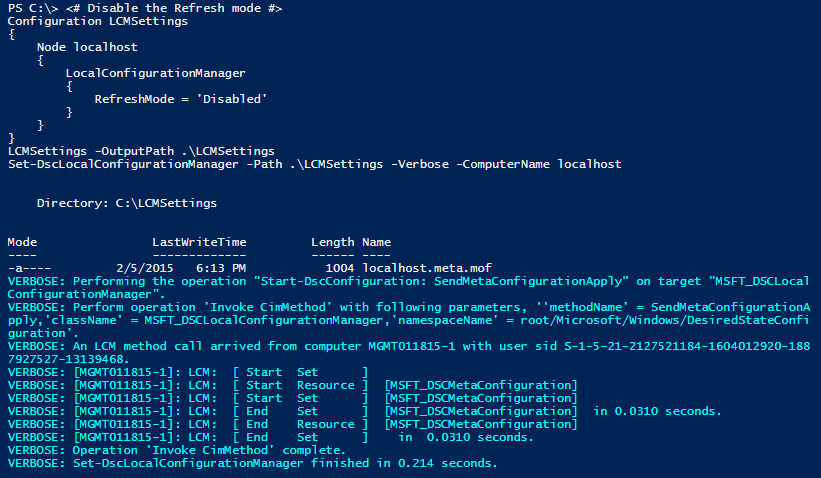
        }

    }

}

LCMSettings -OutputPath .\LCMSettings

Set-DscLocalConfigurationManager -Path .\LCMSettings -Verbose -ComputerName localhost



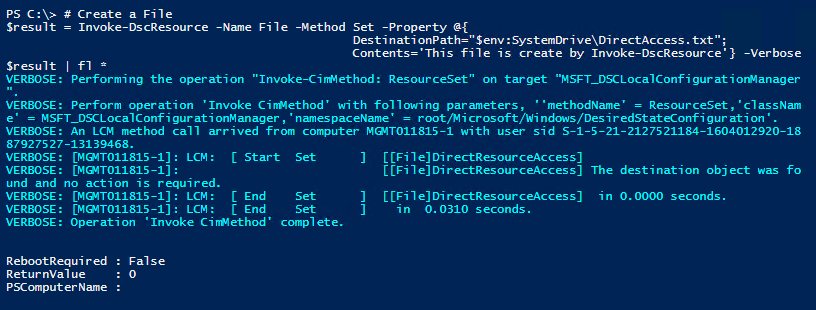
# Create a File

$result = Invoke-DscResource -Name File -Method Set -Property @{

                                DestinationPath="$env:SystemDrive\DirectAccess.txt";

                                Contents='This file is create by Invoke-DscResource'} -Verbose

$result | fl \*



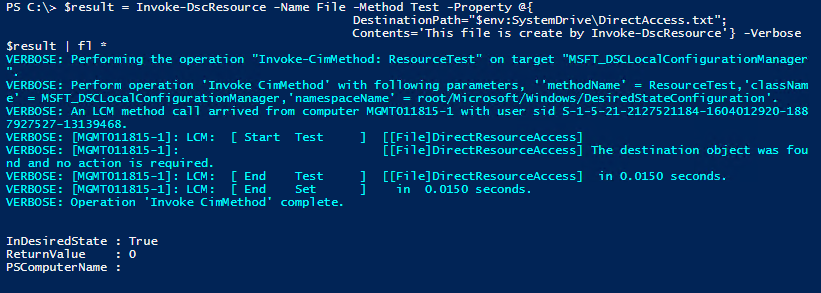
# Test the resource state

$result = Invoke-DscResource -Name File -Method Test -Property @{

                                 DestinationPath="$env:SystemDrive\DirectAccess.txt";

                                 Contents='This file is create by Invoke-DscResource'} -Verbose

$result | fl \*



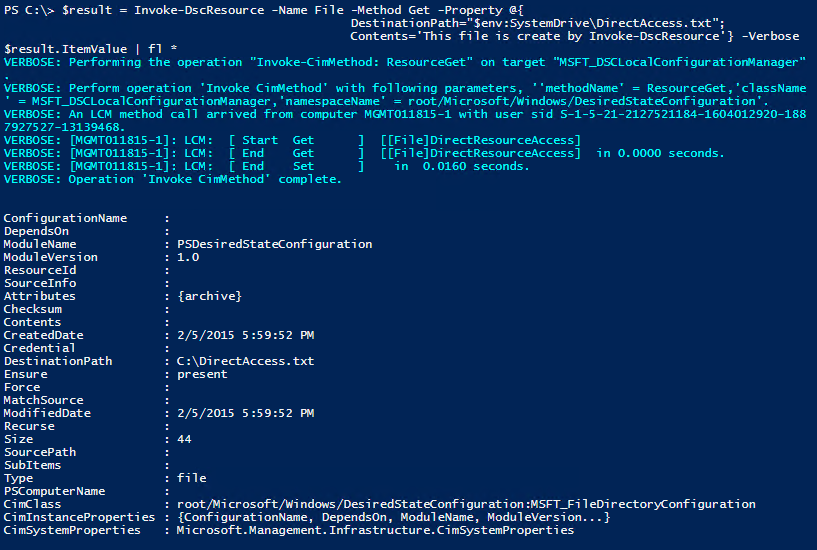
# Get the contents of File

$result = Invoke-DscResource -Name File -Method Get -Property @{

                                DestinationPath="$env:SystemDrive\DirectAccess.txt";

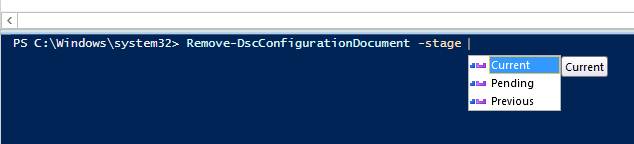
                                Contents='This file is create by Invoke-DscResource'} -Verbose

$result.ItemValue | fl \*



## Remove DSC documents delivered to a system

When a configuration document is delivered to DSC, the document goes through various stages (pending, current, previous). We added a new cmdlet to DSC in Windows PowerShell 4.0, Remove-DscConfigurationDocument, as part of KB3000850. This cmdlet was also available in WMF 5.0 Preview February 2015. Remove-DscConfigurationDocument doesn’t have a **Partial** state in WMF 5.0 Preview April 2015.



## Partial configurations support mixed RefreshModes

When you use partial configurations, you can now define each individual partial configuration with a RefreshMode. Valid values are **PUSH** or **PULL**.

[DscLocalConfigurationManager()]

Configuration partialMeta

{

Settings

{

RefreshMode = "PULL"

RefreshFrequencyMins = 30

ConfigurationModeFrequencyMins = 15

ConfigurationMode = "ApplyAndAutoCorrect"

RebootNodeIfNeeded = $false

ConfigurationId = "922A3987-6647-4665-B770-0FD008436CE3"

}

PartialConfiguration Partial1

{

RefreshMode = "PUSH"

Description = "Partial1"

}

PartialConfiguration Partial2

{

RefreshMode = "PULL"

Description = "Partial1"

ConfigurationSource = "[ConfigurationRepositoryShare]FileShare"

DependsOn = "[PartialConfiguration]Partial1"

}

PartialConfiguration Partial3

{

RefreshMode = "PUSH"

Description = "Partial3"

ConfigurationSource = "[ConfigurationRepositoryWeb]PullServerWeb1"

DependsOn = "[PartialConfiguration]Partial2"

}

ConfigurationRepositoryShare FileShare

{

SourcePath = "\\testserver\pullserver"

}

}

## Generate Windows PowerShell cmdlets based on an OData endpoint

Export-ODataEndpointProxy is a cmdlet that generates a set of Windows PowerShell cmdlets based on the functionality exposed by a given OData endpoint.

The following example shows how to use this new cmdlet:

# Basic use case of Export-ODataEndpoint Proxy

Export-ODataEndpointProxy -Uri 'http://services.odata.org/v3/(S(snyobsk1hhutkb2yulwldgf1))/odata/odata.svc' -OutputModule C:\Users\user\Generated.psd1

ipmo 'C:\Users\user\Generated.psd1'

# Cmdlets are created based on the following heuristics

# New-<EntityType> -<Key> [-<Other Attributes>…]

#

# Get-<EntityType> [-<Key> -Top –Skip –Filter -OrderBy]

# # If there is a complex key, the keys will actually be -<Key1> -<Key2>…

# # Note this rule applies to any other instances of the key

#

# Set-<EntityType> -<Key> [-<Other Attributes>]

#

# Remove-<EntityType> -<Key>

#

# Invoke-<EntityType><Action> [-<Key> -<Other Parameters>]

#

#

# Cmdlets from associations (Note: Get and Remove get additional parameter sets)

# Get-<EntityType> -<AssociatedEntity>

# New-<EntityType> -<AssociatedEntity> -<Key>

# Remove-<EntityType> -<AssociatedEntity> -<Key>

#

#

# Note: Every cmdlet has a –ConnectionURI parameter for explicitly setting the URI of the endpoint. This normally uses the same address that you gave the Export-ODataEndpointProxy cmdlet, but can be overridden in this fashion for the sake of similar endpoints.

#

There are still parts of key use cases in development for this functionality, including, but not limited to:

* Associations
* Passing streams

## Configure DSC’s Local Configuration Manager with the meta-configuration attribute

The **DscLocalConfigurationManager** attribute designates a configuration block as a meta-configuration, which is used to configure the DSC Local Configuration Manager. This attribute restricts a configuration to containing only items which configure the DSC Local Configuration Manager. During processing, this configuration generates a \*.meta.mof file that is then sent to the appropriate target nodes by using the Set-DscLocalConfigurationManager cmdlet.

[DscLocalConfigurationManager()]

configuration meta

{

Node localhost

{

Settings

{

ConfigurationMode = "ApplyAndAutocorrect"

ConfigurationID = "5603f952-d6c6-4971-88c4-948636bf5c3f"

RefreshMode = "Pull"

}

ConfigurationRepositoryWeb PullServer

{

ServerURL = "https://corp.contoso.com/PSDSCPullServer/PSDSCPullServer.svc"

}

}

}

The above example configures the refresh mode for LCM to be in pull mode, changes the configuration mode to ApplyAndAutocorrect, and defines the type and location of the pull server.

This new configuration replaces and extends the functionality of the LocalConfigurationManager resource from DSC v1. LocalConfigurationManager is still supported in configurations without this attribute, for backwards compatibility.

Meta-resources:

|  |  |
| --- | --- |
| **Resource Name** | **Description** |
| LocalConfigurationManager | Various settings for DSC engine execution |
| PartialConfiguration | Partial configuration settings |
| ConfigurationRepositoryWeb | Web-based configuration repository |
| ConfigurationRepositoryShare | File share-based configuration repository |
| ResourceRepositoryWeb | Web-based resource repository |
| ResourceRepositoryShare | File-based resource repository |
| ReportServerWeb | Web-based reporting endpoint for pull scenario |

## Configure piece by piece with partial configurations in DSC

WMF 5.0 Production Preview helps you deliver configuration documents to a node in fragments. For a node to receive multiple fragments of a configuration document, its Local Configuration Manager must be set first to specify the expected fragments, as shown in this example.

[DSCLocalConfigurationManager()]

configuration SQLServerDSCSettings

{

Node localhost

{

Settings

{

ConfigurationModeFrequencyMins = 30

}

ConfigurationRepositoryWeb OSConfigServer

{

ServerURL = "https://corp.contoso.com/OSConfigServer/PSDSCPullServer.svc"

}

ConfigurationRepositoryWeb SQLConfigServer

{

ServerURL = "https://corp.contoso.com/SQLConfigServer/PSDSCPullServer.svc"

}

PartialConfiguration OSConfig

{

Description = 'Configuration for the Base OS'

ExclusiveResources = 'PSDesiredStateConfiguration\\*'

ConfigurationSource = '[ConfigurationRepositoryWeb]OSConfigServer'

}

PartialConfiguration SQLConfig

{

Description = 'Configuration for the SQL Server'

ConfigurationSource = '[ConfigurationRepositoryWeb]SQLConfigServer'

DependsOn = '[PartialConfiguration]OSConfig'

}

}

}

In a partial configuration, the configuration name must match what is defined in the Local Configuration Manager. In the above example, the configurations should be named **OSConfig** and **SQLConfig**.

Setting Local Configuration Manager for partial configurations enables configuration coordination, but does NOT provide security functionality.

## Manage with cross-computer dependencies in DSC

By using the built-in WaitFor\* resources (WaitForAll, WaitForAny, and WaitForSome), you can now specify dependencies across computers during configuration runs, without external orchestrations. These resources provide node-to-node synchronization by using CIM connections over the WS-Man protocol. By using these resources, a configuration can wait for another computer’s specific resource state to change.

For example, in the following configuration, the target node is waiting for the **xADDomain** resource to finish on the **MyDC** node with a few retries, before the target node can join the domain.

configuration JoinDomain

{

Import-DscResource -Module xComputerManagement

WaitForAll DC

{

ResourceName = '[xADDomain]NewDomain'

NodeName = 'MyDC'

RetryIntervalSec = 15

RetryCount = 30

}

xComputer JoinDomain

{

Name = 'MyPC'

DomainName = 'Contoso.com'

Credential = (get-credential)

DependsOn ='[WaitForAll]DC'

}

}

## Find more detail about configuration status in DSC

### Get-DscConfigurationStatus

The Get-DscConfigurationStatus cmdlet gets high-level information about configuration status from a target node. You can obtain the status of the last or all configurations.

Get-DscConfigurationStatus -All [-CimSession <CimSession[]>] [-ThrottleLimit <int>] [-AsJob] [<CommonParameters>]

Get-DscConfigurationStatus [-CimSession <CimSession[]>] [-ThrottleLimit <int>] [-AsJob] [<CommonParameters>]

### Compare-DscConfiguration

The Compare-DscConfiguration cmdlet compares a specified configuration with the actual state of one or more target nodes.

Compare-DscConfiguration [[-ReferenceConfiguration] <string>] [[-ComputerName] <string[]>] [-AsJob] [-Credential <pscredential>] [-ThrottleLimit <int>] [<CommonParameters>]

Compare-DscConfiguration [[-ReferenceConfiguration] <string>] -CimSession <CimSession[]> [-AsJob] [-throttleLimit <int>] [<CommonParameters>]

## More control over configurations in DSC

### Publish-DscConfiguration

The Publish-DscConfiguration cmdlet copies a configuration MOF file to a target node, but does not apply the configuration. This configuration is applied during the next consistency pass, or when you run the Update-DscConfiguration cmdlet.

Publish-DscConfiguration [-Path] <string> [[-ComputerName] <string[]>] [-Force] [-Credential <pscredential>]

[-ThrottleLimit <int>] [-WhatIf] [-Confirm] [<CommonParameters>]

Publish-DscConfiguration [-Path] <string> -CimSession <CimSession[]> [-Force] [-ThrottleLimit <int>] [-WhatIf]

[-Confirm] [<CommonParameters>]

### Update-DscConfiguration

The Update-DscConfiguration cmdlet forces the configuration to be processed. If the LCM is in pull mode, it gets the configuration from the pull server before applying it.

Update-DscConfiguration [[-ComputerName] <string[]>] [-Wait] [-Force] [-JobName <string>] [-Credential

<pscredential>] [-ThrottleLimit <int>] [-WhatIf] [-Confirm] [<CommonParameters>]

Update-DscConfiguration -CimSession <CimSession[]> [-Wait] [-Force] [-JobName <string>] [-ThrottleLimit <int>]

[-WhatIf] [-Confirm] [<CommonParameters>]

## Configuring the Windows PowerShell Desired State Configuration service

To set up a server as a Windows PowerShell Desired State Configuration pull server, additional steps are needed:

**On Windows Server 2008 R2 SP1:**

* Add-WindowsFeature Web-Server
* Add-WindowsFeature Web-Http-Tracing
* dism /online /enable-feature:ManagementOdata
* Add-WindowsFeature NET-Http-Activation
* Add-WindowsFeature Web-ASP-NET
* dism /online /enable-feature:DSC-Service

After installing DSC-Service, you must create a pull server IIS endpoint. Follow the steps in section “**4. Create the pull server” in the** pull server MSDN documentation (<http://technet.microsoft.com/library/dn249913.aspx> ) to get the most current xDSCService resource for installing a pull server endpoint from the [TechNet Gallery](http://gallery.technet.microsoft.com/xPSDesiredStateConfiguratio-417dc71d). You can read more about the pull server endpoint installation resource in the blog post, [PowerShell DSC Resource for Configuring Pull Server Environment](http://blogs.msdn.com/b/powershell/archive/2013/11/21/powershell-dsc-resource-for-configuring-pull-server-environment.aspx).

**On Windows Server 2008 R2 and Windows Server 2012:**

Run **aspnet\_regiis.exe –i** in the following folders:

* $env:SystemRoot\Microsoft.NET\Framework64\v4…
* $env:SystemRoot\Microsoft.NET\Framework\v4…

## More control and remoting in Windows PowerShell debugging

### Break All

The Windows PowerShell console and Windows PowerShell ISE now allow you to break into the debugger for running scripts. This works in both local and remote sessions.

In the console, press **Ctrl+Break**.

In ISE, press **Ctrl+B**, or use the **Debug->Break All** menu command.

### Remote debugging and remote file editing in Windows PowerShell ISE

Windows PowerShell ISE now lets you open and edit files in a remote session by running the PSEdit command.

For example, you can open a file for editing from the command line in a remote session as follows:

[RemoteComputer1]: PS C:\> PSEdit C:\DebugDemoScripts\Test-GetMutex.ps1

In addition, you can now edit and save changes in a remote file that is automatically opened in Windows PowerShell ISE when you hit a breakpoint.

Now, you can debug a script file that is running on a remote computer, edit the file to fix an error, and then rerun the modified script.

### Advanced script debugging

There are new, advanced debugging features that let you attach to any local computer process that has loaded Windows PowerShell, and debug arbitrary runspaces in that process.

#### Runspace debugging

New cmdlets have been added that let you list current runspaces in a process, and attach the Windows PowerShell console or ISE debugger to that runspace for script debugging.

* Get-Runspace
* Debug-Runspace
* Enable-RunspaceDebug
* Disable-RunspaceDebug
* Get-RunspaceDebug

#### Attach to process

You can now attach to any computer process that has Windows PowerShell loaded. You do this by entering into an interactive session with the process, similarly to how you enter into an interactive remote session by running the Enter-PSSession cmdlet.

* Enter-PSHostProcess
* Exit-PSHostProcess

## Rich Information For DSC LCM State

We have made improvements in exposing details about the LCM state. The LCMState that is returned by Get-DscLocalConfigurationManager can now contain the following values:

* **Idle**
* **Busy**
* **PendingReboot**
* **PendingConfiguration**

We have also added an LCMStateDetail property that contains more information about the state.

## Support for -? during DSC configuration compilation

WMF 5.0 Preview April 2015 includes help support for DSC configurations.

Example

<#

.SYNOPSIS

The synopsis for the configuration goes here. This can be one line, or many.

.DESCRIPTION

The description for the configuration is usually a longer, more detailed explanation of what the script or function does. Use as many lines as you need.

.PARAMETER computername

Here, the dotted keyword is followed by a single parameter name. Don't precede that with a hyphen. The following lines describe the purpose of the parameter:

.PARAMETER filePath

Provide a PARAMETER section for each parameter that your script or function accepts.

.EXAMPLE

There's no need to number your examples.

.EXAMPLE

Windows PowerShell will number them for you when it displays your help text to a user.

#>

configuration HelpSample1

{

    param([string]$computername,[string]$filePath)

    File f

    {

Contents="Hello World"

        DestinationPath = "c:\Destination.txt"

    }

}

<#

.SYNOPSIS

The synopsis for the configuration HelpSample2 goes here. This can be one line, or many.

.DESCRIPTION

The description for the configuration bar is usually a longer, more detailed explanation of what the script or function does. Take as many lines as you need.

.PARAMETER computername

Here, the dotted keyword is followed by a single parameter name. Don't precede that with a hyphen. The following lines describe the purpose of the parameter:

.PARAMETER filePath

Provide a PARAMETER section for each parameter that your script or function accepts.

.EXAMPLE

There's no need to number your examples.

.EXAMPLE

Windows PowerShell will number them for you when it displays your help text to a user.

#>

configuration HelpSample2

{

    param([string]$computername,[string]$filePath)

    File f

    {

Contents="Hello World"

        DestinationPath = "c:\Destination.txt"

    }

}

PS C:\Windows\system32> HelpSample1 -?

NAME

HelpSample1

SYNOPSIS

The synopsis for the configuration goes here. This can be one line, or many.

SYNTAX

HelpSample1 [[-InstanceName] <String>] [[-DependsOn] <String[]>] [[-OutputPath] <String>] [[-ConfigurationData] <Hashtable>] [[-computername] <String>] [[-filePath] <String>] [<CommonParameters>]

DESCRIPTION

The description for the configuration is usually a longer, more detailed explanation of what the script or function does. Take as many lines as you need.

RELATED LINKS

REMARKS

To see the examples, type: "get-help HelpSample1 -examples".

For more information, type: "get-help HelpSample1 -detailed".

For technical information, type: "get-help HelpSample1 -full".

PS C:\Windows\system32> get-help HelpSample2 -detailed

NAME

HelpSample2

SYNOPSIS

The synopsis for the configuration HelpSample2 goes here. This can be one line, or many.

SYNTAX

HelpSample2 [[-InstanceName] <String>] [[-DependsOn] <String[]>] [[-OutputPath] <String>] [[-ConfigurationData] <Hashtable>] [[-computername] <String>] [[-filePath] <String>] [<CommonParameters>]

DESCRIPTION

The description for the configuration bar is usually a longer, more detailed explanation of what the script or function does. Take as many lines as you need.

PARAMETERS

-InstanceName <String>

-DependsOn <String[]>

-OutputPath <String>

-ConfigurationData <Hashtable>

-computername <String>

Here, the dotted keyword is followed by a single parameter name. Don't precede that with a hyphen. The following lines describe the purpose of the parameter:

-filePath <String>

Provide a PARAMETER section for each parameter that your script or function accepts.

<CommonParameters>

This cmdlet supports the common parameters: Verbose, Debug,

ErrorAction, ErrorVariable, WarningAction, WarningVariable,

OutBuffer, PipelineVariable, and OutVariable. For more information, see

about\_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

-------------------------- EXAMPLE 1 --------------------------

PS C:\>There's no need to number your examples.

-------------------------- EXAMPLE 2 --------------------------

PS C:\>Windows PowerShell will number them for you when it displays your help text to a user.

REMARKS

To see the examples, type: "get-help HelpSample2 -examples".

For more information, type: "get-help HelpSample2 -detailed".

For technical information, type: "get-help HelpSample2 -full".

PS C:\Windows\system32> HelpSample1 -debug

DEBUG: MSFT\_FileDirectoryConfiguration: RESOURCE PROCESSING STARTED [KeywordName='File'] Function='PSDesiredStateConfiguration\File']

DEBUG: MSFT\_FileDirectoryConfiguration: ResourceID = [File]f

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'DependsOn' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'DependsOn' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'PsDscRunAsCredential' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'PsDscRunAsCredential' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'DestinationPath' [

DEBUG: MSFT\_FileDirectoryConfiguration: Canonicalized property 'DestinationPath' = 'c:\Destination.txt'

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'DestinationPath' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'Ensure' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'Ensure' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'Type' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'Type' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'SourcePath' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'SourcePath' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'Contents' [

DEBUG: MSFT\_FileDirectoryConfiguration: Canonicalized property 'Contents' = 'Hello World'

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'Contents' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'Checksum' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'Checksum' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'Recurse' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'Recurse' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'Force' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'Force' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'Credential' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'Credential' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'Attributes' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'Attributes' ]

DEBUG: MSFT\_FileDirectoryConfiguration: Processing property 'MatchSource' [

DEBUG: MSFT\_FileDirectoryConfiguration: Processing completed 'MatchSource' ]

DEBUG: MSFT\_FileDirectoryConfiguration: MOF alias for this resource is '$MSFT\_FileDirectoryConfiguration1ref'

DEBUG: MSFT\_FileDirectoryConfiguration: RESOURCE PROCESSING COMPLETED. TOTAL ERROR COUNT: 0

Directory: C:\Windows\system32\HelpSample1

Mode LastWriteTime Length Name

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

-a---- 4/16/2015 3:30 PM 1940 localhost.mof

WARNING: The configuration 'HelpSample1' is loading one or more built-in resources without explicitly importing associated modules. Add Import-DscResource –ModuleName ’PSDesiredStateConfiguration’ to your configuration to avoid this

message.

## Support for DSC RunAs credential

WMF 5.0 Preview April 2015 includes support for running **any** DSC resource under a specified set of credentials by using the property PsDscRunAsCredential. It enables scenarios like installing MSI packages in a specific user context, accessing a user’s registry hive, accessing a user’s specific local directory, accessing a network share, etc.

The following shows an example of using the PsDscRunAsCredential property in DSC to change the background color of the command prompt of a user.

Configuration ChangeCmdBackGroundColor

{

    Node ("localhost")

    {

        Registry CmdPath

        {

            Key = "HKEY\_CURRENT\_USER\\Software\Microsoft\\Command Processor"

            ValueName = "DefaultColor"

            ValueData = '1F'

            ValueType = "DWORD"

            Ensure = "Present"

            Force = $true

            Hex = $true

            PsDscRunAsCredential = get-credential

        }

    }

}

$configData = @{

AllNodes = @(

@{

NodeName="localhost";

CertificateFile = "C:\publicKeys\targetNode.cer"

}

)

}

ChangeCmdBackGroundColor -ConfigurationData $configData

### Known issues

In this release, the following are known issues of the DSC RunAs credential feature:

* The WindowsProcess resource doesn’t support RunAs credential.

## Side-by-side installation of DSC resources and PowerShell Modules

Modules can including modules that contain DSC resources can now be installed side-by-side, and configurations can use a specific version of the resource that is installed on the system.

### Known issues

In this release, the following are known issues of side-by-side installation:

* Installing side-by-side modules of class-based DSC resources is not supported.
* Using two different versions of the DSC resource within the same configuration is not supported.

## Separation of node and configuration IDs

DSC currently uses a configuration ID to uniquely identify a single configuration for a single node. This feature separates the configuration ID into two distinct identifiers: Configuration Name and Agent ID. Configuration Name identifies the configuration for a computer; this ID can be shared by multiple nodes. Agent ID uniquely identifies a node; this ID must be unique for every node.

### Use case

Users need the flexibility to deploy the same configuration across multiple nodes, while still tracking status and reporting information for each node individually. This lets users create configurations for roles rather than nodes.

One assumption here (validated by your feedback) is that users do not want to set up unique identifiers for each target node. They do not want to do any bookkeeping with GUIDs. There was a user-defined need to set up each target node with a different identifier. We believe that most users have some automation or workflow to set this up. Our solution should make it easy for you to work with target nodes and the corresponding configuration documents.

### MetaConfig updates for separation of computer and configuration IDs

Because the ConfigurationNames are no longer GUIDs (they are now friendly names), anyone can determine them. To mitigate this issue, we added an extra level of security by adding a registration step before a node can start requesting configurations from a server. A node registers itself with the pull server with a shared secret (which the node and the server both know already), and the name of the configuration it will request. This shared secret need not be unique for each computer. Assumption: the shared secret is a hard-to-guess identifier, like a GUID. We call this shared secret RegistrationKey in the metaconfig.

DscLocalConfigurationManager()]

Configuration SampleMetaConfig

{

Settings

{

RefreshFrequencyMins = 30;

RefreshMode = "PULL";

ConfigurationMode =”ApplyAndMonitor“;

AllowModuleOverwrite = $true;

RebootNodeIfNeeded = $true;

ConfigurationModeFrequencyMins = 60;

}

ConfigurationRepositoryWeb ConfigurationManager

{

ServerURL = “https://PullServerMachine:8080/psdscpullserver.svc”

RegistrationKey = "140a952b-b9d6-406b-b416-e0f759c9c0e4"

ConfigurationNames = @(“WebRole”)

}

}

SampleMetaConfig

The RegistrationKey value is already known to the server. While setting up the pull server, you can set up a text file with the name **RegistrationKeys.txt** in a specific location, and then set this location in the web.config of the pull server, as shown below. The highlighted line shows the functionality in WMF 5.0 Preview April 2015. The rest of the settings were made available in earlier WMF releases.

<add key="ConfigurationPath" value="C:\Program Files\WindowsPowerShell\DscService\Configuration" />

<add key="ModulePath" value="C:\Program Files\WindowsPowerShell\DscService\Modules" />

<add key="RegistrationKeyPath" value="C:\Program Files\WindowsPowerShell\DscService" />

Now, the node is set up to pull the configuration with name WebRole.mof from the PullServer.

This feature is not supported when the pull server is set up to be a file share. It is only supported for the web-based pull server.

### Known Issues

Currently, this feature is not supported for pulling partial configurations. To pull partial configurations, you must fall back to the older protocol. To do this, specify ConfigurationId in the metaconfig.

## PSDesiredStateConfiguration module version updated to 1.1

With WMF 5.0 Preview April 2015, the module version of PsDesiredStateConfiguration is 1.1. This change only affects WMF 5.0 Preview Feb 2015. DSC configurations compiled on WMF 5.0 Preview Feb 2015 and applied to WMF 5.0 Preview April 2015 will not work. Similarly, DSC configurations compiled on WMF 5.0 Preview April 2015 and applied to WMF 5.0 Preview Feb 2015 will not work.

## Software Inventory Logging

Software Inventory Logging helps reduce the operational costs of getting accurate information about the Microsoft software installed locally on a server, but especially across many servers in an IT environment (assuming the software is installed and running across the IT environment). Provided one is set up, you can forward this data to an aggregation server, and collect the log data in one place by using a uniform, automatic process.

While you can also log software inventory data by querying each computer directly, Software Inventory Logging, by employing a forwarding (over the network) architecture initiated by each server, can overcome server discovery challenges that are typical for many software inventory and asset management scenarios. Software Inventory Logging uses SSL to secure data that is forwarded over HTTPS to an aggregation server. Storing the data in one place makes the data easier to analyze, manipulate, and share when necessary.

None of this data is sent to Microsoft as part of the feature functionality. Software Inventory Logging data and functionality is meant for the sole use of the server software’s licensed owner and administrators.

For more information and documentation about Software Inventory Logging cmdlets, see Windows Server 2012 R2 online resources at <http://technet.microsoft.com/library/dn383584.aspx>.

### Known Issues

In WMF 5.0 Production Preview , a known bug prevents you from installing this feature on Windows 7 and Windows Server 2008 R2 platforms.

## Discover, install, and inventory software with PACKAGEMANAGEMENt

PackageManagement (aka OneGet) is a new way to discover and install software packages from around the web. It is a manager or multiplexor of existing package managers (also called package providers) that unifies Windows package management with a single Windows PowerShell interface. With PackageManagement, you can do the following.

* Manage a list of software repositories in which packages can be searched, acquired, and installed
* Discover software packages you need
* Seamlessly install, uninstall, and inventory packages from one or more software repositories

### Install package providers and bootstrapPER

PackageManagement includes a set of providers by default and a bootstrapper provider that enables on-demand installation of other package providers. Available package providers can be discovered by using the command:

PS C:\> Get-PackageProvider

You can install on-demand by running any PackageManagement command that has a -Provider parameter, and specifying the package provider name as the value of the -Provider parameter. If the package provider isn't currently installed, the bootstrap provider tries to download and install it. Add -Force or -ForceBootstrap to skip the prompts.

### Manage software sources

Each PackageManagement provider may have one or multiple software sources, or repositories. PackageManagement provides PowerShell cmdlets to add/remove/query the source. For example, you can add a new chocolatey source by using the command:

PS C:\> Register-PackageSource -ProviderName NuGet -Name Nuget.Org -Location https://nuget.org/api/v2

You can use command **Unregister-PackageSource**, to disconnect with a repository, and **Get-PackageSource**, to discover what the repositories are associated with that provider.

## Software Discover, install, and inventory with PackageManagement

This is the core of PackageManagement to support software discovery, installation, and inventory (SDII). Try out the cmdlets for these operations:

* Find-Package
* Install-Package
* Get-Package
* Save-Package
* Uninstall-Package

PackageManagement in this release has more improvements:

* Huge performance improvements
* Tagging is supported by the NuGet provider. This can be leveraged by any providers that use NuGet, including PowerShellGet.
* Improvement of the default package providers, including Add Remove Program (Programs), MSI, and MSU.

### Use additional package providers from the Internet

With this release, PackageManagement can now bootstrap additional package providers from the Internet. This implementation downloads managed providers to one of two locations:

* If the user is running Windows PowerShell with elevated user rights: **$env:PROGRAMFILES\PackageManagement\ProviderAssemblies**
* If the user is not running with elevated user rights: **$env:LOCALAPPDATA\PackageManagement\ProviderAssemblies**

Be careful when you install providers from the Internet.

### Additional information

The source for the PackageManagement component can be found in the GitHub OneGet project website, <https://oneget.org>.

## Discover modules and DSC resources with PowerShellGet

PowerShellGet is included in the Production Preview release of WMF:

* Find-Module can filter on module metadata with the -Tag parameter
* Find-Module can filter on repository-specific search language with the -Filter parameter
* Find-Module can filter based on module contents with the -Command, -DscResource, and -Includes parameters
* Find-DscResource allows discovery of individual DSC resources in repositories
* Support for installing from and publishing to file shares with NuGet

### Example commands

# Find all modules with tags Tag1 or Tag3

Find-Module -Tag Tag1,Tag3

# Find modules with a specific DscResource

Find-Module -DscResource DscTestResource

#Find modules with specific commands

Find-Module -Command Test-PSGetTestCmdlet, Get-ContosoServer, Get-ContosoClient

# Find all modules with Dsc resources

Find-Module -Includes DscResource

# Find all modules with cmdlets

Find-Module -Includes Cmdlet

# Find all modules with functions

Find-Module -Includes Function

# Find all DSC resources

Find-DscResource

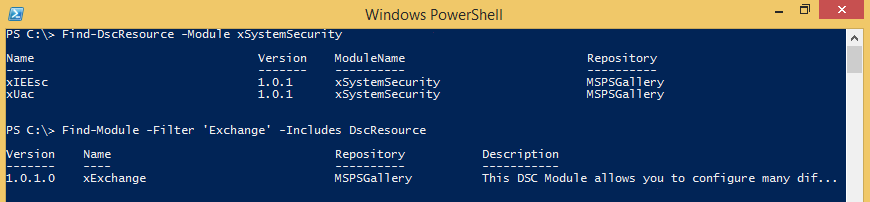
# Find all DSC resources contained within a specific module

Find-DscResource -ModuleName DscTestModule

# Find all DSC resources in modules with Tag1 or Tag3

Find-DscResource -Tag Tag1,Tag3

### Example output



### Register a PSRepository with PowerShellGet

You can configure PowerShellGet to operate against internal repositories. This is done by using the following additions:

* Register-PSRepository: Registers a repository for the current user.
* Unregister-PSRepository: Removes a registered repository for the current user.
* Set-PSRepository: Set values for a registered repository.
* Get-PSRepository: Get all registered repositories for the current user.

After a repository is registered, you can use Find-Module and Install-Module to work with it.

#Register a default repository

Register-PSRepository –Name DemoRepo –SourceLocation “https://www.myget.org/F/powershellgetdemo/api/v2” –PublishLocation “<https://www.myget.org/F/powershellgetdemo/api/v2>/package” –InstallationPolicy –Trusted

#Get all of the registered repositories

Get-PSRepository

Name SourceLocation

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

PSGallery https://msconfiggallery.cloudapp...

DemoRepo https://www.myget.org/F/powershe...

#Search only the new repository for modules

Find-Module -Repository DemoRepo

Repository Version Name

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

DemoRepo 1.0.1 xActiveDirectory

DemoRepo 1.1.1 SomeModule

#By default, PowerShellGet operates against all registered repositories when none is specified. In this example, the “SomeModule” module is installed from the DemoRepo.

Install-Module SomeModule

#Removing a repository

Unregister-PSRepository DemoRepo

## Improvements to the PowerShellGet Module

### New features in PowerShellGet

* Side-by-side version support on Windows PowerShell 5.0 or newer
* Module dependency installation support
* Three new cmdlets
  + Get-InstalledModule
  + Uninstall-Module
  + Save-Module

### Side-by-side module version support is enabled in Windows PowerShell 5.0 or newer.

There is now side-by-side (SxS) module version support in Install-Module, Update-Module, and Publish-Module cmdlets that run in Windows PowerShell 5.0 or newer.

Also, we have added a -RequiredVersion parameter to the Publish-Module cmdlet to specify the version to be published. The Path parameter now supports the module base path with the version folder.

# Install-Module examples

PS C:\windows\system32> Install-Module -Name ContosoServer -RequiredVersion 1.0 -Repository MSPSGallery

PS C:\windows\system32> Get-Module -ListAvailable -Name ContosoServer | Format-List Name,Version,ModuleBase

Name : ContosoServer

Version : 1.0

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\1.0

PS C:\windows\system32> Install-Module -Name ContosoServer -RequiredVersion 2.0 -Repository MSPSGallery

PS C:\windows\system32> Get-Module -ListAvailable -Name ContosoServer | Format-List Name,Version,ModuleBase

Name : ContosoServer

Version : 2.0

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\2.0

Name : ContosoServer

Version : 1.0

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\1.0

PS C:\windows\system32> Get-InstalledModule

Version Name Repository Description

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

1.0 ContosoServer MSPSGallery ContosoServer module

2.0 ContosoServer MSPSGallery ContosoServer module

# Update-Module examples

PS C:\windows\system32> Update-Module -Name ContosoServer -RequiredVersion 1.5

PS C:\windows\system32> Get-Module -ListAvailable -Name ContosoServer | Format-List Name,Version,ModuleBase

Name : ContosoServer

Version : 2.0

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\2.0

Name : ContosoServer

Version : 1.5

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\1.5

Name : ContosoServer

Version : 1.0

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\1.0

PS C:\windows\system32> Get-InstalledModule

Version Name Repository Description

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

1.0 ContosoServer MSPSGallery ContosoServer module

1.5 ContosoServer MSPSGallery ContosoServer module

2.0 ContosoServer MSPSGallery ContosoServer module

PS C:\windows\system32> Update-Module -Name ContosoServer

PS C:\windows\system32> Get-Module -ListAvailable -Name ContosoServer | Format-List Name,Version,ModuleBase

Name : ContosoServer

Version : 2.8.1

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\2.8.1

Name : ContosoServer

Version : 2.0

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\2.0

Name : ContosoServer

Version : 1.5

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\1.5

Name : ContosoServer

Version : 1.0

ModuleBase : C:\Program Files\WindowsPowerShell\Modules\ContosoServer\1.0

PS C:\windows\system32> Get-InstalledModule

Version Name Repository Description

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

1.0 ContosoServer MSPSGallery ContosoServer module

1.5 ContosoServer MSPSGallery ContosoServer module

2.0 ContosoServer MSPSGallery ContosoServer module

2.8.1 ContosoServer MSPSGallery ContosoServer module

# Publish-Module examples

# ContosoServer module with different versions to be published

PS C:\windows\system32> Get-Module -Name ContosoServer -ListAvailable

Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version Name ExportedCommands

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

Manifest 2.8 ContosoServer Get-ContosoServer

Manifest 2.0 ContosoServer Get-ContosoServer

Manifest 1.5 ContosoServer Get-ContosoServer

Manifest 1.0 ContosoServer Get-ContosoServer

PS C:\windows\system32> Publish-Module -Name ContosoServer -RequiredVersion 1.0 -Repository LocalRepo -NuGetApiKey Local-Repo-NuGet-ApiKey

PS C:\windows\system32> Find-Module -Name ContosoServer -Repository LocalRepo

Version Name Repository Description

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

1.0 ContosoServer LocalRepo ContosoServer module

PS C:\windows\system32> Publish-Module -Name ContosoServer -RequiredVersion 1.5 -Repository LocalRepo -NuGetApiKey Local-Repo-NuGet-ApiKey

PS C:\windows\system32> Find-Module -Name ContosoServer -Repository LocalRepo

Version Name Repository Description

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

1.0 ContosoServer LocalRepo ContosoServer module

1.5 ContosoServer LocalRepo ContosoServer module

PS C:\windows\system32> Publish-Module -Path "C:\Program Files\WindowsPowerShell\Modules\ContosoServer\2.0" -Repository LocalRepo -NuGetApiKey Local-Repo-NuGet-ApiKey

PS C:\windows\system32> Find-Module -Name ContosoServer -Repository LocalRepo

Version Name Repository Description

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

1.0 ContosoServer LocalRepo ContosoServer module

1.5 ContosoServer LocalRepo ContosoServer module

2.0 ContosoServer LocalRepo ContosoServer module

### Module dependency installation support, Get-InstalledModule and Uninstall-Module cmdlets

* Added module dependencies population in the Publish-Module cmdlet. The RequiredModules and NestedModules lists of PSModuleInfo are used in preparing the dependency list of a module to be published.
* Added dependency installation support in the Install-Module and Update-Module cmdlets. Module dependencies are installed and updated by default.
* Added an -IncludeDependencies parameter to the Find-Module cmdlet to include module dependencies in the results.
* Added -MaximumVersion support on the Find-Module, Install-Module, and Update-Module cmdlets.
* Added new Get-InstalledModule and Uninstall-Module cmdlets.

PowerShellGet cmdlets demo with module dependencies support:

1. Ensure that module dependencies are available on the repository.

PS C:\windows\system32> Find-Module -Repository LocalRepo -Name RequiredModule1,RequiredModule2,RequiredModule3,NestedRequiredModule1,NestedRequiredModule2,NestedRequiredModule3 | Sort-Object -Property Name

Version    Name                     Repository    Description

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

2.5        NestedRequiredModule1    LocalRepo     NestedRequiredModule1 module

2.5        NestedRequiredModule2    LocalRepo     NestedRequiredModule2 module

2.0        NestedRequiredModule3    LocalRepo     NestedRequiredModule3 module

2.5        RequiredModule1          LocalRepo     RequiredModule1 module

2.5        RequiredModule2          LocalRepo     RequiredModule2 module

2.0        RequiredModule3          LocalRepo     RequiredModule3 module

1. Create a module with dependencies that are specified in the RequiredModules and NestedModules properties of its module manifest.

$RequiredModules = @('RequiredModule1',

                      @{ModuleName = 'RequiredModule2'; ModuleVersion = '1.5'; },

                      @{ModuleName = 'RequiredModule3'; RequiredVersion = '2.0'; })

$NestedRequiredModules = @('TestDepWithNestedRequiredModules1.psm1', 'NestedRequiredModule1',

                            @{ModuleName = 'NestedRequiredModule2'; ModuleVersion = '1.5'; },

                            @{ModuleName = 'NestedRequiredModule3'; RequiredVersion = '2.0'; })

New-ModuleManifest -Path 'C:\Program Files\WindowsPowerShell\Modules\TestDepWithNestedRequiredModules1\TestDepWithNestedRequiredModules1.psd1' `

                   -NestedModules $NestedRequiredModules `

                   -RequiredModules $RequiredModules `

                   -ModuleVersion "1.0" `

                         -Description "TestDepWithNestedRequiredModules1 module"

1. Publish two versions (**“1.0”** and **“2.0”**) of the TestDepWithNestedRequiredModules1 module with dependencies to the repository.

Publish-Module -Name TestDepWithNestedRequiredModules1 -Repository LocalRepo -NuGetApiKey "MyNuGet-ApiKey-For-LocalRepo"

1. Find the TestDepWithNestedRequiredModules1 module with its dependencies by specifying -IncludeDependencies.

PS C:\windows\system32> Find-Module -Name TestDepWithNestedRequiredModules1 -Repository LocalRepo –IncludeDependencies -MaximumVersion "1.0"

Version    Name                                Repository  Description

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

1.0        TestDepWithNestedRequiredModules1   LocalRepo   TestDepWithNestedRequiredModules1 module

2.5        RequiredModule1                     LocalRepo   RequiredModule1 module

2.5        RequiredModule2                     LocalRepo   RequiredModule2 module

2.0        RequiredModule3                     LocalRepo   RequiredModule3 module

2.5        NestedRequiredModule1               LocalRepo   NestedRequiredModule1 module

2.5        NestedRequiredModule2               LocalRepo   NestedRequiredModule2 module

2.0        NestedRequiredModule3               LocalRepo   NestedRequiredModule3 module

1. Use Find-Module metadata to find the module dependencies.

PS C:\windows\system32> $psgetModuleInfo = Find-Module -Repository MSPSGallery -Name ModuleWithDependencies2

PS C:\windows\system32> $psgetModuleInfo.Dependencies.ModuleName

RequiredModule1

RequiredModule2

RequiredModule3

NestedRequiredModule1

NestedRequiredModule2

NestedRequiredModule3

PS C:\windows\system32> $psgetModuleInfo.Dependencies

Name Value

---- -----

ModuleName RequiredModule1

CanonicalId psmodule:RequiredModule1/#http://psget/psGallery/api/v2/

ModuleName RequiredModule2

ModuleVersion 2.0

CanonicalId psmodule:RequiredModule2/2.0#http://psget/psGallery/api/v2/

ModuleName RequiredModule3

RequiredVersion 2.5

CanonicalId psmodule:RequiredModule3/[2.5]#http://psget/psGallery/api/v2/

ModuleName NestedRequiredModule1

CanonicalId psmodule:NestedRequiredModule1/#http://psget/psGallery/api/v2/

ModuleName NestedRequiredModule2

ModuleVersion 2.0

CanonicalId psmodule:NestedRequiredModule2/2.0#http://psget/psGallery/api/v2/

ModuleName NestedRequiredModule3

RequiredVersion 2.5

CanonicalId psmodule:NestedRequiredModule3/[2.5]#http://psget/psGallery/api/v2/

1. Install the TestDepWithNestedRequiredModules1 module with dependencies.

PS C:\windows\system32> Install-Module -Name TestDepWithNestedRequiredModules1 -Repository LocalRepo -RequiredVersion "1.0"

PS C:\windows\system32> Get-InstalledModule

Version    Name                   Repository   Description

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

1.0        NestedRequiredModule1  LocalRepo    NestedRequiredModule1 module

2.5        NestedRequiredModule2  LocalRepo    NestedRequiredModule2 module

2.0        NestedRequiredModule3  LocalRepo    NestedRequiredModule3 module

1.0        RequiredModule1        LocalRepo    RequiredModule1 module

2.5        RequiredModule2                    LocalRepo    RequiredModule2 module

2.0        RequiredModule3                    LocalRepo    RequiredModule3 module

1.0        TestDepWithNestedRequiredModules1  LocalRepo    TestDepWithNestedRequiredModules1 module

1. Update the TestDepWithNestedRequiredModules1 module with dependencies.

PS C:\windows\system32> Find-Module -Name TestDepWithNestedRequiredModules1 -Repository LocalRepo -AllVersions

Version    Name                                Repository  Description

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

1.0        TestDepWithNestedRequiredModules1   LocalRepo   TestDepWithNestedRequiredModules1 module

2.0        TestDepWithNestedRequiredModules1   LocalRepo   TestDepWithNestedRequiredModules1 module

PS C:\windows\system32> Update-Module -Name TestDepWithNestedRequiredModules1 -RequiredVersion 2.0

PS C:\windows\system32> Get-InstalledModule

Version    Name                                Repository  Description

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

1.0        NestedRequiredModule1               LocalRepo   NestedRequiredModule1 module

2.5        NestedRequiredModule2               LocalRepo   NestedRequiredModule2 module

2.0        NestedRequiredModule3               LocalRepo   NestedRequiredModule3 module

2.5        NestedRequiredModule3               LocalRepo   NestedRequiredModule3 module

1.0        RequiredModule1                     LocalRepo   RequiredModule1 module

2.5        RequiredModule2                     LocalRepo   RequiredModule2 module

2.0        RequiredModule3                     LocalRepo   RequiredModule3 module

2.5        RequiredModule3                     LocalRepo   RequiredModule3 module

1.0        TestDepWithNestedRequiredModules1   LocalRepo   TestDepWithNestedRequiredModules1 module

2.0        TestDepWithNestedRequiredModules1   LocalRepo   TestDepWithNestedRequiredModules1 module

1. Run the Uninstall-Module cmdlet to uninstall a module that you installed by using PowerShellGet. If any other module depends on the module that you want to delete, PowerShellGet throws an error.

PS C:\windows\system32> Get-InstalledModule -Name RequiredModule1 | Uninstall-Module

PackageManagement\Uninstall-Package : The module 'RequiredModule1' of version '2.5' in module base folder 'C:\Program

Files\WindowsPowerShell\Modules\RequiredModule1\2.5' cannot be uninstalled, because one or more other modules 'ModuleWithDependencies2' are

dependent on this module. Uninstall the modules that depend on this module before uninstalling module 'RequiredModule1'.

At C:\Program Files\WindowsPowerShell\Modules\PowerShellGet\PSGet.psm1:1303 char:25

+ ... $null = PackageManagement\Uninstall-Package @PSBoundParameters

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidOperation: (Microsoft.Power...ninstallPackage:UninstallPackage) [Uninstall-Package], Exception

+ FullyQualifiedErrorId : UnableToUninstallAsOtherModulesNeedThisModule,Uninstall-Package,Microsoft.PowerShell.PackageManagement.Cmdlets.Uninsta

llPackage

### Save-Module cmdlet

PS C:\windows\system32> Save-Module -Repository MSPSGallery -Name ModuleWithDependencies2 -Path C:\MySavedModuleLocation

PS C:\windows\system32> dir C:\MySavedModuleLocation

Directory: C:\MySavedModuleLocation

Mode LastWriteTime Length Name

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

d----- 4/21/2015 5:40 PM ModuleWithDependencies2

d----- 4/21/2015 5:40 PM NestedRequiredModule1

d----- 4/21/2015 5:40 PM NestedRequiredModule2

d----- 4/21/2015 5:40 PM NestedRequiredModule3

d----- 4/21/2015 5:40 PM RequiredModule1

d----- 4/21/2015 5:40 PM RequiredModule2

d----- 4/21/2015 5:40 PM RequiredModule3

### Updated syntax of all PowerShellGet cmdlets

PS C:\windows\system32> Get-Command -Module PowerShellGet -Syntax

Find-Module [[-Name] <string[]>] [-MinimumVersion <version>] [-MaximumVersion <version>] [-RequiredVersion <version>] [-AllVersions]

[-IncludeDependencies] [-Filter <string>] [-Tag <string[]>] [-Includes <string[]>] [-DscResource <string[]>] [-Command <string[]>] [-Repository

<string[]>] [<CommonParameters>]

Get-InstalledModule [[-Name] <string[]>] [-MinimumVersion <version>] [-RequiredVersion <version>] [-MaximumVersion <version>] [<CommonParameters>]

Get-PSRepository [[-Name] <string[]>] [<CommonParameters>]

Install-Module [-Name] <string[]> [-MinimumVersion <version>] [-MaximumVersion <version>] [-RequiredVersion <version>] [-Repository <string[]>]

[-Scope <string>] [-Force] [-WhatIf] [-Confirm] [<CommonParameters>]

Install-Module [-InputObject] <psobject[]> [-Scope <string>] [-Force] [-WhatIf] [-Confirm] [<CommonParameters>]

Publish-Module -Name <string> -NuGetApiKey <string> [-RequiredVersion <version>] [-Repository <string>] [-FormatVersion <version>] [-ReleaseNotes

<string[]>] [-Tags <string[]>] [-LicenseUri <uri>] [-IconUri <uri>] [-ProjectUri <uri>] [-WhatIf] [-Confirm] [<CommonParameters>]

Publish-Module -Path <string> -NuGetApiKey <string> [-Repository <string>] [-FormatVersion <version>] [-ReleaseNotes <string[]>] [-Tags <string[]>]

[-LicenseUri <uri>] [-IconUri <uri>] [-ProjectUri <uri>] [-WhatIf] [-Confirm] [<CommonParameters>]

Register-PSRepository -Name <string> -SourceLocation <uri> [-PublishLocation <uri>] [-InstallationPolicy <string>] [-PackageManagementProvider

<string>] [<CommonParameters>]

Save-Module [-Name] <string[]> -Path <string> [-MinimumVersion <version>] [-MaximumVersion <version>] [-RequiredVersion <version>] [-Repository

<string[]>] [-Force] [-WhatIf] [-Confirm] [<CommonParameters>]

Save-Module [-Name] <string[]> -LiteralPath <string> [-MinimumVersion <version>] [-MaximumVersion <version>] [-RequiredVersion <version>]

[-Repository <string[]>] [-Force] [-WhatIf] [-Confirm] [<CommonParameters>]

Save-Module [-InputObject] <psobject[]> -LiteralPath <string> [-Force] [-WhatIf] [-Confirm] [<CommonParameters>]

Save-Module [-InputObject] <psobject[]> -Path <string> [-Force] [-WhatIf] [-Confirm] [<CommonParameters>]

Set-PSRepository -Name <string> [-SourceLocation <uri>] [-PublishLocation <uri>] [-InstallationPolicy <string>] [-PackageManagementProvider

<string>] [<CommonParameters>]

Uninstall-Module [-Name] <string[]> [-MinimumVersion <version>] [-RequiredVersion <version>] [-MaximumVersion <version>] [-Force] [-WhatIf]

[-Confirm] [<CommonParameters>]

Uninstall-Module [-InputObject] <psobject[]> [-Force] [-WhatIf] [-Confirm] [<CommonParameters>]

Unregister-PSRepository [-Name] <string[]> [<CommonParameters>]

Update-Module [[-Name] <string[]>] [-RequiredVersion <version>] [-MaximumVersion <version>] [-Force] [-WhatIf] [-Confirm] [<CommonParameters>]

## Report configuration status from DSC to central location

High-level information about DSC configuration status (Success or Failure) can be sent to a server running the DSC service. By defining the report server in a nodes meta-configuration, this status is sent to the server when an error occurs or the configuration completes successfully. This information is sent when a node is configured in either pull or push mode. Status information is stored in the DSC service database.

A new OData endpoint is created with the DSC service which exposes this status information. By passing a configuration ID to the endpoint, all of the status for a node can be collected and parsed.

#### Sample meta-configuration for reporting status

[DscLocalConfigurationManager()]

Configuration ReportingClientMetaConfig  
{  
    Settings  
        {  
            ConfigurationID = "$guid"  
            RefreshFrequencyMins = 30  
            RefreshMode = "PUSH"  
            ConfigurationModeFrequencyMins = 15  
            AllowModuleOverwrite = $true  
        }

        ReportServerWeb ReportManager  
        {  
            ServerUrL = "<http://localhost:8080/PSDSCPullServer/PSDSCPullserver.svc>"  
            AllowUnsecureConnection  = $true  
        }             
}

#### Sample web request to gather configuration status

$statusReports = Invoke-WebRequest -Uri "[http://localhost:8080/PSDSCPullserver/PSDSCPullserver.svc/Node(ConfigurationId='$guid')/StatusReport](http://localhost:8080/PSDSCPullserver/psdscpullserver.svc/Node(ConfigurationId='$guid')/StatusReport)s" -UseBasicParsing -UseDefaultCredentials -ContentType "application/json;odata=minimalmetadata;streaming=true;charset=utf-8" -Headers @{Accept = "application/json"; ProtocolVersion = “1.1”}

## Develop with classes in Windows PowerShell

WMF 5.0 Preview September 2014 added support to the Windows PowerShell language for defining classes and other user-defined types, by using formal syntax and semantics that are similar to other object-oriented programming languages. The goal is to enable developers and IT professionals to embrace Windows PowerShell for a wider range of use cases, simplify development of Windows PowerShell artifacts (such as DSC resources), and accelerate coverage of management surfaces.

### Define custom types in Windows PowerShell

WMF 5.0 Preview September 2014 introduces the following new language elements in Windows PowerShell.

| Element | Description |
| --- | --- |
| **Class** keyword | Defines a new class. This is a true .NET Framework type.  Class members are public, but only public within the module scope. You can't refer to the type name as a string (for example, New-Object doesn't work), and in this release, you can't use a type literal (for example, [MyClass]) outside the script/module file in which the class is defined.  class MyClass  {  } |
| **Enum** keyword and enumerations | Support for the **enum** keyword has been added, which uses newline as the delimiter.  Current limitations: you cannot define an enumerator in terms of itself, but you can initialize enum in terms of another enum, as shown in the following example. The base type cannot currently be specified; it is always [int].  enum Color2  {  Yellow = [Color]::Blue  }  An enumerator value must be a parse time constant; you cannot set it to the result of an invoked command.  enum MyEnum  {  Enum1  Enum2  Enum3 = 42  Enum4 = [int]::MaxValue  }  Enums support arithmetic operations, as shown in the following example.  enum SomeEnum { Max = 42 }  enum OtherEnum { Max = [SomeEnum]::Max + 1 } |
| Import-DscResource | Import-DscResource is now a true dynamic keyword. Windows PowerShell parses the specified module’s root module, searching for classes that contain the **DscResource** attribute. |
| ImplementingAssembly | A new field, **ImplementingAssembly**, has been added to ModuleInfo. It is set to the dynamic assembly created for a script module if the script defines classes, or the loaded assembly for binary modules. It is not set when ModuleType = Manifest.  Reflection on the **ImplementingAssembly** field discovers resources in a module. This means you can discover resources written in either PowerShell or other managed languages.  Fields with initializers:  [int] $i = 5  Static is supported; it works like an attribute, as do the type constraints, so it can be specified in any order.  static [int] $count = 0  A type is optional.  $s = "hello"  All members are public. |
| Constructors and instantiation | Windows PowerShell classes can have constructors; they have the same name as their class. Constructors can be overloaded. Static constructors are supported. Properties with initialization expressions are initialized before running any code in a constructor. Static properties are initialized before the body of a static constructor, and instance properties are initialized before the body of the non-static constructor. Currently, there is no syntax for calling a constructor from another constructor (like the C# syntax ": this()"). The workaround is to define a common Init method.  The following are ways of instantiating classes in this release.  # Instantiating by using the default constructor. Note that New-Object is not supported in this release.  $a = [MyClass]::new()  # Calling a constructor with a parameter  $b = [MyClass]::new(42)  # Passing an array to a constructor with multiple parameters  $c = [MyClass]::new(@(42,43,44), "Hello")  In this release, New-Object does not work with classes defined in Windows PowerShell. Also for this release, the type name is only visible lexically, meaning it is not visible outside of the module or script that defines the class. Functions can return instances of a class defined in Windows PowerShell, and instances work well outside of the module or script.  Get-Member -Static lists constructors, so you can view overloads like any other method. The performance of this syntax is also considerably faster than New-Object.  The pseudo-static method named **new** works with .Net types, as shown in the following example.  [hashtable]::new()  You can now see constructor overloads with Get-Member, or as shown in this example:  PS> [hashtable]::new  OverloadDefinitions  -------------------  hashtable new()  hashtable new(int capacity)  hashtable new(int capacity, float loadFactor) |
| Methods | A Windows PowerShell class method is implemented as a ScriptBlock that has only an end block. All methods are public. The following shows an example of defining a method named **DoSomething**.  class MyClass  {  DoSomething($x)  {  $this.\_doSomething($x) # method syntax  }  private \_doSomething($a) {}  }  Method invocation:  $b = [MyClass]::new()  $b.DoSomething(42)  Overloaded methods—that is, those that are named the same as an existing method, but differentiated by their specified values—are also supported. |
| Properties | All properties are public. Properties require either a newline or semicolon. If no object type is specified, the property type is object.  Properties that use validation attributes or argument transformation attributes (e.g. [ValidateSet("aaa")]) work as expected. |
| Hidden | A new keyword, **Hidden**, has been added. **Hidden** can be applied to properties and methods (including constructors).  Hidden members are public, but do not appear in the output of Get-Member unless the -Force parameter is added.  Hidden members are not included when tab completing or using Intellisense unless the completion occurs in the class defining the hidden member.  A new attribute, System.Management.Automation.HiddenAttribute has been added so that C# code can have the same semantics within Windows PowerShell. |
| Return types | Return type is a contract; the return value is converted to the expected type. If no return type is specified, the return type is void. There is no streaming of objects; objects cannot be written to the pipeline either intentionally or by accident. |
| Attributes | Four new attributes, **DscResource**, **DscResourceKey**, **DscResourceMandatory**, and **DscResourceOut** have been added. |
| Lexical scoping of variables | The following shows an example of how lexical scoping works in this release.  $d = 42 # Script scope  function bar  {  $d = 0 # Function scope  [MyClass]::DoSomething()  }  class MyClass  {  static [object] DoSomething()  {  return $d # error, not found dynamically  return $script:d # no error  $d = $script:d  return $d # no error, found lexically  }  }  $v = bar  $v -eq $d # true |

The following example creates several new, custom classes to implement an HTML dynamic style sheet language (DSL). Then, the example adds helper functions to create specific element types as part of the element class, such as heading styles and tables, because types cannot be used outside the scope of a module.

# Classes that define the structure of the document

#

class Html

{

[string] $docType

[HtmlHead] $Head

[Element[]] $Body

[string] Render()

{

$text = "<html>`n<head>`n"

$text += $this.Head

$text += "`n</head>`n<body>`n"

$text += $this.Body -join "`n" # Render all of the body elements

$text += "</body>`n</html>"

return $text

}

[string] ToString() { return $this.Render() }

}

class HtmlHead

{

$Title

$Base

$Link

$Style

$Meta

$Script

[string] Render() { return "<title>$($this.Title)</title>" }

[string] ToString() { return $this.Render() }

}

class Element

{

[string] $Tag

[string] $Text

[hashtable] $Attributes

[string] Render() {

$attributesText= ""

if ($this.Attributes)

{

foreach ($attr in $this.Attributes.Keys)

{

#BUGBUG - need to validate keys against attribute

$attributesText += " $attr=`"$($this.Attributes[$attr])`""

}

}

return "<$($this.tag)${attributesText}>$($this.text)</$($this.tag)>`n"

}

[string] ToString() { return $this.Render() }

}

#

# Helper functions for creating specific element types on top of the classes.

# These are required because types aren’t visible outside of the module.

#

function H1 { [Element] @{ Tag = "H1" ; Text = $args.foreach{$\_} -join " " }}

function H2 { [Element] @{ Tag = "H2" ; Text = $args.foreach{$\_} -join " " }}

function H3 { [Element] @{ Tag = "H3" ; Text = $args.foreach{$\_} -join " " }}

function P { [Element] @{ Tag = "P" ; Text = $args.foreach{$\_} -join " " }}

function B { [Element] @{ Tag = "B" ; Text = $args.foreach{$\_} -join " " }}

function I { [Element] @{ Tag = "I" ; Text = $args.foreach{$\_} -join " " }}

function HREF

{

param (

$Name,

$Link

)

return [Element] @{

Tag = "A"

Attributes = @{ HREF = $link }

Text = $name

}

}

function Table

{

param (

[Parameter(Mandatory)]

[object[]]

$Data,

[Parameter()]

[string[]]

$Properties = "\*",

[Parameter()]

[hashtable]

$Attributes = @{ border=2; cellpadding=2; cellspacing=2 }

)

$bodyText = ""

# Add the header tags

$bodyText += $Properties.foreach{TH $\_}

# Add the rows

$bodyText += foreach ($row in $Data)

{

TR (-join $Properties.Foreach{ TD ($row.$\_) } )

}

$table = [Element] @{

Tag = "Table"

Attributes = $Attributes

Text = $bodyText

}

$table

}

function TH { ([Element] @{ Tag = "TH" ; Text = $args.foreach{$\_} -join " " }) }

function TR { ([Element] @{ Tag = "TR" ; Text = $args.foreach{$\_} -join " " }) }

function TD { ([Element] @{ Tag = "TD" ; Text = $args.foreach{$\_} -join " " }) }

function Style

{

return [Element] @{

Tag = "style"

Text = "$args"

}

}

# Takes a hash table, casts it to and HTML document

# and then returns the resulting type.

#

function Html ([HTML] $doc) { return $doc }

## More control and remoting in Windows PowerShell debugging

There have been some improvements to Windows PowerShell debugging after WMF 5.0 September Preview.

* The Enter-PSHostProcess cmdlet has a new, optional **AppDomainName** parameter. If you specify this parameter, Enter-PSHostProcess tries to connect to the specified app domain in the target process. If you do not specify this parameter, Enter-PSHostProcess tries to connect to the default domain in the process.
* Get-PSHostProcessInfo now returns AppDomain names along with processes to which you can connect by running the Enter-PSHostProcess cmdlet.

# Known issues

## Reverting to a Windows 8.1 (2012 R2) build from WMF 5.0 Preview can break DSC cmdlets because of updates in metaconfig.mof

Resolution: Delete metaconfig.mof.

1. Open **powershell.exe** with elevated user rights (run as administrator).
2. Run the following command in the console:
   1. Remove-Item -Path $env:SystemRoot\system32\Configuration\metaconfig.mof

## WMF 5.0 Preview installation can appear to succeed, but the installation rolls back after the system is restarted (or just fails)

Resolution: Delete the **\\root\microsoft\windows\desiredstateconfiguration** namespace in WMI.

1. Open **powershell.exe** with elevated user rights (**Run as Administrator**).
2. Run the following commands:
   1. $dscNamespace = Get-CimInstance -Namespace root\microsoft\windows -Query "select \* from \_\_namespace where name = 'desiredstateconfiguration'"
   2. $dscNamespace | Remove-CimInstance
   3. mofcomp.exe $env:windir\system32\wbem\DSCCoreConfProv.mof
3. Install the WMF 5.0 Preview package.

## Modules and DSC Resources not working reporting error about executionpolicy on Windows 7

Resolution: Change ExecutionPolicy to RemoteSigned

1. Open **powershell.exe** with elevated user rights (run as administrator).
2. Run the following command in the console:
   1. Set-ExecutionPolicy RemoteSigned

## Connecting to an old remote exchange endpoint causes a crash

The old Exchange endpoint redirects to a new endpoint. There is a bug in the redirection logic that results in a crash.

Resolution: Connect directly to the new endpoint

## PowerShell Shortcuts are broken When used for the first time

Resolution: Perform one of the following actions:

1. Right click on the PowerShell shortcut. Select “Windows PowerShell” to launch in a non-elevated mode.
2. Right click on the PowerShell shortcut. Right click on “Windows PowerShell” and select “Run As Administrator” to launch in an elevated mode.

Once you have performed either of the above actions, the PowerShell shortcuts will work. These actions only need to be performed only once.

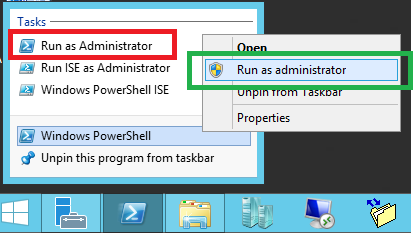


Figure 1: Select the green box, not the red box.

# Known incompatibilities

## Systems that are running the following server applications should not run Windows Management Framework 5.0 at this time

* + - System Center 2012 Configuration Manager (not including SP1)
    - Windows Small Business Server 2011 Standard

## WindowsOptionalFeature not available in Windows 7

The WindowsOptionalFeature DSC resource is not available in Windows 7. This resource requires the DISM module, and DISM cmdlets that are available starting in Windows 8 and newer releases of the Windows operating system.

# Feedback and contact

For issues or feedback you would like to report to us, use our Connect website.

<https://connect.microsoft.com/PowerShell/Feedback>

# Additional links

Windows PowerShell Team Blog: http://blogs.msdn.com/b/powershell/