

Table of Contents

Overview

[What is Azure Automation?](#)

[Security in Azure Automation](#)

Get started

[Create runbook](#)

[Create PowerShell runbook](#)

[Create PowerShell Workflow runbook](#)

[Manage role-based access control](#)

How to

[Create runbooks](#)

[Runbook types](#)

[Create and import runbooks](#)

[Edit textual runbooks](#)

[Edit graphical runbooks](#)

[Test a runbook](#)

[Learning PowerShell Workflow](#)

[Child runbooks](#)

[Runbook output](#)

[Source control integration](#)

[Automate runbooks](#)

[Start a runbook](#)

[Start a runbook from a webhook](#)

[Run runbooks in your datacenter](#)

[Configure runbook input parameters](#)

[Error handling in graphical runbooks](#)

[Track a runbook job](#)

[Change runbook settings](#)

[Manage Azure Automation data](#)

[Call Azure Automation Runbook from Log Analytics alert](#)

[Deploy configuration management \(DSC\)](#)

[Desired State Configuration \(DSC\)](#)

[Getting started](#)

[Onboarding machines for management](#)

[Compiling DSC configurations](#)

[Continuous deployment using Chocolatey](#)

[Set up authentication](#)

[Azure Service Management and Resource Manager](#)

[Amazon Web Services](#)

[Azure RunAs account](#)

[Manage automation assets](#)

[Certificates](#)

[Connections](#)

[Credentials](#)

[Integration Modules](#)

[Schedules](#)

[Variables](#)

[Update Azure PowerShell modules](#)

[Automate scenarios](#)

[Runbook gallery](#)

[Start/stop virtual machines](#)

[Start/stop virtual machines with PowerShell](#)

[Create Amazon Web Service VM](#)

[Remediate Azure VM alert](#)

[Start/stop VM with JSON Tags](#)

[Remove Resource Group](#)

[Start/stop VMs during off-hours](#)

[Source control integration with GitHub Enterprise](#)

[Source control integration with VSTS](#)

[Monitor](#)

[Forward Azure Automation job data to Log Analytics](#)

[Unlink Azure Automation account from Log Analytics](#)

Migrate

[Migrate from Orchestrator](#)

[Move Automation Account](#)

Troubleshoot

[Troubleshoot common errors](#)

[Troubleshoot Hybrid Runbook Worker](#)

Reference

[PowerShell](#)

[PowerShell \(Classic\)](#)

[.NET](#)

[REST](#)

[REST \(Classic\)](#)

Resources

[Release notes](#)

[Pricing](#)

[MSDN forum](#)

[Stack Overflow](#)

[Videos](#)

[Service updates](#)

[Azure Automation training](#)

[Learning path](#)

[Automation introduction video](#)

Azure Automation overview

1/17/2017 • 6 min to read • [Edit on GitHub](#)

Microsoft Azure Automation provides a way for users to automate the manual, long-running, error-prone, and frequently repeated tasks that are commonly performed in a cloud and enterprise environment. It saves time and increases the reliability of regular administrative tasks and even schedules them to be automatically performed at regular intervals. You can automate processes using runbooks or automate configuration management using Desired State Configuration. This article provides brief overview of Azure Automation and answers some common questions. You can refer to other articles in this library for more detailed information on the different topics.

Automating processes with runbooks

A runbook is a set of tasks that perform some automated process in Azure Automation. It may be a simple process such as starting a virtual machine and creating a log entry, or you may have a complex runbook that combines other smaller runbooks to perform a complex process across multiple resources or even multiple clouds and on-premise environments.

For example, you might have an existing manual process for truncating a SQL database if it's approaching maximum size that includes multiple steps such as connecting to the server, connecting to the database, get the current size of database, check if threshold has exceeded and then truncate it and notify user. Instead of manually performing each of these steps, you could create a runbook that would perform all of these tasks as a single process. You would start the runbook, provide the required information such as the SQL server name, database name, and recipient e-mail and then sit back while the process completes.

What can runbooks automate?

Runbooks in Azure Automation are based on Windows PowerShell or Windows PowerShell Workflow, so they do anything that PowerShell can do. If an application or service has an API, then a runbook can work with it. If you have a PowerShell module for the application, then you can load that module into Azure Automation and include those cmdlets in your runbook. Azure Automation runbooks run in the Azure cloud and can access any cloud resources or external resources that can be accessed from the cloud. Using [Hybrid Runbook Worker](#), runbooks can run in your local data center to manage local resources.

Getting runbooks from the community

The [Runbook Gallery](#) contains runbooks from Microsoft and the community that you can either use unchanged in your environment or customize them for your own purposes. They are also useful to as references to learn how to create your own runbooks. You can even contribute your own runbooks to the gallery that you think other users may find useful.

Creating Runbooks with Azure Automation

You can [create your own runbooks](#) from scratch or modify runbooks from the [Runbook Gallery](#) for your own requirements. There are four different [runbook types](#) that you can choose from based on your requirements and PowerShell experience. If you prefer to work directly with the PowerShell code, then you can use a [PowerShell runbook](#) or [PowerShell Workflow runbook](#) that you edit offline or with the [textual editor](#) in the Azure portal. If you prefer to edit a runbook without being exposed to the underlying code, then you can create a [Graphical runbook](#) using the [graphical editor](#) in the Azure portal.

Prefer watching to reading? Have a look at the below video from Microsoft Ignite session in May 2015. Note: While

the concepts and features discussed in this video are correct, Azure Automation has progressed a lot since this video was recorded, it now has a more extensive UI in the Azure portal, and supports additional capabilities.

Automating configuration management with Desired State Configuration

[PowerShell DSC](#) is a management platform that allows you to manage, deploy and enforce configuration for physical hosts and virtual machines using a declarative PowerShell syntax. You can define configurations on a central DSC Pull Server that target machines can automatically retrieve and apply. DSC provides a set of PowerShell cmdlets that you can use to manage configurations and resources.

[Azure Automation DSC](#) is a cloud based solution for PowerShell DSC that provides services required for enterprise environments. You can manage your DSC resources in Azure Automation and apply configurations to virtual or physical machines that retrieve them from a DSC Pull Server in the Azure cloud. It also provides reporting services that inform you of important events such as when nodes have deviated from their assigned configuration and when a new configuration has been applied.

Creating your own DSC configurations with Azure Automation

[DSC configurations](#) specify the desired state of a node. Multiple nodes can apply the same configuration to assure that they all maintain an identical state. You can create a configuration using any text editor on your local machine and then import it into Azure Automation where you can compile it and apply it nodes.

Getting modules and configurations

You can get [PowerShell modules](#) containing cmdlets that you can use in your runbooks and DSC configurations from the [PowerShell Gallery](#). You can launch this gallery from the Azure portal and import modules directly into Azure Automation, or you can download and import them manually. You cannot install the modules directly from the Azure portal, but you can download them and install them as you would any other module.

Example practical applications of Azure Automation

Following are just a few examples of what are the kinds of automation scenarios with Azure Automation.

- Create and copy virtual machines in different Azure subscriptions.
- Schedule file copies from a local machine to an Azure Blob Storage container.
- Automate security functions such as deny requests from a client when a denial of service attack is detected.
- Ensure machines continually align with configured security policy.
- Manage continuous deployment of application code across cloud and on premises infrastructure.
- Build an Active Directory forest in Azure for your lab environment.
- Truncate a table in a SQL database if DB is approaching maximum size.
- Remotely update environment settings for an Azure website.

How does Azure Automation relate to other automation tools?

[Service Management Automation \(SMA\)](#) is intended to automate management tasks in the private cloud. It is

installed locally in your data center as a component of [Microsoft Azure Pack](#). SMA and Azure Automation use the same runbook format based on Windows PowerShell and Windows PowerShell Workflow, but SMA does not support [graphical runbooks](#).

[System Center 2012 Orchestrator](#) is intended for automation of on-premises resources. It uses a different runbook format than Azure Automation and Service Management Automation and has a graphical interface to create runbooks without requiring any scripting. Its runbooks are composed of activities from Integration Packs that are written specifically for Orchestrator.

Where can I get more information?

A variety of resources are available for you to learn more about Azure Automation and creating your own runbooks.

- [**Azure Automation Library**](#) is where you are right now. The articles in this library provide complete documentation on the configuration and administration of Azure Automation and for authoring your own runbooks.
- [**Azure PowerShell cmdlets**](#) provides information for automating Azure operations using Windows PowerShell. Runbooks use these cmdlets to work with Azure resources.
- [**Management Blog**](#) provides the latest information on Azure Automation and other management technologies from Microsoft. You should subscribe to this blog to stay up to date with the latest from the Azure Automation team.
- [**Automation Forum**](#) allows you to post questions about Azure Automation to be addressed by Microsoft and the Automation community.
- [**Azure Automation Cmdlets**](#) provides information for automating administration tasks. It contains cmdlets to manage Automation accounts, assets, runbooks, DSC.

Can I provide feedback?

Please give us feedback! If you are looking for an Azure Automation runbook solution or an integration module, post a Script Request on Script Center. If you have feedback or feature requests for Azure Automation, post them on [User Voice](#). Thanks!

Azure Automation security

1/17/2017 • 3 min to read • [Edit on GitHub](#)

Azure Automation allows you to automate tasks against resources in Azure, on-premises, and with other cloud providers such as Amazon Web Services (AWS). In order for a runbook to perform its required actions, it must have permissions to securely access the resources with the minimal rights required within the subscription. This article will cover the various authentication scenarios supported by Azure Automation and will show you how to get started based on the environment or environments you need to manage.

Automation Account overview

When you start Azure Automation for the first time, you must create at least one Automation account. Automation accounts allow you to isolate your Automation resources (runbooks, assets, configurations) from the resources contained in other Automation accounts. You can use Automation accounts to separate resources into separate logical environments. For example, you might use one account for development, another for production, and another for your on-premises environment. An Azure Automation account is different from your Microsoft account or accounts created in your Azure subscription.

The Automation resources for each Automation account are associated with a single Azure region, but Automation accounts can manage all the resources in your subscription. The main reason to create Automation accounts in different regions would be if you have policies that require data and resources to be isolated to a specific region.

NOTE

Automation accounts, and the resources they contain that are created in the Azure portal, cannot be accessed in the Azure classic portal. If you want to manage these accounts or their resources with Windows PowerShell, you must use the Azure Resource Manager modules.

All of the tasks that you perform against resources using Azure Resource Manager and the Azure cmdlets in Azure Automation must authenticate to Azure using Azure Active Directory organizational identity credential-based authentication. Certificate-based authentication was the original authentication method with Azure Service Management mode, but it was complicated to setup. Authenticating to Azure with Azure AD user was introduced back in 2014 to not only simplify the process to configure an Authentication account, but also support the ability to non-interactively authenticate to Azure with a single user account that worked with both Azure Resource Manager and classic resources.

Currently when you create a new Automation account in the Azure portal, it automatically creates:

- Run As account which creates a new service principal in Azure Active Directory, a certificate, and assigns the Contributor role-based access control (RBAC), which will be used to manage Resource Manager resources using runbooks.
- Classic Run As account by uploading a management certificate, which will be used to manage Azure Service Management or classic resources using runbooks.

Role-based access control is available with Azure Resource Manager to grant permitted actions to an Azure AD user account and Run As account, and authenticate that service principal. Please read [Role-based access control in Azure Automation article](#) for further information to help develop your model for managing Automation permissions.

Runbooks running on a Hybrid Runbook Worker in your datacenter or against computing services in AWS cannot use the same method that is typically used for runbooks authenticating to Azure resources. This is because those

resources are running outside of Azure and therefore, will require their own security credentials defined in Automation to authenticate to resources that they will access locally.

Authentication methods

The following table summarizes the different authentication methods for each environment supported by Azure Automation and the article describing how to setup authentication for your runbooks.

METHOD	ENVIRONMENT	ARTICLE
Azure AD User Account	Azure Resource Manager and Azure Service Management	Authenticate Runbooks with Azure AD User account
Azure Run As Account	Azure Resource Manager	Authenticate Runbooks with Azure Run As account
Azure Classic Run As Account	Azure Service Management	Authenticate Runbooks with Azure Run As account
Windows Authentication	On-Premises Datacenter	Authenticate Runbooks for Hybrid Runbook Workers
AWS Credentials	Amazon Web Services	Authenticate Runbooks with Amazon Web Services (AWS)

My first graphical runbook

1/17/2017 • 14 min to read • [Edit on GitHub](#)

This tutorial walks you through the creation of a [graphical runbook](#) in Azure Automation. We'll start with a simple runbook that we'll test and publish while we explain how to track the status of the runbook job. Then we'll modify the runbook to actually manage Azure resources, in this case starting an Azure virtual machine. We'll then make the runbook more robust by adding runbook parameters and conditional links.

Prerequisites

To complete this tutorial, you will need the following.

- Azure subscription. If you don't have one yet, you can [activate your MSDN subscriber benefits](#) or [\[sign up for a free account\]\(https://azure.microsoft.com/free/\)](#).
- [Azure Run As Account](#) to hold the runbook and authenticate to Azure resources. This account must have permission to start and stop the virtual machine.
- An Azure virtual machine. We will stop and start this machine so it should not be production.

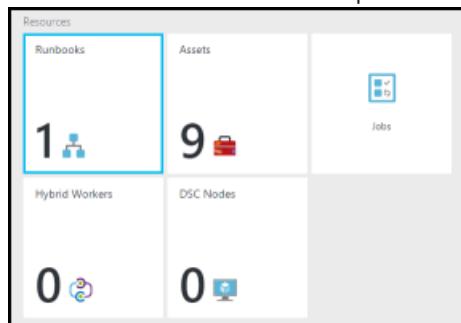
Step 1 - Create new runbook

We'll start by creating a simple runbook that outputs the text *Hello World*.

1. In the Azure Portal, open your Automation account.

The Automation account page gives you a quick view of the resources in this account. You should already have some Assets. Most of those are the modules that are automatically included in a new Automation account. You should also have the Credential asset that's mentioned in the [prerequisites](#).

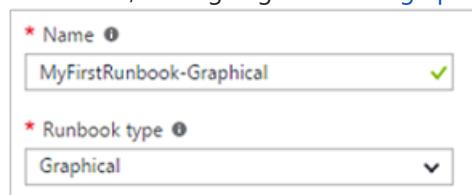
2. Click on the **Runbooks** tile to open the list of runbooks.



3. Create a new runbook by clicking on the **Add a runbook** button and then **Create a new runbook**.

4. Give the runbook the name *MyFirstRunbook-Graphical*.

5. In this case, we're going to create a [graphical runbook](#) so select **Graphical** for **Runbook type**.

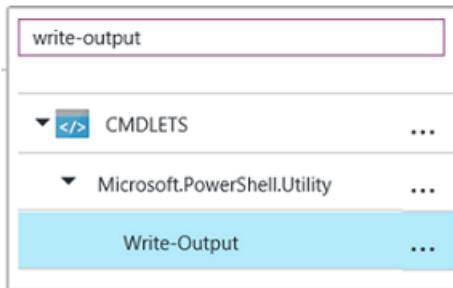


6. Click **Create** to create the runbook and open the graphical editor.

Step 2 - Add activities to the runbook

The Library control on the left side of the editor allows you to select activities to add to your runbook. We're going to add a **Write-Output** cmdlet to output text from the runbook.

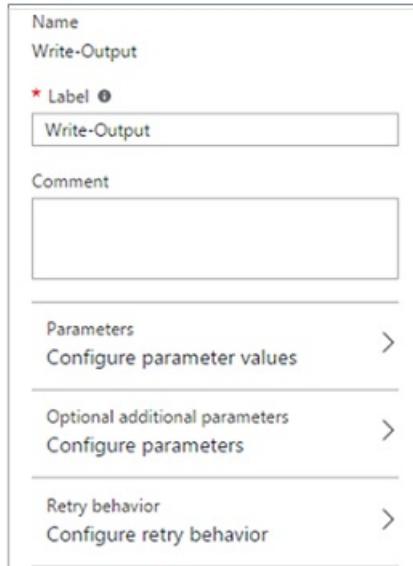
1. In the Library control, click in the search textbox and type **Write-Output**. The search results will be displayed below.



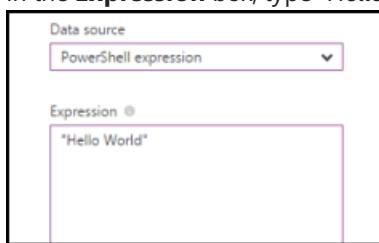
2. Scroll down to the bottom of the list. You can either right click **Write-Output** and select **Add to canvas** or click on the ellipse next to the cmdlet and then select **Add to canvas**.
3. Click on the **Write-Output** activity on the canvas. This opens the Configuration control blade which allows you to configure the activity.
4. The **Label** defaults to the name of the cmdlet, but we can change it to something more friendly. Change it to *Write Hello World to output*.

5. Click **Parameters** to provide values for the cmdlet's parameters.

Some cmdlets have multiple parameter sets, and you need to select which you will use. In this case, **Write-Output** has only one parameter set, so you don't need to select one.



6. Select the **InputObject** parameter. This is the parameter where we will specify the text to send to the output stream.
7. In the **Data source** dropdown, select **PowerShell expression**. The **Data source** dropdown provides different sources that you use to populate a parameter value.
You can use output from such sources such as another activity, an Automation asset, or a PowerShell expression. In this case, we just want to output the text *Hello World*. We can use a PowerShell expression and specify a string.
8. In the **Expression** box, type "*Hello World*" and then click **OK** twice to return to the canvas.



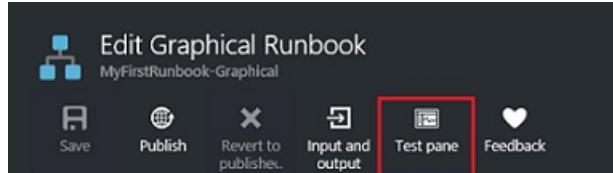
9. Save the runbook by clicking **Save**.



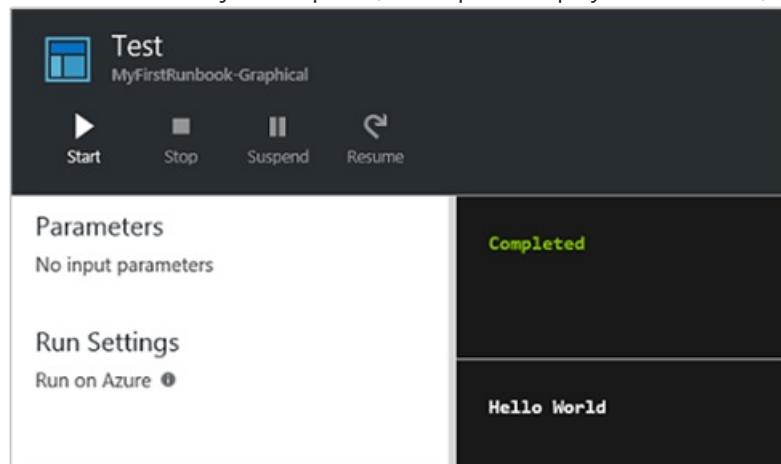
Step 3 - Test the runbook

Before we publish the runbook to make it available in production, we want to test it to make sure that it works properly. When you test a runbook, you run its **Draft** version and view its output interactively.

1. Click **Test pane** to open the Test blade.



2. Click **Start** to start the test. This should be the only enabled option.
3. A **runbook job** is created and its status displayed in the pane.
The job status will start as *Queued* indicating that it is waiting for a runbook worker in the cloud to become available. It will then move to *Starting* when a worker claims the job, and then *Running* when the runbook actually starts running.
4. When the runbook job completes, its output is displayed. In our case, we should see *Hello World*.



5. Close the Test blade to return to the canvas.

Step 4 - Publish and start the runbook

The runbook that we just created is still in Draft mode. We need to publish it before we can run it in production. When you publish a runbook, you overwrite the existing Published version with the Draft version. In our case, we don't have a Published version yet because we just created the runbook.

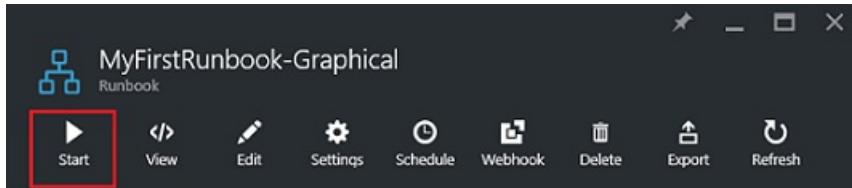
1. Click **Publish** to publish the runbook and then **Yes** when prompted.



2. If you scroll left to view the runbook in the **Runbooks** blade, it will show an **Authoring Status of Published**.
3. Scroll back to the right to view the blade for **MyFirstRunbook**.

The options across the top allow us to start the runbook, schedule it to start at some time in the future, or create a [webhook](#) so it can be started through an HTTP call.

4. We just want to start the runbook so click **Start** and then **Yes** when prompted.



5. A job blade is opened for the runbook job that we just created. We can close this blade, but in this case we'll leave it open so we can watch the job's progress.

6. The job status is shown in **Job Summary** and matches the statuses that we saw when we tested the runbook.

Job Summary	
Job ID: 2609d497-b5c3-41d4-9f43-aeeb12b65499	Runbook
Created: 3/14/2016 4:30 PM	
Last update: 3/14/2016 4:31 PM	
Ran on Azure	
Completed	

Status		
Errors	0 ✖	Warnings
		0 ⚠ All Logs

Exception	
None	

7. Once the runbook status shows *Completed*, click **Output**. The **Output** blade is opened, and we can see our *Hello World* in the pane.

Job Summary	
Job ID: 2609d497-b5c3-41d4-9f43-aeeb12b65499	Runbook
Created: 3/14/2016 4:30 PM	
Last update: 3/14/2016 4:31 PM	
Ran on Azure	
Completed	

Status		
Errors	0 ✖	Warnings
		0 ⚠ All Logs

Exception	
None	

Output	
Hello World	

8. Close the Output blade.

9. Click **All Logs** to open the Streams blade for the runbook job. We should only see *Hello World* in the output stream, but this can show other streams for a runbook job such as Verbose and Error if the runbook writes to them.

The screenshot shows the 'Job Summary' section with a green checkmark indicating 'Completed'. It also displays the job ID, creation date, last update, and run location. Below this is the 'Status' section with 'Errors' (0) and 'Warnings' (0). A red box highlights the 'All Logs' button, which is currently active. At the bottom, there's an 'Exception' section stating 'None'.

10. Close the All Logs blade and the Job blade to return to the MyFirstRunbook blade.
11. Click **Jobs** to open the Jobs blade for this runbook. This lists all of the jobs created by this runbook. We should only see one job listed since we only ran the job once.

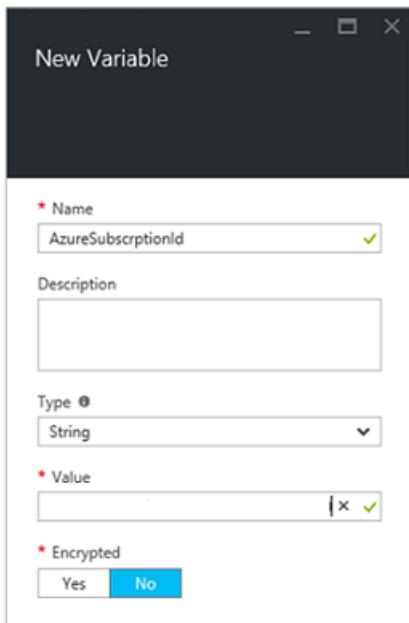
The screenshot shows the 'Details' section of the Jobs blade, which contains a single job entry. A red box highlights this job entry. To the right are sections for 'Schedules' (0) and 'Webhooks' (0).

12. You can click on this job to open the same Job pane that we viewed when we started the runbook. This allows you to go back in time and view the details of any job that was created for a particular runbook.

Step 5 - Create variable assets

We've tested and published our runbook, but so far it doesn't do anything useful. We want to have it manage Azure resources. Before we configure the runbook to authenticate, we will create a variable to hold the subscription ID and reference it after we setup the activity to authenticate in step 6 below. Including a reference to the subscription context allows you to easily work between multiple subscriptions. Before proceeding, copy your subscription ID from the Subscriptions option off of the Navigation pane.

1. In the Automation Accounts blade, click on the **Assets** tile and the **Assets** blade is opened.
2. In the Assets blade, click on the **Variables** tile.
3. On the Variables blade, click **Add a variable**.

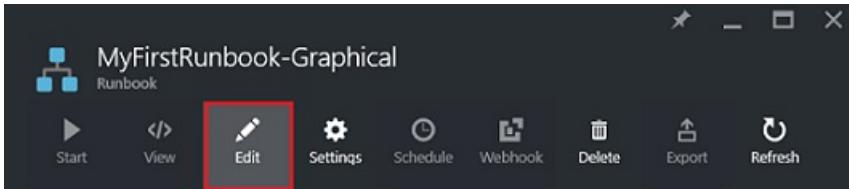


4. In the New variable blade, in the **Name** box, enter **AzureSubscriptionId** and in the **Value** box enter your Subscription ID. Keep *string* for the **Type** and the default value for **Encryption**.
5. Click **Create** to create the variable.

Step 6 - Add authentication to manage Azure resources

Now that we have a variable to hold our subscription ID, we can configure our runbook to authenticate with the Run As credentials that are referred to in the [prerequisites](#). We do that by adding the Azure Run As connection **Asset** and **Add-AzureRMAccount** cmdlet to the canvas.

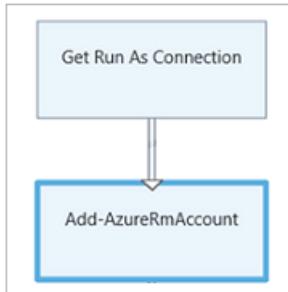
1. Open the graphical editor by clicking **Edit** on the MyFirstRunbook blade.



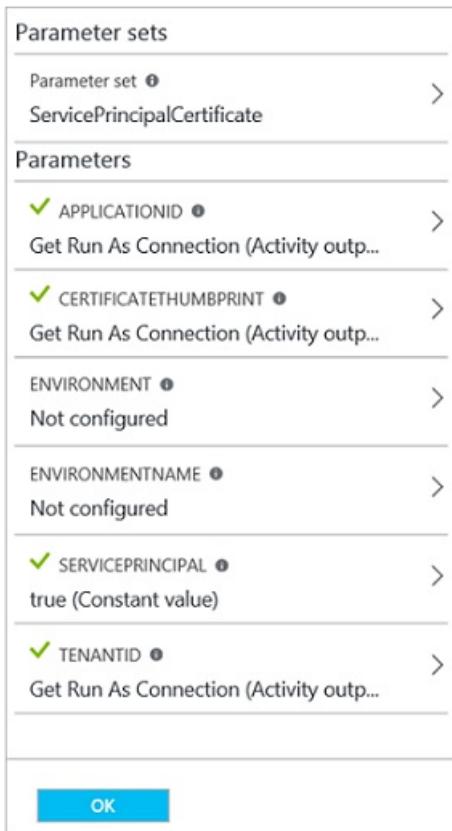
2. We don't need the **Write Hello World to output** anymore, so right click it and select **Delete**.
3. In the Library control, expand **Connections** and add **AzureRunAsConnection** to the canvas by selecting **Add to canvas**.
4. On the canvas, select **AzureRunAsConnection** and in the Configuration control pane, type **Get Run As Connection** in the **Label** textbox. This is the connection
5. In the Library control, type **Add-AzureRmAccount** in the search textbox.
6. Add **Add-AzureRmAccount** to the canvas.

add-azureRM		
CMDLETS		
▼	AzureRM.Compute	...
	Add-AzureRmVhd	...
	Add-AzureRmVMAdditionalUnit	...
	Add-AzureRmVMDataDisk	...
	Add-AzureRmVMNetworkInterface	...
	Add-AzureRmVMSecret	...
	Add-AzureRmVMSshPublicKey	...
▼	AzureRM.Profile	...
	Add-AzureRmAccount	...

7. Hover over **Get Run As Connection** until a circle appears on the bottom of the shape. Click the circle and drag the arrow to **Add-AzureRmAccount**. The arrow that you just created is a *link*. The runbook will start with **Get Run As Connection** and then run **Add-AzureRmAccount**.

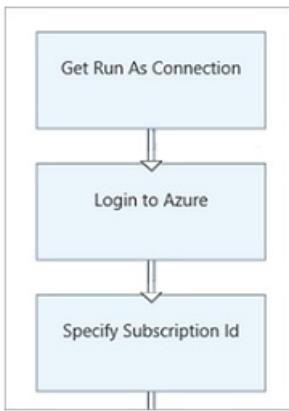


8. On the canvas, select **Add-AzureRmAccount** and in the Configuration control pane type **Login to Azure** in the **Label** textbox.
9. Click **Parameters** and the Activity Parameter Configuration blade appears.
10. **Add-AzureRmAccount** has multiple parameter sets, so we need to select one before we can provide parameter values. Click **Parameter Set** and then select the **ServicePrincipalCertificate** parameter set.
11. Once you select the parameter set, the parameters are displayed in the Activity Parameter Configuration blade. Click **APPLICATIONID**.



12. In the Parameter Value blade, select **Activity output** for the **Data source** and select **Get Run As Connection** from the list, in the **Field path** textbox type **ApplicationId**, and then click **OK**. We are specifying the name of the property for the Field path because the activity outputs an object with multiple properties.
13. Click **CERTIFICATETHUMBPRINT**, and in the Parameter Value blade, select **Activity output** for the **Data source**. Select **Get Run As Connection** from the list, in the **Field path** textbox type **CertificateThumbprint**, and then click **OK**.
14. Click **SERVICEPRINCIPAL**, and in the Parameter Value blade, select **ConstantValue** for the **Data source**, click the option **True**, and then click **OK**.
15. Click **TENANTID**, and in the Parameter Value blade, select **Activity output** for the **Data source**. Select **Get Run As Connection** from the list, in the **Field path** textbox type **TenantId**, and then click **OK** twice.
16. In the Library control, type **Set-AzureRmContext** in the search textbox.
17. Add **Set-AzureRmContext** to the canvas.
18. On the canvas, select **Set-AzureRmContext** and in the Configuration control pane type **Specify Subscription Id** in the **Label** textbox.
19. Click **Parameters** and the Activity Parameter Configuration blade appears.
20. **Set-AzureRmContext** has multiple parameter sets, so we need to select one before we can provide parameter values. Click **Parameter Set** and then select the **SubscriptionId** parameter set.
21. Once you select the parameter set, the parameters are displayed in the Activity Parameter Configuration blade. Click **SubscriptionID**
22. In the Parameter Value blade, select **Variable Asset** for the **Data source** and select **AzureSubscriptionId** from the list and then click **OK** twice.
23. Hover over **Login to Azure** until a circle appears on the bottom of the shape. Click the circle and drag the arrow to **Specify Subscription Id**.

Your runbook should look like the following at this point:



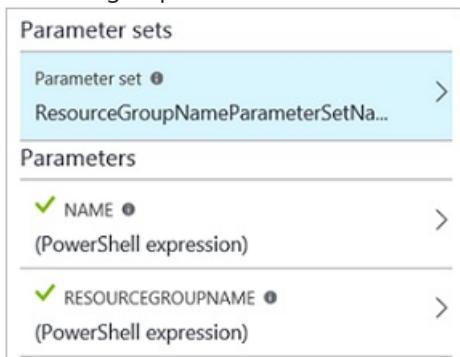
Step 7 - Add activity to start a virtual machine

We'll now add a **Start-AzureRmVM** activity to start a virtual machine. You can pick any virtual machine in your Azure subscription, and for now we'll be hardcoding that name into the cmdlet.

1. In the Library control, type **Start-AzureRm** in the search textbox.
2. Add **Start-AzureRmVM** to the canvas and then click and drag it underneath **Specify Subscription Id**.
3. Hover over **Specify Subscription Id** until a circle appears on the bottom of the shape. Click the circle and drag the arrow to **Start-AzureRmVM**.
4. Select **Start-AzureRmVM**. Click **Parameters** and then **Parameter Set** to view the sets for **Start-AzureRmVM**. Select the **ResourceGroupNameParameterSetName** parameter set. Note that **ResourceGroupName** and **Name** have exclamation points next them. This indicates that they are required parameters. Also note both expect string values.
5. Select **Name**. Select **PowerShell expression** for the **Data source** and type in the name of the virtual machine surrounded with double quotes that we will start with this runbook. Click **OK**.

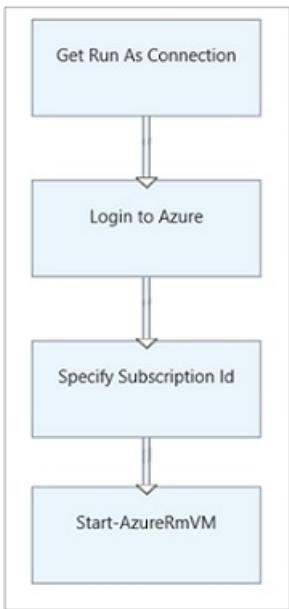


6. Select **ResourceGroupName**. Use **PowerShell expression** for the **Data source** and type in the name of the resource group surrounded with double quotes. Click **OK**.



7. Click Test pane so that we can test the runbook.
8. Click **Start** to start the test. Once it completes, check that the virtual machine was started.

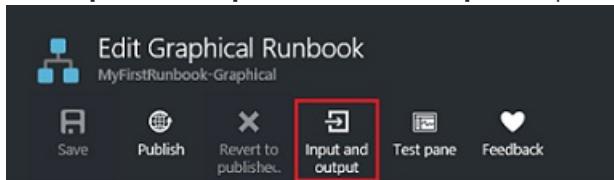
Your runbook should look like the following at this point:



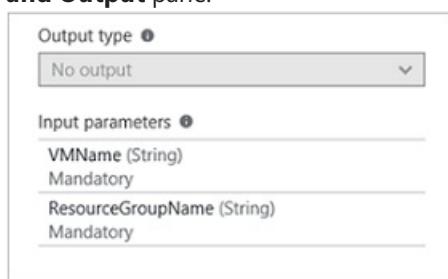
Step 8 - Add additional input parameters to the runbook

Our runbook currently starts the virtual machine in the resource group that we specified in the **Start-AzureRmVM** cmdlet, but our runbook would be more useful if we could specify both when the runbook is started. We will now add input parameters to the runbook to provide that functionality.

1. Open the graphical editor by clicking **Edit** on the **MyFirstRunbook** pane.
2. Click **Input and output** and then **Add input** to open the Runbook Input Parameter pane.



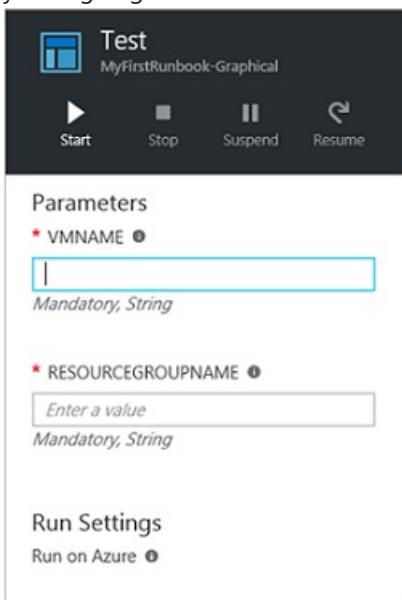
3. Specify *VMName* for the **Name**. Keep *string* for the **Type**, but change **Mandatory** to Yes. Click **OK**.
4. Create a second mandatory input parameter called *ResourceGroupName* and then click **OK** to close the **Input and Output** pane.



5. Select the **Start-AzureRmVM** activity and then click **Parameters**.
6. Change the **Data source** for **Name** to **Runbook input** and then select **VMName**.
7. Change the **Data source** for **ResourceGroupName** to **Runbook input** and then select **ResourceGroupName**.

The screenshot shows the 'Parameter sets' and 'Parameters' sections of the Azure Runbook configuration interface. Under 'Parameter sets', there is one entry: 'Parameter set 1' with 'ResourceGroupNameParameterSetNa...'. Under 'Parameters', there are two entries: 'NAME' (Runbook input) and 'RESOURCEGROUPNAME' (Runbook input), both marked with a green checkmark.

8. Save the runbook and open the Test pane. Note that you can now provide values for the two input variables that will be used in the test.
9. Close the Test pane.
10. Click **Publish** to publish the new version of the runbook.
11. Stop the virtual machine that you started in the previous step.
12. Click **Start** to start the runbook. Type in the **VMName** and **ResourceGroupName** for the virtual machine that you're going to start.



13. When the runbook completes, check that the virtual machine was started.

Step 9 - Create a conditional link

We will now modify the runbook so that it will only attempt to start the virtual machine if it is not already started. We'll do this by adding a **Get-AzureRmVM** cmdlet to the runbook that will get the instance level status of the virtual machine. We'll then add a PowerShell Workflow code module called **Get Status** with a snippet of PowerShell code to determine if the virtual machine state is running or stopped. A conditional link from the **Get Status** module will only run **Start-AzureRmVM** if the current running state is stopped. Finally, we will output a message to inform you if the VM was successfully started or not using the PowerShell Write-Output cmdlet.

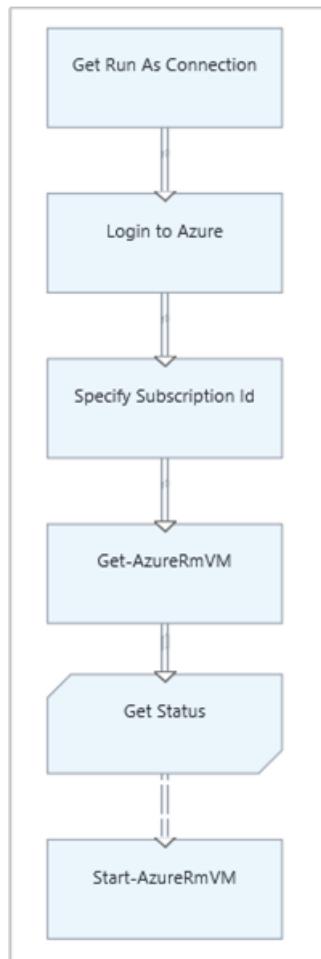
1. Open **MyFirstRunbook** in the graphical editor.
2. Remove the link between **Specify Subscription Id** and **Start-AzureRmVM** by clicking on it and then pressing the *Delete* key.
3. In the Library control, type **Get-AzureRm** in the search textbox.
4. Add **Get-AzureRmVM** to the canvas.
5. Select **Get-AzureRmVM** and then **Parameter Set** to view the sets for **Get-AzureRmVM**. Select the **GetVirtualMachineInResourceGroupNameParamSet** parameter set. Note that **ResourceGroupName** and **Name** have exclamation points next them. This indicates that they are required parameters. Also note both

expect string values.

6. Under **Data source** for **Name**, select **Runbook input** and then select **VMName**. Click **OK**.
7. Under **Data source** for **ResourceGroupName**, select **Runbook input** and then select **ResourceGroupName**. Click **OK**.
8. Under **Data source** for **Status**, select **Constant value** and then click on **True**. Click **OK**.
9. Create a link from **Specify Subscription Id** to **Get-AzureRmVM**.
10. In the library control, expand **Runbook Control** and add **Code** to the canvas.
11. Create a link from **Get-AzureRmVM** to **Code**.
12. Click **Code** and in the Configuration pane, change label to **Get Status**.
13. Select **Code** parameter, and the **Code Editor** blade appears.
14. In the code editor, paste the following snippet of code:

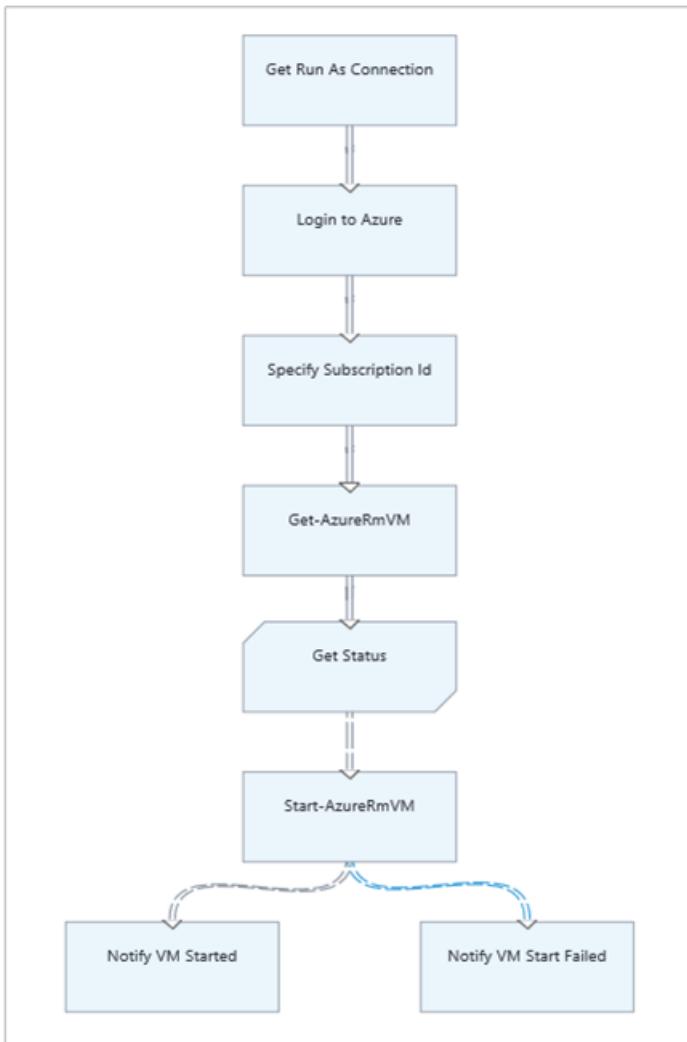
```
$StatusesJson = $ActivityOutput['Get-AzureRmVM'].StatusesText
$Statuses = ConvertFrom-Json $StatusesJson
$StatusOut =""
foreach ($Status in $Statuses){
    if($Status.Code -eq "Powerstate/running"){$StatusOut = "running"}
    elseif ($Status.Code -eq "Powerstate/deallocated") {$StatusOut = "stopped"}
}
$StatusOut
```

15. Create a link from **Get Status** to **Start-AzureRmVM**.



16. Select the link and in the Configuration pane, change **Apply condition** to **Yes**. Note the link turns to a dashed line indicating that the target activity will only be run if the condition resolves to true.
17. For the **Condition expression**, type `$ActivityOutput['Get Status'] -eq "Stopped"`. **Start-AzureRmVM** will now only run if the virtual machine is stopped.
18. In the Library control, expand **Cmdlets** and then **Microsoft.PowerShell.Utility**.

19. Add **Write-Output** to the canvas twice.



20. On the first **Write-Output** control, click **Parameters** and change the **Label** value to *Notify VM Started*.

21. For **InputObject**, change **Data source** to **PowerShell expression** and type in the expression "\$VMName successfully started."

22. On the second **Write-Output** control, click **Parameters** and change the **Label** value to *Notify VM Start Failed*.

23. For **InputObject**, change **Data source** to **PowerShell expression** and type in the expression "\$VMName could not start."

24. Create a link from **Start-AzureRmVM** to **Notify VM Started** and **Notify VM Start Failed**.

25. Select the link to **Notify VM Started** and change **Apply condition** to **True**.

26. For the **Condition expression**, type \$ActivityOutput['Start-AzureRmVM'].IsSuccessStatusCode -eq \$true. This Write-Output control will now only run if the virtual machine is successfully started.

27. Select the link to **Notify VM Start Failed** and change **Apply condition** to **True**.

28. For the **Condition expression**, type \$ActivityOutput['Start-AzureRmVM'].IsSuccessStatusCode -ne \$true. This Write-Output control will now only run if the virtual machine is not successfully started.

29. Save the runbook and open the Test pane.

30. Start the runbook with the virtual machine stopped, and it should start.

Next steps

- To learn more about Graphical Authoring, see [Graphical authoring in Azure Automation](#)
- To get started with PowerShell runbooks, see [My first PowerShell runbook](#)
- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)

My first PowerShell runbook

1/17/2017 • 7 min to read • [Edit on GitHub](#)

This tutorial walks you through the creation of a [PowerShell runbook](#) in Azure Automation. We'll start with a simple runbook that we'll test and publish while we explain how to track the status of the runbook job. Then we'll modify the runbook to actually manage Azure resources, in this case starting an Azure virtual machine. We'll then make the runbook more robust by adding runbook parameters.

Prerequisites

To complete this tutorial, you will need the following.

- Azure subscription. If you don't have one yet, you can [activate your MSDN subscriber benefits](#) or [\[sign up for a free account\]\(https://azure.microsoft.com/free/\)](#).
- [Automation account](#) to hold the runbook and authenticate to Azure resources. This account must have permission to start and stop the virtual machine.
- An Azure virtual machine. We will stop and start this machine so it should not be production.

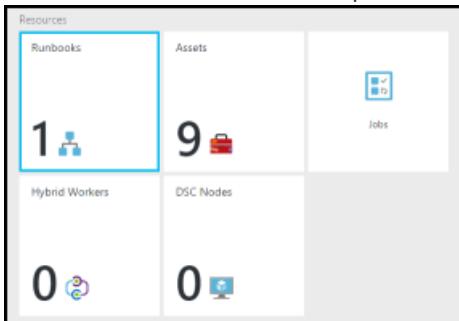
Step 1 - Create new runbook

We'll start by creating a simple runbook that outputs the text *Hello World*.

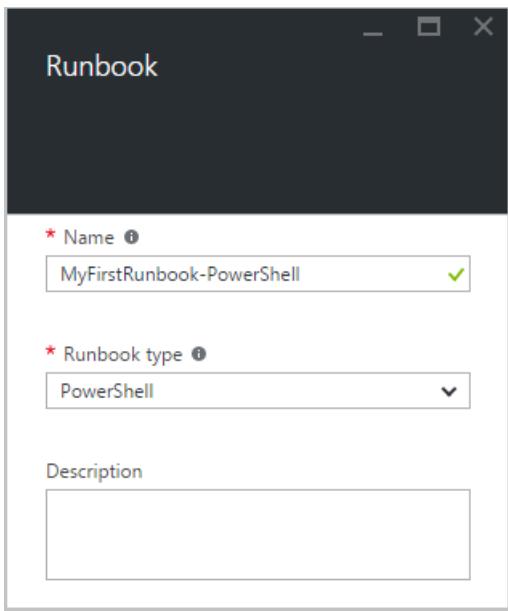
1. In the Azure Portal, open your Automation account.

The Automation account page gives you a quick view of the resources in this account. You should already have some Assets. Most of those are the modules that are automatically included in a new Automation account. You should also have the Credential asset that's mentioned in the [prerequisites](#).

2. Click on the **Runbooks** tile to open the list of runbooks.



3. Create a new runbook by clicking on the **Add a runbook** button and then **Create a new runbook**.
4. Give the runbook the name *MyFirstRunbook-PowerShell*.
5. In this case, we're going to create a [PowerShell runbook](#) so select **Powershell** for **Runbook type**.

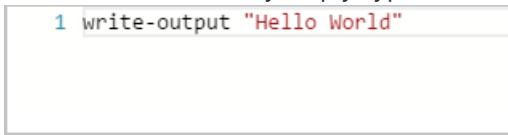


6. Click **Create** to create the runbook and open the textual editor.

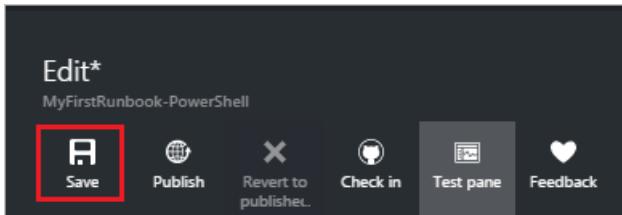
Step 2 - Add code to the runbook

You can either type code directly into the runbook, or you can select cmdlets, runbooks, and assets from the Library control and have them added to the runbook with any related parameters. For this walkthrough, we'll type directly into the runbook.

1. Our runbook is currently empty, type *Write-Output "Hello World."*.



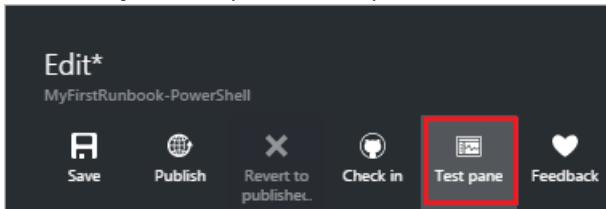
2. Save the runbook by clicking **Save**.



Step 3 - Test the runbook

Before we publish the runbook to make it available in production, we want to test it to make sure that it works properly. When you test a runbook, you run its **Draft** version and view its output interactively.

1. Click **Test pane** to open the Test pane.

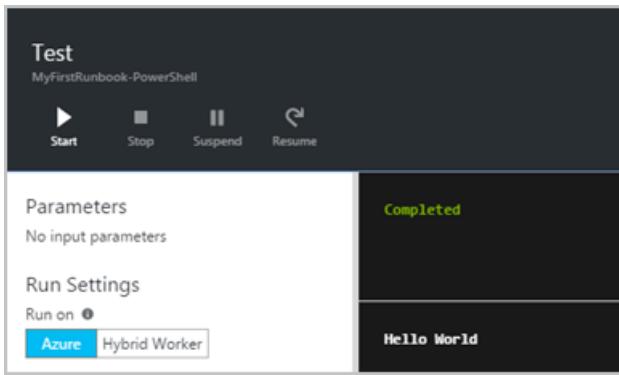


2. Click **Start** to start the test. This should be the only enabled option.

3. A [runbook job](#) is created and its status displayed.

The job status will start as *Queued* indicating that it is waiting for a runbook worker in the cloud to come available. It will then move to *Starting* when a worker claims the job, and then *Running* when the runbook actually starts running.

4. When the runbook job completes, its output is displayed. In our case, we should see *Hello World*

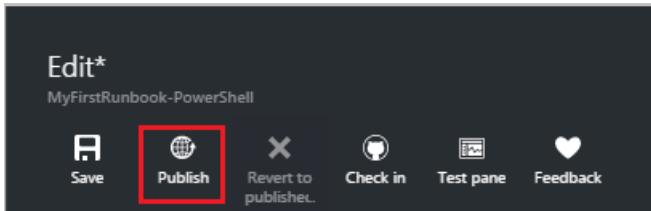


5. Close the Test pane to return to the canvas.

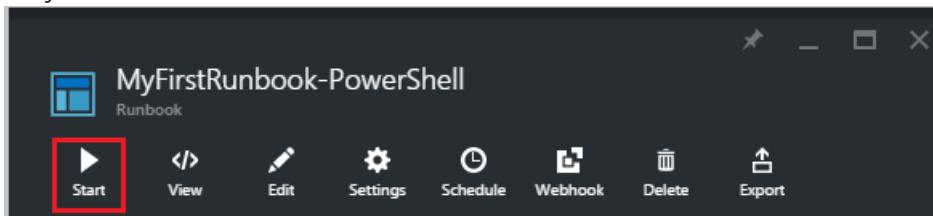
Step 4 - Publish and start the runbook

The runbook that we just created is still in Draft mode. We need to publish it before we can run it in production. When you publish a runbook, you overwrite the existing Published version with the Draft version. In our case, we don't have a Published version yet because we just created the runbook.

1. Click **Publish** to publish the runbook and then **Yes** when prompted.



2. If you scroll left to view the runbook in the **Runbooks** pane now, it will show an **Authoring Status** of **Published**.
3. Scroll back to the right to view the pane for **MyFirstRunbook-PowerShell**.
The options across the top allow us to start the runbook, view the runbook, schedule it to start at some time in the future, or create a [webhook](#) so it can be started through a HTTP call.
4. We just want to start the runbook so click **Start** and then click **Ok** when the Start Runbook blade opens.



5. A job pane is opened for the runbook job that we just created. We can close this pane, but in this case we'll leave it open so we can watch the job's progress.
6. The job status is shown in **Job Summary** and matches the statuses that we saw when we tested the runbook.

MyFirstRunbook-PowerShell 12/9/2015, 11:44 AM

Job

Resume Stop Suspend View source

Overview Add tiles +

Job Summary

Job ID: 4a138396-3db7-457b-82ee-cece8d72081e
Created: 12/9/2015, 11:44 AM
Last update: 12/9/2015, 11:44 AM
Ran on Azure

Completed

Runbook

INPUT Output

Status Add tiles +

Errors: 0 ✖ Warnings: 0 ⚠ All Logs

Exception

None

The screenshot shows the 'Overview' section of the Azure Runbook interface. A red box highlights the 'Job Summary' area, which displays the runbook's ID, creation date, last update, and the fact that it ran on Azure. Below this, a green checkmark indicates the job is 'Completed'. To the right, there are tiles for 'Runbook' (with a blue folder icon), 'INPUT' (with a blue square icon), and 'Output' (with a blue square and arrow icon). Below the summary, there are sections for 'Status' (Errors: 0, Warnings: 0), 'All Logs', and an 'Exception' section which is currently empty.

- Once the runbook status shows **Completed**, click **Output**. The Output pane is opened, and we can see our *Hello World*.

MyFirstRunbook-PowerShell 12/9/2015, 11:44 AM

Job

Resume Stop Suspend View source

Overview Add tiles +

Job Summary

Job ID: 4a138396-3db7-457b-82ee-cece8d72081e
Created: 12/9/2015, 11:44 AM
Last update: 12/9/2015, 11:44 AM
Ran on Azure

Completed

Runbook

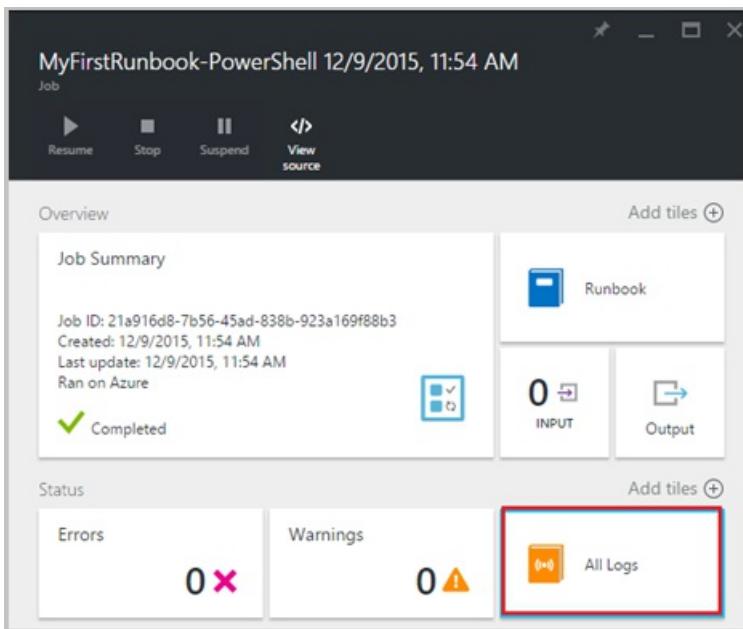
INPUT Output

Status Add tiles +

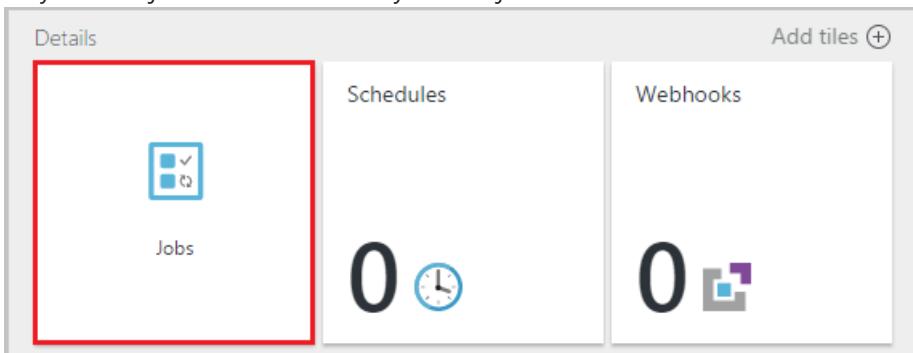
Errors: 0 ✖ Warnings: 0 ⚠ All Logs

The screenshot is identical to the previous one, showing the 'Overview' section of the Azure Runbook interface. However, the 'Output' tile in the 'Runbook' section is now highlighted with a red box, indicating it has been selected. The rest of the interface remains the same, showing a completed runbook job with no errors or warnings.

- Close the Output pane.
- Click **All Logs** to open the Streams pane for the runbook job. We should only see *Hello World* in the output stream, but this can show other streams for a runbook job such as Verbose and Error if the runbook writes to them.



10. Close the Streams pane and the Job pane to return to the MyFirstRunbook-PowerShell pane.
11. Click **Jobs** to open the Jobs pane for this runbook. This lists all of the jobs created by this runbook. We should only see one job listed since we only ran the job once.

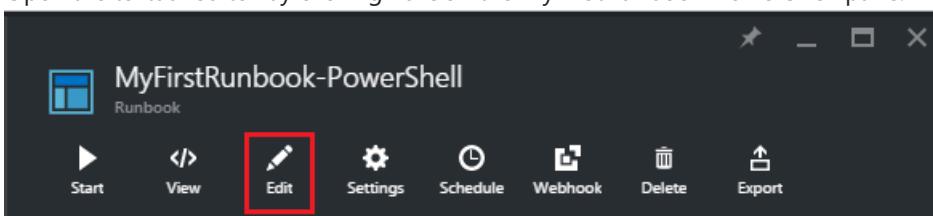


12. You can click on this job to open the same Job pane that we viewed when we started the runbook. This allows you to go back in time and view the details of any job that was created for a particular runbook.

Step 5 - Add authentication to manage Azure resources

We've tested and published our runbook, but so far it doesn't do anything useful. We want to have it manage Azure resources. It won't be able to do that though unless we have it authenticate using the credentials that are referred to in the [prerequisites](#). We do that with the **Add-AzureRmAccount** cmdlet.

1. Open the textual editor by clicking **Edit** on the MyFirstRunbook-PowerShell pane.



2. We don't need the **Write-Output** line anymore, so go ahead and delete it.
3. Type or copy and paste the following code that will handle the authentication with your Automation Run As account:

```
$Conn = Get-AutomationConnection -Name AzureRunAsConnection
Add-AzureRmAccount -ServicePrincipal -Tenant $Conn.TenantID `
```

4. Click **Test pane** so that we can test the runbook.
5. Click **Start** to start the test. Once it completes, you should receive output similar to the following, displaying basic information from your account. This confirms that the credential is valid.

The screenshot shows the 'Test pane' results. At the top, it says 'Completed'. Below that, under 'Environments', it lists several Azure environments: 'AzureCloud', 'AzureCloud', 'AzureChinaCloud', 'AzureChinaCloud', 'AzureUSGovernment', and 'AzureUSGovernment'. To the right, under 'Context', it shows the Microsoft Azure context.

```

Completed

Environments
-----
{[AzureCloud, AzureCloud], [AzureChinaCloud, AzureChinaCloud], [AzureUSGovernment, AzureUSGovernment]} Microsoft.Azur...

```

Step 6 - Add code to start a virtual machine

Now that our runbook is authenticating to our Azure subscription, we can manage resources. We'll add a command to start a virtual machine. You can pick any virtual machine in your Azure subscription, and for now we'll be hardcoding that name into the cmdlet.

1. After `Add-AzureRmAccount`, type `Start-AzureRmVM -Name 'VMName' -ResourceGroupName 'NameofResourceGroup'` providing the name and Resource Group name of the virtual machine to start.

```

$Conn = Get-AutomationConnection -Name AzureRunAsConnection
Add-AzureRmAccount -ServicePrincipal -Tenant $Conn.TenantID ` 
-ApplicationID $Conn.ApplicationID -CertificateThumbprint $Conn.CertificateThumbprint
Start-AzureRmVM -Name 'VMName' -ResourceGroupName 'ResourceGroupName'

```

2. Save the runbook and then click **Test pane** so that we can test it.
3. Click **Start** to start the test. Once it completes, check that the virtual machine was started.

Step 7 - Add an input parameter to the runbook

Our runbook currently starts the virtual machine that we hardcoded in the runbook, but it would be more useful if we could specify the virtual machine when the runbook is started. We will now add input parameters to the runbook to provide that functionality.

1. Add parameters for `VMName` and `ResourceGroupName` to the runbook and use these variables with the **Start-AzureRmVM** cmdlet as in the example below.

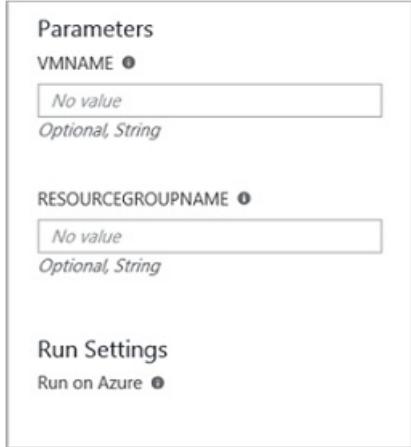
```

Param(
    [string]$VMName,
    [string]$ResourceGroupName
)
$Conn = Get-AutomationConnection -Name AzureRunAsConnection
Add-AzureRmAccount -ServicePrincipal -Tenant $Conn.TenantID ` 
-ApplicationID $Conn.ApplicationID -CertificateThumbprint $Conn.CertificateThumbprint
Start-AzureRmVM -Name $VMName -ResourceGroupName $ResourceGroupName

```

2. Save the runbook and open the Test pane. Note that you can now provide values for the two input variables that will be used in the test.

3. Close the Test pane.
4. Click **Publish** to publish the new version of the runbook.
5. Stop the virtual machine that you started in the previous step.
6. Click **Start** to start the runbook. Type in the **VMName** and **ResourceGroupName** for the virtual machine that you're going to start.



7. When the runbook completes, check that the virtual machine was started.

Differences from PowerShell Workflow

PowerShell runbooks have the same lifecycle, capabilities and management as PowerShell Workflow runbooks but there are some differences and limitations:

1. PowerShell runbooks run fast compared to PowerShell Workflow runbooks as they don't have compilation step.
2. PowerShell Workflow runbooks support checkpoints, using checkpoints, PowerShell Workflow runbooks can resume from any point in the runbook whereas PowerShell runbooks can only resume from the beginning.
3. PowerShell Workflow runbooks support parallel and serial execution whereas PowerShell runbooks can only execute commands serially.
4. In a PowerShell Workflow runbook, an activity, a command or a script block can have its own runspace whereas in a PowerShell runbook, everything in a script runs in a single runspace. There are also some [syntactic differences](#) between a native PowerShell runbook and a PowerShell Workflow runbook.

Next steps

- To get started with Graphical runbooks, see [My first graphical runbook](#)
- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)
- To know more about runbook types, their advantages and limitations, see [Azure Automation runbook types](#)
- For more information on PowerShell script support feature, see [Native PowerShell script support in Azure Automation](#)

My first PowerShell Workflow runbook

1/17/2017 • 7 min to read • [Edit on GitHub](#)

This tutorial walks you through the creation of a [PowerShell Workflow runbook](#) in Azure Automation. We'll start with a simple runbook that we'll test and publish while we explain how to track the status of the runbook job. Then we'll modify the runbook to actually manage Azure resources, in this case starting an Azure virtual machine. We'll then make the runbook more robust by adding runbook parameters.

Prerequisites

To complete this tutorial, you will need the following.

- Azure subscription. If you don't have one yet, you can [activate your MSDN subscriber benefits](#) or [\[sign up for a free account\]\(https://azure.microsoft.com/free/\)](#).
- [Automation account](#) to hold the runbook and authenticate to Azure resources. This account must have permission to start and stop the virtual machine.
- An Azure virtual machine. We will stop and start this machine so it should not be production.

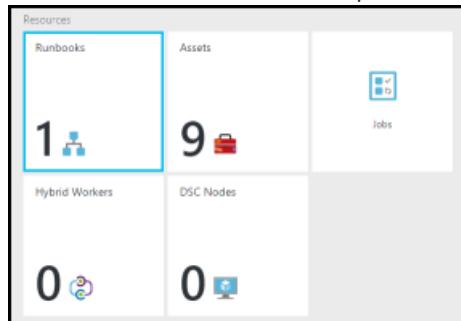
Step 1 - Create new runbook

We'll start by creating a simple runbook that outputs the text *Hello World*.

1. In the Azure Portal, open your Automation account.

The Automation account page gives you a quick view of the resources in this account. You should already have some Assets. Most of those are the modules that are automatically included in a new Automation account. You should also have the Credential asset that's mentioned in the [prerequisites](#).

2. Click on the **Runbooks** tile to open the list of runbooks.



3. Create a new runbook by clicking on the **Add a runbook** button and then **Create a new runbook**.
4. Give the runbook the name *MyFirstRunbook-Workflow*.
5. In this case, we're going to create a [PowerShell Workflow runbook](#) so select **Powershell Workflow** for **Runbook type**.

A screenshot of the 'Create a new runbook' dialog box. It has three main sections: 'Name' (with a placeholder 'Enter the runbook name...'), 'Description' (an empty text area), and 'Runbook type' (set to 'PowerShell Workflow' in a dropdown menu).

6. Click **Create** to create the runbook and open the textual editor.

Step 2 - Add code to the runbook

You can either type code directly into the runbook, or you can select cmdlets, runbooks, and assets from the Library control and have them added to the runbook with any related parameters. For this walkthrough, we'll type directly into the runbook.

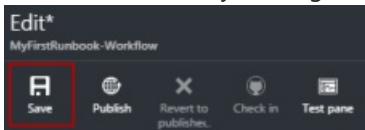
1. Our runbook is currently empty with only the required *workflow* keyword, the name of our runbook, and the braces that will encase the entire workflow.

```
Workflow MyFirstRunbook-Workflow
{
}
```

2. Type *Write-Output "Hello World."* between the braces.

```
Workflow MyFirstRunbook-Workflow
{
    Write-Output "Hello World"
}
```

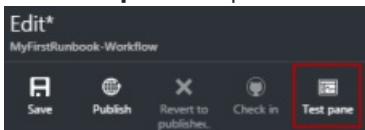
3. Save the runbook by clicking **Save**.



Step 3 - Test the runbook

Before we publish the runbook to make it available in production, we want to test it to make sure that it works properly. When you test a runbook, you run its **Draft** version and view its output interactively.

1. Click **Test pane** to open the Test pane.

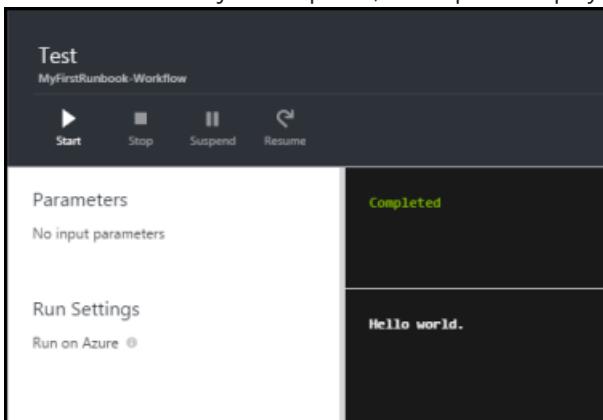


2. Click **Start** to start the test. This should be the only enabled option.

3. A **runbook job** is created and its status displayed.

The job status will start as *Queued* indicating that it is waiting for a runbook worker in the cloud to come available. It will then move to *Starting* when a worker claims the job, and then *Running* when the runbook actually starts running.

4. When the runbook job completes, its output is displayed. In our case, we should see *Hello World*.

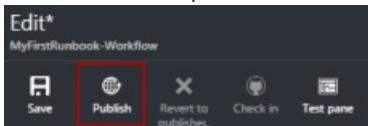


5. Close the Test pane to return to the canvas.

Step 4 - Publish and start the runbook

The runbook that we just created is still in Draft mode. We need to publish it before we can run it in production. When you publish a runbook, you overwrite the existing Published version with the Draft version. In our case, we don't have a Published version yet because we just created the runbook.

1. Click **Publish** to publish the runbook and then **Yes** when prompted.

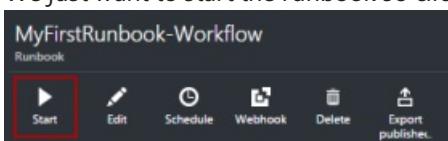


2. If you scroll left to view the runbook in the **Runbooks** pane now, it will show an **Authoring Status** of **Published**.

3. Scroll back to the right to view the pane for **MyFirstRunbook-Workflow**.

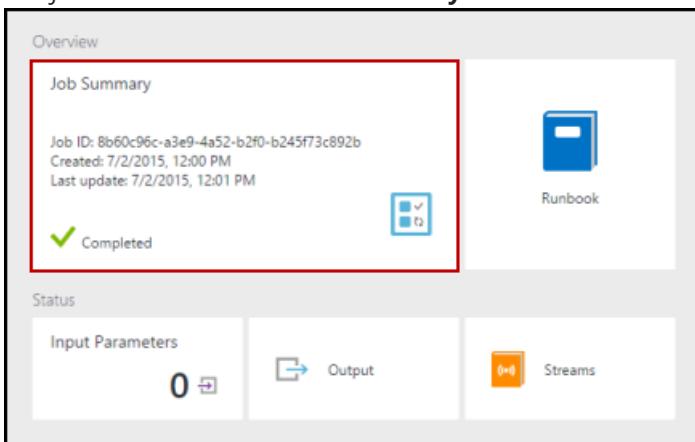
The options across the top allow us to start the runbook, schedule it to start at some time in the future, or create a [webhook](#) so it can be started through an HTTP call.

4. We just want to start the runbook so click **Start** and then **Yes** when prompted.

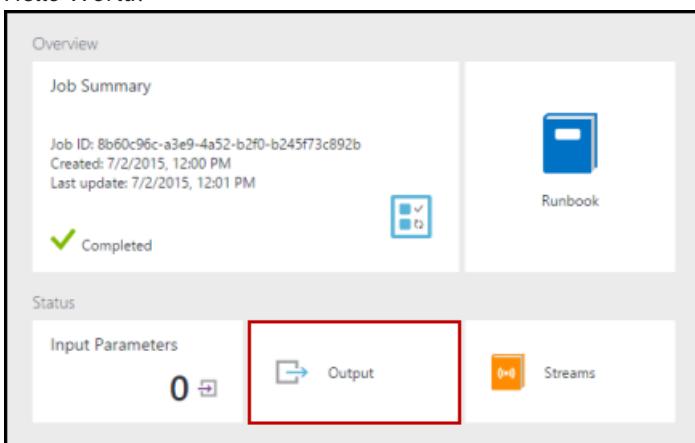


5. A job pane is opened for the runbook job that we just created. We can close this pane, but in this case we'll leave it open so we can watch the job's progress.

6. The job status is shown in **Job Summary** and matches the statuses that we saw when we tested the runbook.

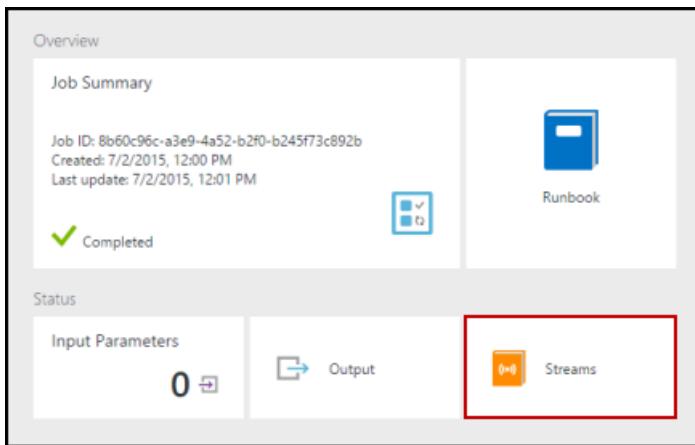


7. Once the runbook status shows *Completed*, click **Output**. The Output pane is opened, and we can see our *Hello World*.

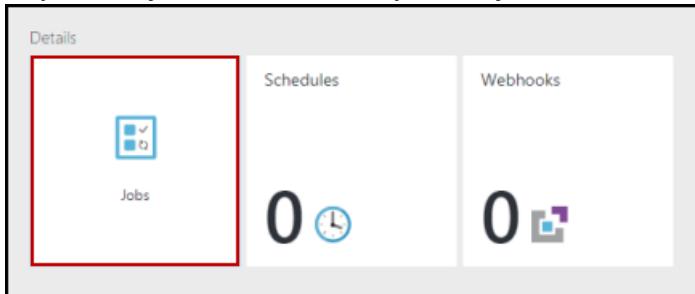


8. Close the Output pane.

9. Click **Streams** to open the Streams pane for the runbook job. We should only see *Hello World* in the output stream, but this can show other streams for a runbook job such as Verbose and Error if the runbook writes to them.



10. Close the Streams pane and the Job pane to return to the MyFirstRunbook pane.
11. Click **Jobs** to open the Jobs pane for this runbook. This lists all of the jobs created by this runbook. We should only see one job listed since we only ran the job once.

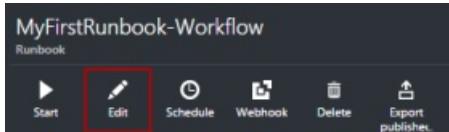


12. You can click on this job to open the same Job pane that we viewed when we started the runbook. This allows you to go back in time and view the details of any job that was created for a particular runbook.

Step 5 - Add authentication to manage Azure resources

We've tested and published our runbook, but so far it doesn't do anything useful. We want to have it manage Azure resources. It won't be able to do that though unless we have it authenticate using the credentials that are referred to in the [prerequisites](#). We do that with the **Add-AzureRMAccount** cmdlet.

1. Open the textual editor by clicking **Edit** on the MyFirstRunbook-Workflow pane.



2. We don't need the **Write-Output** line anymore, so go ahead and delete it.
3. Position the cursor on a blank line between the braces.
4. Type or copy and paste the following code that will handle the authentication with your Automation Run As account:

```
$Conn = Get-AutomationConnection -Name AzureRunAsConnection
Add-AzureRMAccount -ServicePrincipal -Tenant $Conn.TenantID `
```

```
-ApplicationId $Conn.ApplicationID -CertificateThumbprint $Conn.CertificateThumbprint
```

5. Click **Test pane** so that we can test the runbook.
6. Click **Start** to start the test. Once it completes, you should receive output similar to the following, displaying basic information from your account. This confirms that the credential is valid.

```
Completed

Environments ----- {[AzureCloud, AzureCloud], [AzureChinaCloud, AzureChinaCloud], [AzureUSGovernment, AzureUSGovernment]} Microsoft.Azur...
Context -----
```

Step 6 - Add code to start a virtual machine

Now that our runbook is authenticating to our Azure subscription, we can manage resources. We'll add a command to start a virtual machine. You can pick any virtual machine in your Azure subscription, and for now we'll be hardcoding that name into the cmdlet.

1. After `Add-AzureRmAccount`, type `Start-AzureRmVM -Name 'VMName' -ResourceGroupName 'NameofResourceGroup'` providing the name and Resource Group name of the virtual machine to start.

```
workflow MyFirstRunbook-Workflow
{
    $Conn = Get-AutomationConnection -Name AzureRunAsConnection
    Add-AzureRMAccount -ServicePrincipal -Tenant $Conn.TenantID -ApplicationId $Conn.ApplicationID -
    CertificateThumbprint $Conn.CertificateThumbprint
    Start-AzureRmVM -Name 'VMName' -ResourceGroupName 'ResourceGroupName'
}
```

2. Save the runbook and then click **Test pane** so that we can test it.
3. Click **Start** to start the test. Once it completes, check that the virtual machine was started.

Step 7 - Add an input parameter to the runbook

Our runbook currently starts the virtual machine that we hardcoded in the runbook, but it would be more useful if we could specify the virtual machine when the runbook is started. We will now add input parameters to the runbook to provide that functionality.

1. Add parameters for `VMName` and `ResourceGroupName` to the runbook and use these variables with the **Start-AzureRmVM** cmdlet as in the example below.

```
workflow MyFirstRunbook-Workflow
{
    Param(
        [string]$VMName,
        [string]$ResourceGroupName
    )
    $Conn = Get-AutomationConnection -Name AzureRunAsConnection
    Add-AzureRMAccount -ServicePrincipal -Tenant $Conn.TenantID -ApplicationId $Conn.ApplicationID -
    CertificateThumbprint $Conn.CertificateThumbprint
    Start-AzureRmVM -Name $VMName -ResourceGroupName $ResourceGroupName
}
```

2. Save the runbook and open the Test pane. Note that you can now provide values for the two input variables that will be used in the test.
3. Close the Test pane.
4. Click **Publish** to publish the new version of the runbook.
5. Stop the virtual machine that you started in the previous step.

6. Click **Start** to start the runbook. Type in the **VMName** and **ResourceGroupName** for the virtual machine that you're going to start.

The screenshot shows the 'Parameters' section of a runbook configuration. It contains two parameters: 'VMNAME' and 'RESOURCEGROUPNAME', both of which have their values set to 'No value'. Below the parameters is a 'Run Settings' section where it is specified that the runbook should 'Run on Azure'.

Parameter	Type	Value
VMNAME	String	No value
RESOURCEGROUPNAME	String	No value

Run Settings
Run on Azure

7. When the runbook completes, check that the virtual machine was started.

Next steps

- To get started with Graphical runbooks, see [My first graphical runbook](#)
- To get started with PowerShell runbooks, see [My first PowerShell runbook](#)
- To learn more about runbook types, their advantages and limitations, see [Azure Automation runbook types](#)
- For more information on PowerShell script support feature, see [Native PowerShell script support in Azure Automation](#)

Role-based access control in Azure Automation

1/17/2017 • 8 min to read • [Edit on GitHub](#)

Role-based access control

Role-based access control (RBAC) enables access management for Azure resources. Using [RBAC](#), you can segregate duties within your team and grant only the amount of access to users, groups and applications that they need to perform their jobs. Role-based access can be granted to users using the Azure portal, Azure Command-Line tools or Azure Management APIs.

RBAC in Automation Accounts

In Azure Automation, access is granted by assigning the appropriate RBAC role to users, groups, and applications at the Automation account scope. Following are the built-in roles supported by an Automation account:

ROLE	DESCRIPTION
Owner	The Owner role allows access to all resources and actions within an Automation account including providing access to other users, groups and applications to manage the Automation account.
Contributor	The Contributor role allows you to manage everything except modifying other user's access permissions to an Automation account.
Reader	The Reader role allows you to view all the resources in an Automation account but cannot make any changes.
Automation Operator	The Automation Operator role allows you to perform operational tasks such as start, stop, suspend, resume and schedule jobs. This role is helpful if you want to protect your Automation Account resources like credentials assets and runbooks from being viewed or modified but still allow members of your organization to execute these runbooks.
User Access Administrator	The User Access Administrator role allows you to manage user access to Azure Automation accounts.

NOTE

You cannot grant access rights to a specific runbook or runbooks, only to the resources and actions within the Automation account.

In this article we will walk you through how to set up RBAC in Azure Automation. But first, let's take a closer look at the individual permissions granted to the Contributor, Reader, Automation Operator and User Access Administrator so that we gain a good understanding before granting anyone rights to the Automation account. Otherwise it could result in unintended or undesirable consequences.

Contributor role permissions

The following table presents the specific actions that can be performed by the Contributor role in Automation.

RESOURCE TYPE	READ	WRITE	DELETE	OTHER ACTIONS
Azure Automation Account	✓	✓	✓	
Automation Certificate Asset	✓	✓	✓	
Automation Connection Asset	✓	✓	✓	
Automation Connection Type Asset	✓	✓	✓	
Automation Credential Asset	✓	✓	✓	
Automation Schedule Asset	✓	✓	✓	
Automation Variable Asset	✓	✓	✓	
Automation Desired State Configuration				✓
Hybrid Runbook Worker Resource Type	✓		✓	
Azure Automation Job	✓	✓		✓
Automation Job Stream	✓			
Automation Job Schedule	✓	✓	✓	
Automation Module	✓	✓	✓	

RESOURCE TYPE	READ	WRITE	DELETE	OTHER ACTIONS
Azure Automation Runbook	✓	✓	✓	✓
Automation Runbook Draft	✓			✓
Automation Runbook Draft Test Job	✓	✓		✓
Automation Webhook	✓	✓	✓	✓

Reader role permissions

The following table presents the specific actions that can be performed by the Reader role in Automation.

RESOURCE TYPE	READ	WRITE	DELETE	OTHER ACTIONS
Classic subscription administrator	✓			
Management lock	✓			
Permission	✓			
Provider operations	✓			
Role assignment	✓			
Role definition	✓			

Automation Operator role permissions

The following table presents the specific actions that can be performed by the Automation Operator role in Automation.

RESOURCE TYPE	READ	WRITE	DELETE	OTHER ACTIONS
Azure Automation Account				
Automation Certificate Asset				
Automation Connection Asset				
Automation Connection Type Asset				
Automation Credential Asset				
Automation Schedule Asset				
Automation Variable Asset				
Automation Desired State Configuration				
Hybrid Runbook Worker Resource Type				
Azure Automation Job				
Automation Job Stream				
Automation Job Schedule				
Automation Module				
Azure Automation Runbook				
Automation Runbook Draft				
Automation Runbook Draft Test Job				

RESOURCE TYPE	READ	WRITE	DELETE	OTHER ACTIONS
Automation Webhook				

For further details, the [Automation operator actions](#) lists the actions supported by the Automation operator role on the Automation account and its resources.

User Access Administrator role permissions

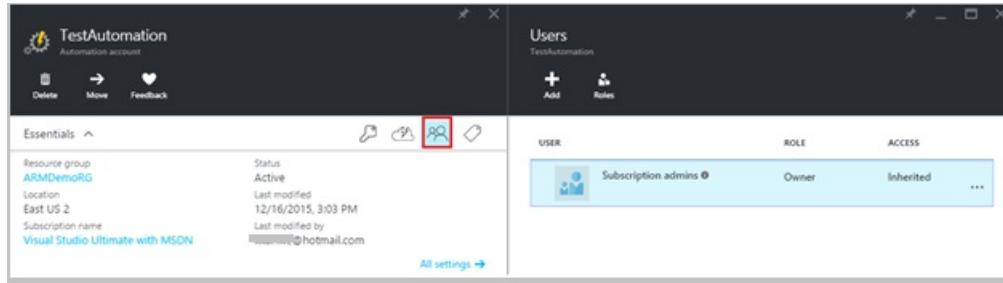
The following table presents the specific actions that can be performed by the User Access Administrator role in Automation.

RESOURCE TYPE	READ	WRITE	DELETE	OTHER ACTIONS
Azure Automation Account	✓			
Automation Certificate Asset	✓			
Automation Connection Asset	✓			
Automation Connection Type Asset	✓			
Automation Credential Asset	✓			
Automation Schedule Asset	✓			
Automation Variable Asset	✓			
Automation Desired State Configuration				
Hybrid Runbook Worker Resource Type	✓			
Azure Automation Job	✓			

RESOURCE TYPE	READ	WRITE	DELETE	OTHER ACTIONS
Automation Job Stream	✓			
Automation Job Schedule	✓			
Automation Module	✓			
Azure Automation Runbook	✓			
Automation Runbook Draft	✓			
Automation Runbook Draft Test Job	✓			
Automation Webhook	✓			

Configure RBAC for your Automation Account using Azure Portal

1. Log in to the [Azure Portal](#) and open your Automation account from the Automation Accounts blade.
2. Click on the **Access** control at the top right corner. This opens the **Users** blade where you can add new users, groups and applications to manage your Automation account and view existing roles that can be configured for the Automation Account.

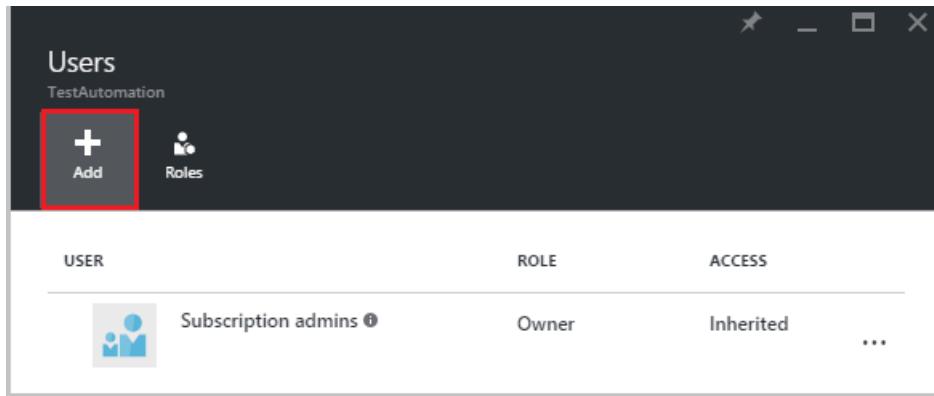


NOTE

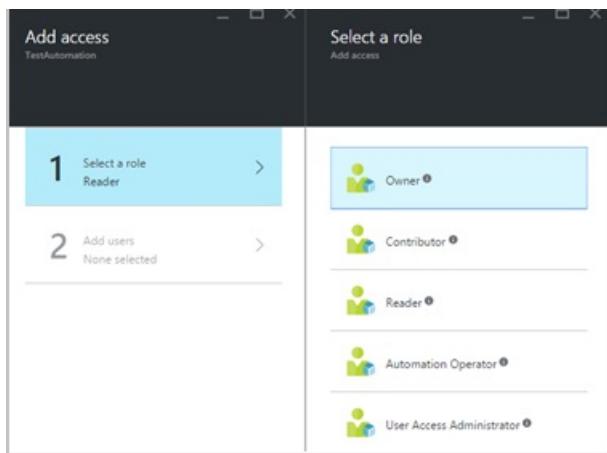
Subscription admins already exists as the default user. The subscription admins active directory group includes the service administrator(s) and co-administrator(s) for your Azure subscription. The Service admin is the owner of your Azure subscription and its resources, and will have the owner role inherited for the automation accounts too. This means that the access is **Inherited** for **service administrators and co-admins** of a subscription and it's **Assigned** for all the other users. Click **Subscription admins** to view more details about their permissions.

Add a new user and assign a role

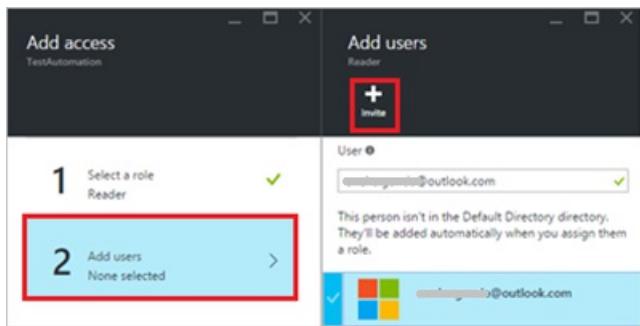
1. From the Users blade, click **Add** to open the **Add access blade** where you can add a user, group, or application, and assign a role to them.



2. Select a role from the list of available roles. We will choose the **Reader** role, but you can choose any of the available built-in roles that an Automation Account supports or any custom role you may have defined.



3. Click on **Add users** to open the **Add users** blade. If you have added any users, groups, or applications to manage your subscription then those users are listed and you can select them to add access. If there aren't any users listed, or if the user you are interested in adding is not listed then click **invite** to open the **Invite a guest** blade, where you can invite a user with a valid Microsoft account email address such as Outlook.com, OneDrive, or Xbox Live Ids. Once you have entered the email address of the user, click **Select** to add the user, and then click **OK**.



Now you should see the user added to the **Users** blade with the **Reader** role assigned.

USER	ROLE	ACCESS
Subscription admins	Owner	Inherited
[User icon]	Reader	Assigned

You can also assign a role to the user from the **Roles** blade.

4. Click **Roles** from the Users blade to open the **Roles blade**. From this blade, you can view the name of the role, the number of users and groups assigned to that role.

NAME	USERS	GROUPS
Owner	0	1
Contributor	0	0
Reader	0	0
Automation Operator	0	0
User Access Administrator	0	0

NOTE

Role-based access control can only be set at the Automation Account level and not at any resource below the Automation Account.

You can assign more than one role to a user, group, or application. For example, if we add the **Automation Operator** role along with the **Reader role** to the user, then they can view all the Automation resources, as well as execute the runbook jobs. You can expand the dropdown to view a list of roles assigned to the user.

USER	ROLE	ACCESS
Subscription admins	Owner	Inherited
[User icon]	Reader, Automation Operator	Assigned
[User icon]	Reader	Assigned
[User icon]	Automation Operator	Assigned
[User icon]	Subscription admins	Owner

Remove a user

You can remove the access permission for a user who is not managing the Automation Account, or who no longer works for the organization. Following are the steps to remove a user:

1. From the **Users** blade, select the role assignment that you wish to remove.
2. Click the **Remove** button in the assignment details blade.
3. Click **Yes** to confirm removal.

The screenshot shows two windows side-by-side. The left window is titled 'Users' under 'TestAutomation' and lists two users: 'Sneha Gunda' (outlook.com) with 'Reader' role and 'Assigned' access, and 'Subscription admins' with 'Owner' role and 'Inherited' access. The right window is titled 'Reader - TestAutomation' for the user 'Sneha Gunda'. It shows the 'Remove' button highlighted with a red box, and the 'Properties' section which includes 'Role' (Reader), 'Assigned To' (Sneha Gunda), and 'ACCESS' (Assigned).

Role Assigned User

When a user assigned to a role logs in to their Automation account, they can now see the owner's account listed in the list of **Default Directories**. In order to view the Automation account that they have been added to, they must switch the default directory to the owner's default directory.

The screenshot shows the user profile for 'Sneha Gunda' with the 'DEFAULT DIRECTORY (SNEHA...)' dropdown menu open. The 'Default Directory' option is highlighted with a red box. Other options visible are 'MyDirectory'.

User experience for Automation operator role

When a user, who is assigned to the Automation Operator role views the Automation account they are assigned to, they can only view the list of runbooks, runbook jobs and schedules created in the Automation account but can't view their definition. They can start, stop, suspend, resume or schedule the runbook job. The user will not have access to other Automation resources such as configurations, hybrid worker groups or DSC nodes.

The screenshot shows the 'Resources' blade in the Azure portal. It displays tiles for 'Runbooks' (23), 'Assets' (8), 'DSC Configurations' (No access), 'Jobs' (No access highlighted with a red box), 'Hybrid Worker Groups' (No access), and 'DSC Nodes' (No access). The 'Monitoring' tab is selected at the bottom.

When the user clicks on the runbook, the commands to view the source or edit the runbook are not provided as the Automation operator role doesn't allow access to them.

The screenshot shows a 'HybridRunbook' details page. The top navigation bar includes 'Start', 'Settings', 'Schedule', and 'Export' buttons. The main content area is currently empty.

The user will have access to view and to create schedules but will not have access to any other asset type.

The screenshot shows the 'Assets' blade in the Azure portal. It contains six tiles:

- Schedules**: Shows 8 items.
- Modules**: Shows 'No access'.
- Certificates**: Shows 'No access'.
- Connections**: Shows 'No access'.
- Variables**: Shows 'No access'.
- Credentials**: Shows 'No access'.

At the bottom right of the blade, there is a 'Add tiles +' button and a 'Add a group +' button.

This user also doesn't have access to view the webhooks associated with a runbook

The screenshot shows the 'Runbooks' blade in the Azure portal. It contains three tiles:

- Details**: Shows 'Jobs 0'.
- Schedules**: Shows 'No access'.
- Webhooks**: Shows 'No access'.

Configure RBAC for your Automation Account using Azure PowerShell

Role-based access can also be configured to an Automation Account using the following [Azure PowerShell cmdlets](#).

- [Get-AzureRmRoleDefinition](#) lists all RBAC roles that are available in Azure Active Directory. You can use this command along with the **Name** property to list all the actions that can be performed by a specific role.

Example:

```
PS C:\Users\sngun> Get-AzureRmRoleDefinition -Name "Automation Operator"

Name          : Automation Operator
Id            : d3881f73-407a-4167-8283-e981cbba0404
IsCustom      : False
Description   : Automation Operators are able to start, stop, suspend, and resume jobs
Actions       : {Microsoft.Authorization/*/read, Microsoft.Automation/automationAccounts/jobs/read,
               Microsoft.Automation/automationAccounts/jobs/resume/action,
               Microsoft.Automation/automationAccounts/jobs/stop/action...}
NotActions    : {}
AssignableScopes : {}
```

- [Get-AzureRmRoleAssignment](#) lists Azure AD RBAC role assignments at the specified scope. Without any parameters, this command returns all the role assignments made under the subscription. Use the **ExpandPrincipalGroups** parameter to list access assignments for the specified user as well as the groups the user is a member of.

Example: Use the following command to list all the users and their roles within an automation account.

```
Get-AzureRMRoleAssignment -scope "/subscriptions/<SubscriptionID>/resourcegroups/<Resource Group Name>/Providers/Microsoft.Automation/automationAccounts/<Automation Account Name>"
```

```
RoleAssignmentId : /subscriptions/3f646aad-f988-40d5-aaf4-704239578f3c/providers/Microsoft.Authorization/roleAssignments/80089d58-ae91-41bb-8a87-969554104cda
Scope           : /subscriptions/3f646aad-f988-40d5-aaf4-704239578f3c
DisplayName     : [REDACTED]@outlook.com
SignInName      : [REDACTED]@outlook.com
RoleDefinitionName : Contributor
RoleDefinitionId : b24988ac-6180-42a0-ab88-20f7382dd24c
ObjectId        : 5c786adc-e6cd-4201-b9fc-b6054ca43491
ObjectType       : User
```

- [New-AzureRmRoleAssignment](#) to assign access to users, groups and applications to a particular scope.

Example: Use the following command to assign the "Automation Operator" role for a user in the Automation Account scope.

```
New-AzureRmRoleAssignment -SignInName <sign-in Id of a user you wish to grant access> -RoleDefinitionName "Automation operator" -Scope "/subscriptions/<SubscriptionID>/resourcegroups/<Resource Group Name>/Providers/Microsoft.Automation/automationAccounts/<Automation Account Name>"
```

```
RoleAssignmentId : /subscriptions/3f646aad-f988-40d5-aaf4-704239578f3c/resourcegroups/TestRG/providers/Microsoft.Automation/automationAccounts/TestAutomation/providers/Microsoft.Authorization/roleAssignments/057cb4d1-5369-4bc
Scope           : /subscriptions/3f646aad-f988-40d5-aaf4-704239578f3c/resourcegroups/TestRG/providers/Microsoft.Automation/automationAccounts/TestAutomation
DisplayName     : [REDACTED]@outlook.com
SignInName      : [REDACTED]@outlook.com
RoleDefinitionName : Automation Operator
RoleDefinitionId : d3881f73-407a-4167-8283-e981cbb0404
ObjectId        : 5c786adc-e6cd-4201-b9fc-b6054ca43491
ObjectType       : User
```

- Use [Remove-AzureRmRoleAssignment](#) to remove access of a specified user, group or application from a particular scope.

Example: Use the following command to remove the user from the "Automation Operator" role in the Automation Account scope.

```
Remove-AzureRmRoleAssignment -SignInName <sign-in Id of a user you wish to remove> -RoleDefinitionName "Automation Operator" -Scope "/subscriptions/<SubscriptionID>/resourcegroups/<Resource Group Name>/Providers/Microsoft.Automation/automationAccounts/<Automation Account Name>"
```

In the above examples, replace **sign in Id**, **subscription Id**, **resource group name** and **Automation account name** with your account details. Choose **yes** when prompted to confirm before continuing to remove user role assignment.

Next Steps

- For information on different ways to configure RBAC for Azure Automation, refer to [manage RBAC with Azure PowerShell](#).
- For details on different ways to start a runbook, see [Starting a runbook](#)
- For information about different runbook types, refer to [Azure Automation runbook types](#)

Azure Automation runbook types

1/17/2017 • 3 min to read • [Edit on GitHub](#)

Azure Automation supports four types of runbooks that are briefly described in the following table. The sections below provide further information about each type including considerations on when to use each.

Type	Description
Graphical	Based on Windows PowerShell and created and edited completely in graphical editor in Azure portal.
Graphical PowerShell Workflow	Based on Windows PowerShell Workflow and created and edited completely in the graphical editor in Azure portal.
PowerShell	Text runbook based on Windows PowerShell script.
PowerShell Workflow	Text runbook based on Windows PowerShell Workflow.

Graphical runbooks

Graphical and Graphical PowerShell Workflow runbooks are created and edited with the graphical editor in the Azure portal. You can export them to a file and then import them into another automation account, but you cannot create or edit them with another tool. Graphical runbooks generate PowerShell code, but you can't directly view or modify the code. Graphical runbooks cannot be converted to one of the [text formats](#), nor can a text runbook be converted to graphical format. Graphical runbooks can be converted to Graphical PowerShell Workflow runbooks during import and vice-versa.

Advantages

- Visual insert-link-configure authoring model
- Focus on how data flows through the process
- Visually represent management processes
- Include other runbooks as child runbooks to create high level workflows
- Encourages modular programming

Limitations

- Can't edit runbook outside of Azure portal.
- May require a Code activity containing PowerShell code to perform complex logic.
- Can't view or directly edit the PowerShell code that is created by the graphical workflow. Note that you can view the code you create in any Code activities.

PowerShell runbooks

PowerShell runbooks are based on Windows PowerShell. You directly edit the code of the runbook using the text editor in the Azure portal. You can also use any offline text editor and [import the runbook](#) into Azure Automation.

Advantages

- Implement all complex logic with PowerShell code without the additional complexities of PowerShell Workflow.

- Runbook starts faster than Graphical or PowerShell Workflow runbooks since it doesn't need to be compiled before running.

Limitations

- Must be familiar with PowerShell scripting.
- Can't use [parallel processing](#) to perform multiple actions in parallel.
- Can't use [checkpoints](#) to resume runbook in case of error.
- PowerShell Workflow runbooks and Graphical runbooks can only be included as child runbooks by using the `Start-AzureAutomationRunbook` cmdlet which creates a new job.

Known Issues

Following are current known issues with PowerShell runbooks.

- PowerShell runbooks cannot retrieve an unencrypted [variable asset](#) with a null value.
- PowerShell runbooks cannot retrieve a [variable asset](#) with ~ in the name.
- `Get-Process` in a loop in a PowerShell runbook may crash after about 80 iterations.
- A PowerShell runbook may fail if it attempts to write a very large amount of data to the output stream at once. You can typically work around this issue by outputting just the information you need when working with large objects. For example, instead of outputting something like `Get-Process`, you can output just the required fields with `Get-Process | Select ProcessName, CPU`.

PowerShell Workflow runbooks

PowerShell Workflow runbooks are text runbooks based on [Windows PowerShell Workflow](#). You directly edit the code of the runbook using the text editor in the Azure portal. You can also use any offline text editor and [import the runbook](#) into Azure Automation.

Advantages

- Implement all complex logic with PowerShell Workflow code.
- Use [checkpoints](#) to resume runbook in case of error.
- Use [parallel processing](#) to perform multiple actions in parallel.
- Can include other Graphical runbooks and PowerShell Workflow runbooks as child runbooks to create high level workflows.

Limitations

- Author must be familiar with PowerShell Workflow.
- Runbook must deal with the additional complexity of PowerShell Workflow such as [deserialized objects](#).
- Runbook takes longer to start than PowerShell runbooks since it needs to be compiled before running.
- PowerShell runbooks can only be included as child runbooks by using the `Start-AzureAutomationRunbook` cmdlet which creates a new job.

Considerations

You should take into account the following additional considerations when determining which type to use for a particular runbook.

- You can't convert runbooks from graphical to textual type or vice-versa.
- There are limitations using runbooks of different types as a child runbook. See [Child runbooks in Azure Automation](#) for more information.

Next steps

- To learn more about Graphical runbook authoring, see [Graphical authoring in Azure Automation](#)

- To understand the differences between PowerShell and PowerShell workflows for runbooks, see [Learning Windows PowerShell Workflow](#)
- For more information on how to create or import a Runbook, see [Creating or Importing a Runbook](#)

Creating or importing a runbook in Azure Automation

1/17/2017 • 5 min to read • [Edit on GitHub](#)

You can add a runbook to Azure Automation by either [creating a new one](#) or by importing an existing runbook from a file or from the [Runbook Gallery](#). This article provides information on creating and importing runbooks from a file. You can get all of the details on accessing community runbooks and modules in [Runbook and module galleries for Azure Automation](#).

Creating a new runbook

You can create a new runbook in Azure Automation using one of the Azure portals or Windows PowerShell. Once the runbook has been created, you can edit it using information in [Learning PowerShell Workflow](#) and [Graphical authoring in Azure Automation](#).

To create a new Azure Automation runbook with the Azure Classic portal

You can only work with [PowerShell Workflow runbooks](#) in the Azure portal.

1. In the Azure Classic portal, click, **New, App Services, Automation, Runbook, Quick Create**.
2. Enter the required information, and then click **Create**. The runbook name must start with a letter and can have letters, numbers, underscores, and dashes.
3. If you want to edit the runbook now, then click **Edit Runbook**. Otherwise, click **OK**.
4. Your new runbook will appear on the **Runbooks** tab.

To create a new Azure Automation runbook with the Azure portal

1. In the Azure portal, open your Automation account.
2. Click on the **Runbooks** tile to open the list of runbooks.
3. Click on the **Add a runbook** button and then **Create a new runbook**.
4. Type a **Name** for the runbook and select its **Type**. The runbook name must start with a letter and can have letters, numbers, underscores, and dashes.
5. Click **Create** to create the runbook and open the editor.

To create a new Azure Automation runbook with Windows PowerShell

You can use the `New-AzureRmAutomationRunbook` cmdlet to create an empty [PowerShell Workflow runbook](#).

You can either specify the **Name** parameter to create an empty runbook that you can later edit, or you can specify the **Path** parameter to import a runbook file. The **Type** parameter should also be included to specify one of the four runbook types.

The following sample commands show how to create a new empty runbook.

```
New-AzureRmAutomationRunbook -AutomationAccountName MyAccount `  
-Name NewRunbook -ResourceGroupName MyResourceGroup -Type PowerShell
```

Importing a runbook from a file into Azure Automation

You can create a new runbook in Azure Automation by importing a PowerShell script or PowerShell Workflow (.ps1 extension) or an exported graphical runbook (.graphrunbook). You must specify the [type of runbook](#) that will be created from the import taking into account the following considerations.

- A .graphrunbook file may only be imported into a new [graphical runbook](#), and graphical runbooks can only be created from a .graphrunbook file.
- A .ps1 file containing a PowerShell Workflow can only be imported into a [PowerShell Workflow runbook](#). If the file contains multiple PowerShell Workflows, then the import will fail. You must save each workflow to its own file and import each separately.
- A .ps1 file that does not contain a workflow can be imported into either a [PowerShell runbook](#) or a [PowerShell Workflow runbook](#). If it is imported into a PowerShell Workflow runbook, then it will be converted to a workflow, and comments will be included in the runbook specifying the changes that were made.

To import a runbook from a file with the Azure Classic portal

You can use the following procedure to import a script file into Azure Automation. Note that you can only import a .ps1 file into a PowerShell Workflow runbook using this portal. You must use the Azure portal for other types.

1. In the Azure Management portal, select **Automation** and then select an Automation Account.
2. Click **Import**.
3. Click **Browse for File** and locate the script file to import.
4. If you want to edit the runbook now, then click **Edit Runbook**. Otherwise, click OK.
5. The new runbook will appear on the **Runbooks** tab for the Automation Account.
6. You must [publish the runbook](#) before you can run it.

To import a runbook from a file with the Azure portal

You can use the following procedure to import a script file into Azure Automation.

NOTE

Note that you can only import a .ps1 file into a PowerShell Workflow runbook using the portal.

1. In the Azure portal, open your Automation account.
2. Click on the **Runbooks** tile to open the list of runbooks.
3. Click on the **Add a runbook** button and then **Import**.
4. Click **Runbook file** to select the file to import
5. If the **Name** field is enabled, then you have the option to change it. The runbook name must start with a letter and can have letters, numbers, underscores, and dashes.
6. The [runbook type](#) will be automatically selected, but you can change the type after taking the applicable restrictions into account.
7. The new runbook will appear in the list of runbooks for the Automation Account.
8. You must [publish the runbook](#) before you can run it.

NOTE

After you import a graphical runbook or a graphical PowerShell workflow runbook, you have the option to convert to the other type if wanted. You can't convert to textual.

To import a runbook from a script file with Windows PowerShell

You can use the [Import-AzureRMAutomationRunbook](#) cmdlet to import a script file as a draft PowerShell Workflow runbook. If the runbook already exists, the import will fail unless you use the *-Force* parameter.

The following sample commands show how to import a script file into a runbook.

```
$automationAccountName = "AutomationAccount"
$runbookName = "Sample_TestRunbook"
$scriptPath = "C:\Runbooks\Sample_TestRunbook.ps1"
$RGName = "ResourceGroup"

Import-AzureRMAutomationRunbook -Name $runbookName -Path $scriptPath ` 
-ResourceGroupName $RGName -AutomationAccountName $automationAccountName ` 
-Type PowerShellWorkflow
```

Publishing a runbook

When you create or import a new runbook, you must publish it before you can run it. Each runbook in Automation has a Draft and a Published version. Only the Published version is available to be run, and only the Draft version can be edited. The Published version is unaffected by any changes to the Draft version. When the Draft version should be made available, then you publish it which overwrites the Published version with the Draft version.

To publish a runbook using the Azure Classic portal

1. Open the runbook in the Azure Classic portal.
2. At the top of the screen, click **Author**.
3. At the bottom of the screen, click **Publish** and then **Yes** to the verification message.

To publish a runbook using the Azure portal

1. Open the runbook in the Azure portal.
2. Click the **Edit** button.
3. Click the **Publish** button and then **Yes** to the verification message.

To publish a runbook using Windows PowerShell

You can use the [Publish-AzureRmAutomationRunbook](#) cmdlet to publish a runbook with Windows PowerShell. The following sample commands show how to publish a sample runbook.

```
$automationAccountName = "AutomationAccount"
$runbookName = "Sample_TestRunbook"
$RGName = "ResourceGroup"

Publish-AzureRmAutomationRunbook -AutomationAccountName $automationAccountName ` 
-Name $runbookName -ResourceGroupName $RGName
```

Next Steps

- To learn about how you can benefit from the Runbook and PowerShell Module Gallery, see [Runbook and module galleries for Azure Automation](#)
- To learn more about editing PowerShell and PowerShell Workflow runbooks with a textual editor, see [Editing textual runbooks in Azure Automation](#)
- To learn more about Graphical runbook authoring, see [Graphical authoring in Azure Automation](#)

Editing textual runbooks in Azure Automation

1/17/2017 • 5 min to read • [Edit on GitHub](#)

The textual editor in Azure Automation can be used to edit [PowerShell runbooks](#) and [PowerShell Workflow runbooks](#). This has the typical features of other code editors such as intellisense and color coding with additional special features to assist you in accessing resources common to runbooks. This article provides detailed steps for performing different functions with this editor.

The textual editor includes a feature to insert code for activities, assets, and child runbooks into a runbook. Rather than typing in the code yourself, you can select from a list of available resources and have the appropriate code inserted into the runbook.

Each runbook in Azure Automation has two versions, Draft and Published. You edit the Draft version of the runbook and then publish it so it can be executed. The Published version cannot be edited. See [Publishing a runbook](#) for more information.

To work with [Graphical Runbooks](#), see [Graphical authoring in Azure Automation](#).

To edit a runbook with the Azure portal

Use the following procedure to open a runbook for editing in the textual editor.

1. In the Azure portal, select your automation account.
2. Click the **Runbooks** tile to open the list of runbooks.
3. Click the name of the runbook you want to edit and then click the **Edit** button.
4. Perform the required editing.
5. Click **Save** when your edits are complete.
6. Click **Publish** if you want the latest draft version of the runbook to be published.

To insert a cmdlet into a runbook

1. In the Canvas of the textual editor, position the cursor where you want to place the cmdlet.
2. Expand the **Cmdlets** node in the Library control.
3. Expand the module containing the cmdlet you want to use.
4. Right click the cmdlet to insert and select **Add to canvas**. If the cmdlet has more than one parameter set, then the default set will be added. You can also expand the cmdlet to select a different parameter set.
5. The code for the cmdlet is inserted with its entire list of parameters.
6. Provide an appropriate value in place of the data type surrounded by braces <> for any required parameters. Remove any parameters you don't need.

To insert code for a child runbook into a runbook

1. In the Canvas of the textual editor, position the cursor where you want to place the code for the child runbook.
2. Expand the **Runbooks** node in the Library control.
3. Right click the runbook to insert and select **Add to canvas**.
4. The code for the child runbook is inserted with any placeholders for any runbook parameters.
5. Replace the placeholders with appropriate values for each parameter.

To insert an asset into a runbook

1. In the Canvas of the textual editor, position the cursor where you want to place the code for the child runbook.
2. Expand the **Assets** node in the Library control.

3. Expand the node for the type of asset you want.
4. Right click the asset to insert and select **Add to canvas**. For [variable assets](#), select either **Add "Get Variable" to canvas** or **Add "Set Variable" to canvas** depending on whether you want to get or set the variable.
5. The code for the asset is inserted into the runbook.

To edit a runbook with the Azure portal

Use the following procedure to open a runbook for editing in the textual editor.

1. In the Azure portal, select **Automation** and then click the name of an automation account.
2. Select the **Runbooks** tab.
3. Click the name of the runbook you want to edit and then select the **Author** tab.
4. Click the **Edit** button at the bottom of the screen.
5. Perform the required editing.
6. Click **Save** when your edits are complete.
7. Click **Publish** if you want the latest draft version of the runbook to be published.

To insert an activity into a Runbook

1. In the Canvas of the textual editor, position the cursor where you want to place the activity.
2. At the bottom of the screen, click **Insert** and then **Activity**.
3. In the **Integration Module** column, select the module that contains the activity.
4. In the **Activity** pane, select an activity.
5. In the **Description** column, note the description of the activity. Optionally, you can click View detailed help to launch help for the activity in the browser.
6. Click the right arrow. If the activity has parameters, they will be listed for your information.
7. Click the check button. Code to run the activity will be inserted into the runbook.
8. If the activity requires parameters, provide an appropriate value in place of the data type surrounded by braces <>.

To insert code for a child runbook into a runbook

1. In the Canvas of the textual editor, position the cursor where you want to place the [child runbook](#).
2. At the bottom of the screen, click **Insert** and then **Runbook**.
3. Select the runbook to insert from the center column and click the right arrow.
4. If the runbook has parameters, they will be listed for your information.
5. Click the check button. Code to run the selected runbook will be inserted into the current runbook.
6. If the runbook requires parameters, provide an appropriate value in place of the data type surrounded by braces <>.

To insert an asset into a runbook

1. In the Canvas of the textual editor, position the cursor where you want to place the activity to retrieve the asset.
2. At the bottom of the screen, click **Insert** and then **Setting**.
3. In the **Setting Action** column, select the action that you want.
4. Select from the available assets in the center column.
5. Click the check button. Code to get or set the asset will be inserted into the runbook.

To edit an Azure Automation runbook using Windows PowerShell

To edit a runbook with Windows PowerShell, you use the editor of your choice and save it to a .ps1 file. You can use the [Get-AzureAutomationRunbookDefinition](#) cmdlet to retrieve the contents of the runbook and then [Set-AzureAutomationRunbookDefinition](#) cmdlet to replace the existing draft runbook with the modified one.

To Retrieve the Contents of a Runbook Using Windows PowerShell

The following sample commands show how to retrieve the script for a runbook and save it to a script file. In this example, the Draft version is retrieved. It is also possible to retrieve the Published version of the runbook although this version cannot be changed.

```
$automationAccountName = "MyAutomationAccount"
$runbookName = "Sample-TestRunbook"
$scriptPath = "c:\runbooks\Sample-TestRunbook.ps1"

$runbookDefinition = Get-AzureAutomationRunbookDefinition -AutomationAccountName $automationAccountName -Name $runbookName -Slot Draft
$runbookContent = $runbookDefinition.Content

Out-File -InputObject $runbookContent -FilePath $scriptPath
```

To Change the Contents of a Runbook Using Windows PowerShell

The following sample commands show how to replace the existing contents of a runbook with the contents of a script file. Note that this is the same sample procedure as in [To import a runbook from a script file with Windows PowerShell](#).

```
$automationAccountName = "MyAutomationAccount"
$runbookName = "Sample-TestRunbook"
$scriptPath = "c:\runbooks\Sample-TestRunbook.ps1"

Set-AzureAutomationRunbookDefinition -AutomationAccountName $automationAccountName -Name $runbookName -Path $scriptPath -Overwrite
Publish-AzureAutomationRunbook -AutomationAccountName $automationAccountName -Name $runbookName
```

Related articles

- [Creating or importing a runbook in Azure Automation](#)
- [Learning PowerShell workflow](#)
- [Graphical authoring in Azure Automation](#)
- [Certificates](#)
- [Connections](#)
- [Credentials](#)
- [Schedules](#)
- [Variables](#)

Graphical authoring in Azure Automation

1/17/2017 • 22 min to read • [Edit on GitHub](#)

Introduction

Graphical Authoring allows you to create runbooks for Azure Automation without the complexities of the underlying Windows PowerShell or PowerShell Workflow code. You add activities to the canvas from a library of cmdlets and runbooks, link them together and configure to form a workflow. If you have ever worked with System Center Orchestrator or Service Management Automation (SMA), then this should look familiar to you.

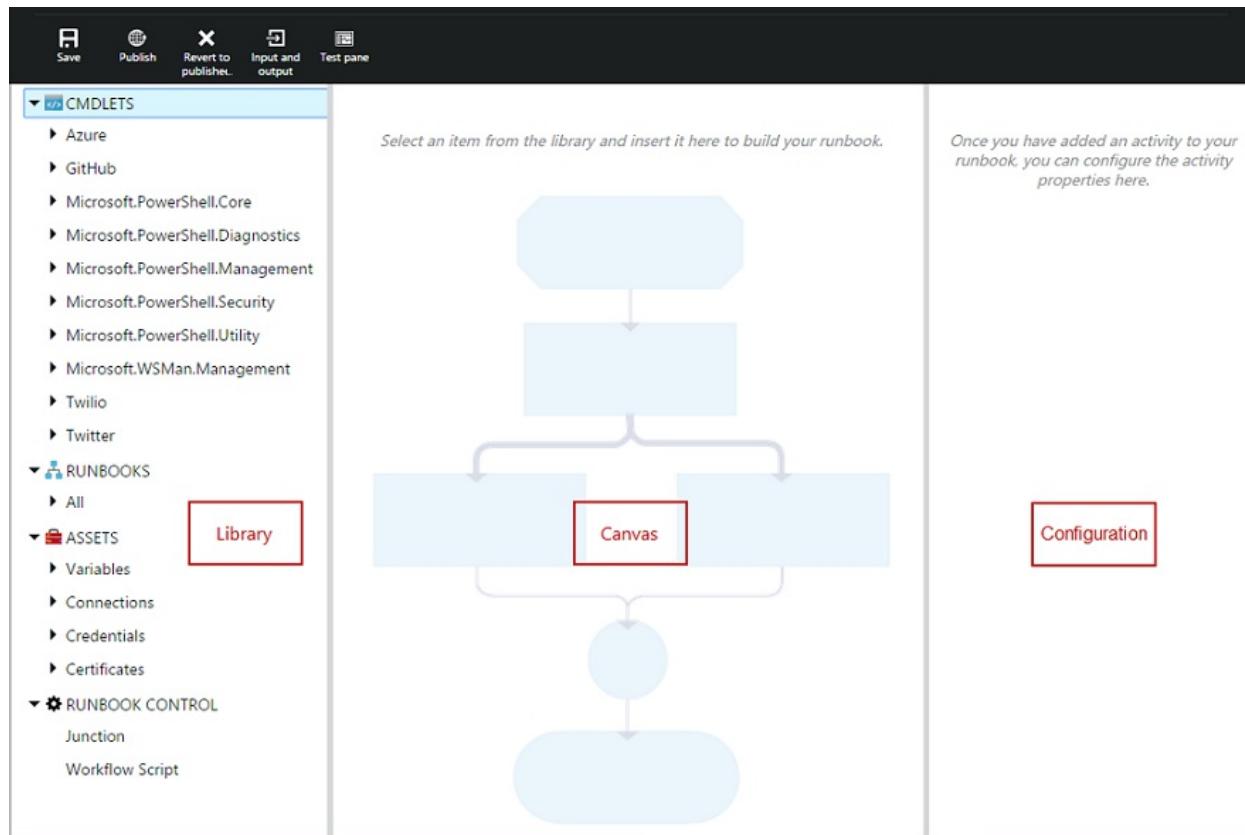
This article provides an introduction to graphical authoring and the concepts you need to get started in creating a graphical runbook.

Graphical runbooks

All runbooks in Azure Automation are Windows PowerShell Workflows. Graphical and Graphical PowerShell Workflow runbooks generate PowerShell code that is run by the Automation workers, but you are not able to view it or directly modify it. A Graphical runbook can be converted to a Graphical PowerShell Workflow runbook and vice-versa, but they cannot be converted to a textual runbook. An existing textual runbook cannot be imported into the graphical editor.

Overview of graphical editor

You can open the graphical editor in the Azure portal by creating or editing a graphical runbook.



The following sections describe the controls in the graphical editor.

Canvas

The Canvas is where you design your runbook. You add activities from the nodes in the Library control to the runbook and connect them with links to define the logic of the runbook.

You can use the controls at the bottom of the canvas to zoom in and out.



Library control

The Library control is where you select [activities](#) to add to your runbook. You add them to the canvas where you connect them to other activities. It includes four sections described in the following table.

SECTION	DESCRIPTION
Cmdlets	Includes all the cmdlets that can be used in your runbook. Cmdlets are organized by module. All of the modules that you have installed in your automation account will be available.
Runbooks	Includes the runbooks in your automation account. These runbooks can be added to the canvas to be used as child runbooks. Only runbooks of the same core type as the runbook being edited are shown; for Graphical runbooks only PowerShell-based runbooks are shown, while for Graphical PowerShell Workflow runbooks only PowerShell-Workflow-based runbooks are shown.
Assets	Includes the automation assets in your automation account that can be used in your runbook. When you add an asset to a runbook, it will add a workflow activity that gets the selected asset. In the case of variable assets, you can select whether to add an activity to get the variable or set the variable.
Runbook Control	Includes runbook control activities that can be used in your current runbook. A <i>Junction</i> takes multiple inputs and waits until all have completed before continuing the workflow. A <i>Code</i> activity runs one or more lines of PowerShell or PowerShell Workflow code depending on the graphical runbook type. You can use this activity for custom code or for functionality that is difficult to achieve with other activities.

Configuration control

The Configuration control is where you provide details for an object selected on the canvas. The properties available in this control will depend on the type of object selected. When you select an option in the Configuration control, it will open additional blades in order to provide additional information.

Test control

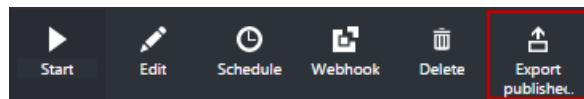
The Test control is not displayed when the graphical editor is first started. It is opened when you interactively [test a graphical runbook](#).

Graphical runbook procedures

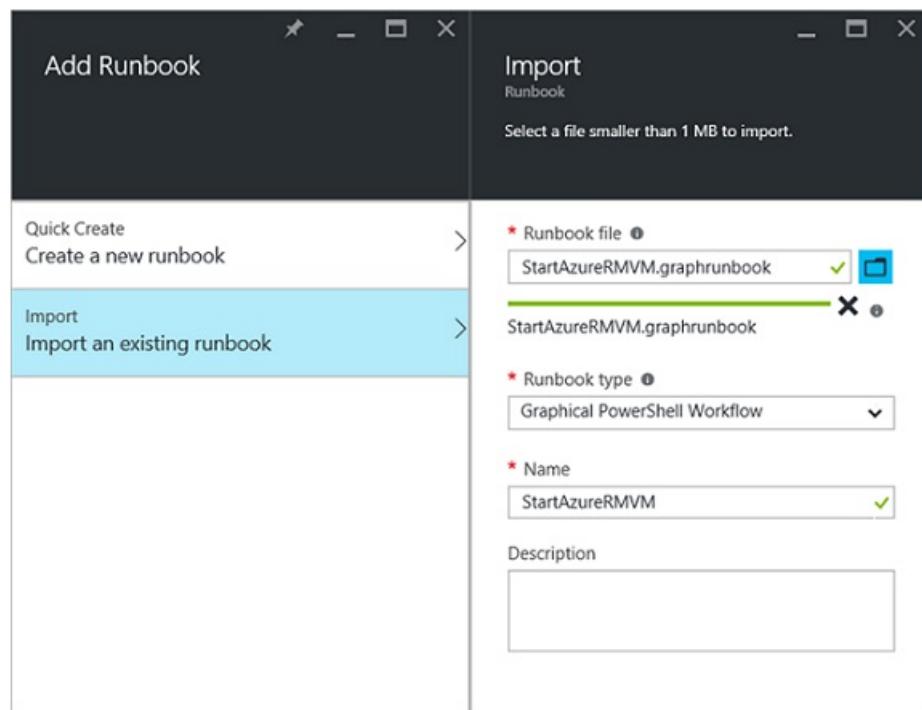
Exporting and importing a graphical runbook

You can only export the published version of a graphical runbook. If the runbook has not yet been published, then the **Export published** button will be disabled. When you click the **Export published** button, the runbook

is downloaded to your local computer. The name of the file matches the name of the runbook with a **graphrunbook** extension.



You can import a Graphical or Graphical PowerShell Workflow runbook file by selecting the **Import** option when adding a runbook. When you select the file to import, you can keep the same **Name** or provide a new one. The Runbook Type field will display the type of runbook after it assesses the file selected and if you attempt to select a different type that is not correct, a message will be presented noting there are potential conflicts and during conversion, there could be syntax errors.



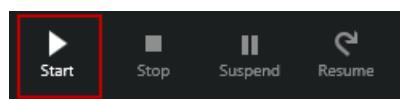
Testing a graphical runbook

You can test the Draft version of a runbook in the Azure portal while leaving the published version of the runbook unchanged, or you can test a new runbook before it has been published. This allows you to verify that the runbook is working correctly before replacing the published version. When you test a runbook, the Draft runbook is executed and any actions that it performs are completed. No job history is created, but output is displayed in the Test Output Pane.

Open the Test control for a runbook by opening the runbook for edit and then click on the **Test pane** button.



The Test control will prompt for any input parameters, and you can start the runbook by clicking on the **Start** button.



Publishing a graphical runbook

Each runbook in Azure Automation has a Draft and a Published version. Only the Published version is available to be run, and only the Draft version can be edited. The Published version is unaffected by any changes to the Draft version. When the Draft version is ready to be available, then you publish it which overwrites the Published version with the Draft version.

You can publish a graphical runbook by opening the runbook for editing and then clicking on the **Publish** button.



When a runbook has not yet been published, it has a status of **New**. When it is published, it has a status of **Published**. If you edit the runbook after it has been published, and the Draft and Published versions are different, the runbook has a status of **In edit**.

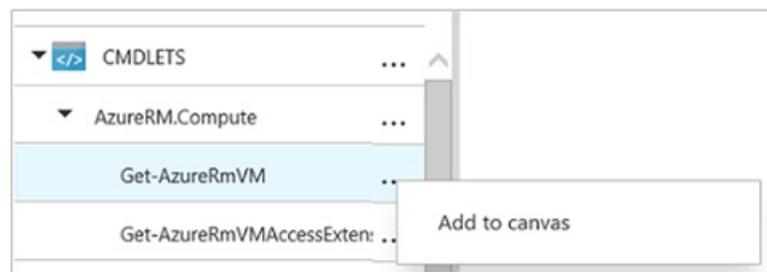
NAME	AUTHORING STATUS	LAST MODIFIED	TAGS
Test	✓ Published	3/4/2016 6:16 PM	
Create-SQLDB	✳ New	5/18/2016 3:16 PM	
Create-StdSQLVM	✳ New	5/18/2016 3:17 PM	
Get-AzureVMTutorial	✍ In edit	5/18/2016 3:25 PM	
Start-AIIVMs	✳ New	5/18/2016 3:15 PM	

You also have the option to revert to the Published version of a runbook. This throws away any changes made since the runbook was last published and replaces the Draft version of the runbook with the Published version.



Activities

Activities are the building blocks of a runbook. An activity can be a PowerShell cmdlet, a child runbook, or a workflow activity. You add an activity to the runbook by right clicking it in the Library control and selecting **Add to canvas**. You can then click and drag the activity to place it anywhere on the canvas that you like. The location of the activity on the canvas does not effect the operation of the runbook in any way. You can layout your runbook however you find it most suitable to visualize its operation.



Select the activity on the canvas to configure its properties and parameters in the Configuration blade. You can change the **Label** of the activity to something that is descriptive to you. The original cmdlet is still being run, you are simply changing its display name that will be used in the graphical editor. The label must be unique within the runbook.

Parameter sets

A parameter set defines the mandatory and optional parameters that will accept values for a particular cmdlet. All cmdlets have at least one parameter set, and some have multiple. If a cmdlet has multiple parameter sets, then you must select which one you will use before you can configure parameters. The parameters that you can configure will depend on the parameter set that you choose. You can change the parameter set used by an activity by selecting **Parameter Set** and selecting another set. In this case, any parameter values that you configured are lost.

In the following example, the Get-AzureRmVM cmdlet has three parameter sets. You cannot configure parameter values until you select one of the parameter sets. The ListVirtualMachineInResourceGroupParamSet parameter set is for returning all virtual machines in a resource group and has a single optional parameter. The GetVirtualMachineInResourceGroupParamSet is for specifying the virtual machine you want to return and has two mandatory and one optional parameter.

Parameter values

When you specify a value for a parameter, you select a data source to determine how the value will be specified. The data sources that are available for a particular parameter will depend on the valid values for that parameter. For example, Null will not be an available option for a parameter that does not allow null values.

DATA SOURCE	DESCRIPTION
Constant Value	Type in a value for the parameter. This is only available for the following data types: Int32,Int64,String,Boolean,DateTime,Switch.
Activity Output	Output from an activity that precedes the current activity in the workflow. All valid activities will be listed. Select just the activity to use its output for the parameter value. If the activity outputs an object with multiple properties, then you can type in the name of the property after selecting the activity.
Runbook Input	Select a runbook input parameter as input to the activity parameter.
Variable Asset	Select an Automation Variable as input.
Credential Asset	Select an Automation Credential as input.
Certificate Asset	Select an Automation Certificate as input.
Connection Asset	Select an Automation Connection as input.
PowerShell Expression	Specify simple PowerShell expression . The expression will be evaluated before the activity and the result used for the parameter value. You can use variables to refer to the output of an activity or a runbook input parameter.
Not configured	Clears any value that was previously configured.

Optional additional parameters

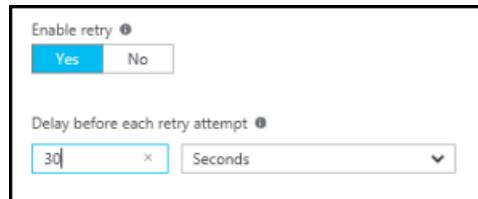
All cmdlets will have the option to provide additional parameters. These are PowerShell common parameters or

other custom parameters. You are presented with a text box where you can provide parameters using PowerShell syntax. For example, to use the **Verbose** common parameter, you would specify "**-Verbose:\$True**".

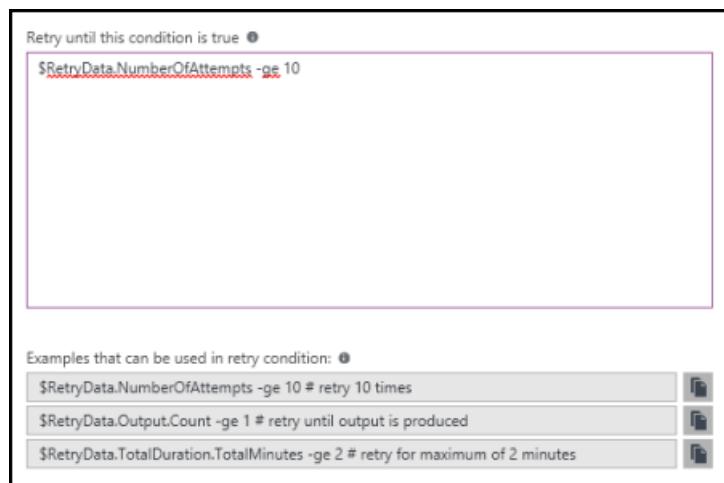
Retry activity

Retry Behavior allows an activity to be run multiple times until a particular condition is met, much like a loop. You can use this feature for activities that should run multiple times, are error prone and may need more than one attempt for success, or test the output information of the activity for valid data.

When you enable retry for an activity, you can set a delay and a condition. The delay is the time (measured in seconds or minutes) that the runbook will wait before it runs the activity again. If no delay is specified, then the activity will run again immediately after it completes.



The retry condition is a PowerShell expression that is evaluated after each time the activity runs. If the expression resolves to True, then the activity runs again. If the expression resolves to False then the activity does not run again, and the runbook moves on to the next activity.



The retry condition can use a variable called \$RetryData that provides access to information about the activity retries. This variable has the properties in the following table.

PROPERTY	DESCRIPTION
NumberOfAttempts	Number of times that the activity has been run.
Output	Output from the last run of the activity.
TotalDuration	Timed elapsed since the activity was started the first time.
StartedAt	Time in UTC format the activity was first started.

Following are examples of activity retry conditions.

```

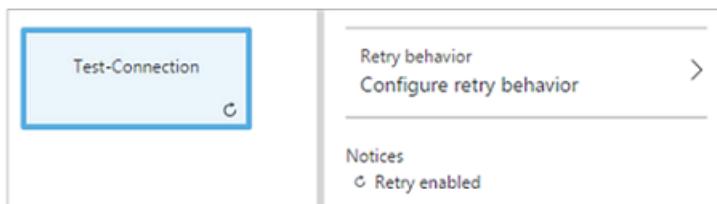
# Run the activity exactly 10 times.
$RetryData.NumberOfAttempts -ge 10

# Run the activity repeatedly until it produces any output.
$RetryData.Output.Count -ge 1

# Run the activity repeatedly until 2 minutes has elapsed.
$RetryData.TotalDuration.TotalMinutes -ge 2

```

After you configure a retry condition for an activity, the activity includes two visual cues to remind you. One is presented in the activity and the other is when you review the configuration of the activity.



Workflow Script control

A Code control is a special activity that accepts PowerShell or PowerShell Workflow script depending on the type of graphical runbook being authored in order to provide functionality that may otherwise not be available. It cannot accept parameters, but it can use variables for activity output and runbook input parameters. Any output of the activity is added to the databus unless it has no outgoing link in which case it is added to the output of the runbook.

For example the following code performs date calculations using a runbook input variable called \$NumberOfDays. It then sends a calculated date time as output to be used by subsequent activities in the runbook.

```

$DateTimeNow = (Get-Date).ToUniversalTime()
$DateTimeStart = ($DateTimeNow).AddDays(-$NumberOfDays)
$DateTimeStart

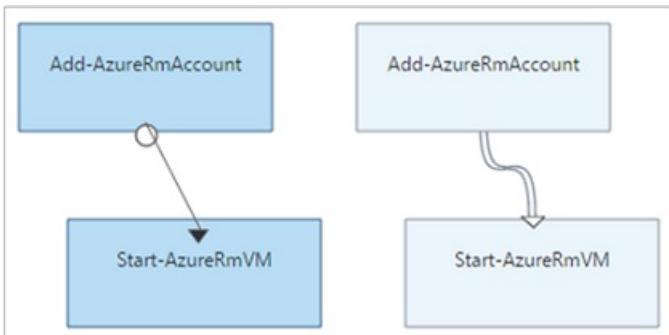
```

Links and workflow

A **link** in a graphical runbook connects two activities. It is displayed on the canvas as an arrow pointing from the source activity to the destination activity. The activities run in the direction of the arrow with the destination activity starting after the source activity completes.

Create a link

Create a link between two activities by selecting the source activity and clicking the circle at the bottom of the shape. Drag the arrow to the destination activity and release.



Select the link to configure its properties in the Configuration blade. This will include the link type which is described in the following table.

LINK TYPE	DESCRIPTION
Pipeline	The destination activity is run once for each object output from the source activity. The destination activity does not run if the source activity results in no output. Output from the source activity is available as an object.
Sequence	The destination activity runs only once. It receives an array of objects from the source activity. Output from the source activity is available as an array of objects.

Starting activity

A graphical runbook will start with any activities that do not have an incoming link. This will often be only one activity which would act as the starting activity for the runbook. If multiple activities do not have an incoming link, then the runbook will start by running them in parallel. It will then follow the links to run other activities as each completes.

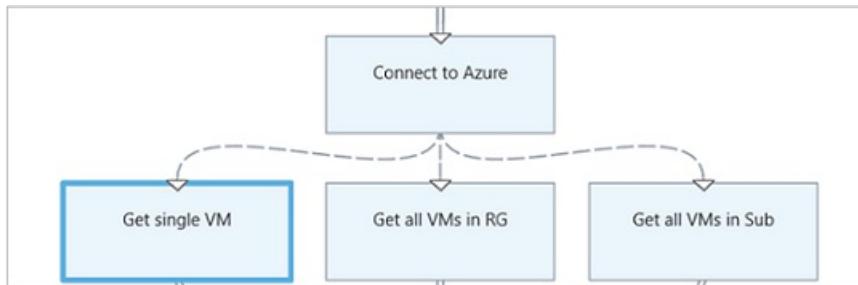
Conditions

When you specify a condition on a link, the destination activity is only run if the condition resolves to true. You will typically use an \$ActivityOutput variable in a condition to retrieve the output from the source activity.

For a pipeline link, you specify a condition for a single object, and the condition is evaluated for each object output by the source activity. The destination activity is then run for each object that satisfies the condition. For example, with a source activity of Get-AzureRmVm, the following syntax could be used for a conditional pipeline link to retrieve only virtual machines in the resource group named *Group1*.

```
$ActivityOutput['Get Azure VMs'].Name -match "Group1"
```

For a sequence link, the condition is only evaluated once since a single array is returned containing all objects output from the source activity. Because of this, a sequence link cannot be used for filtering like a pipeline link but will simply determine whether or not the next activity is run. Take for example the following set of activities in our Start VM runbook.



There are three different sequence links that are verifying values were provided to two runbook input parameters representing VM name and Resource Group name in order to determine which is the appropriate action to take - start a single VM, start all VMs in the resource group, or all VMs in a subscription. For the sequence link between Connect to Azure and Get single VM, here is the condition logic:

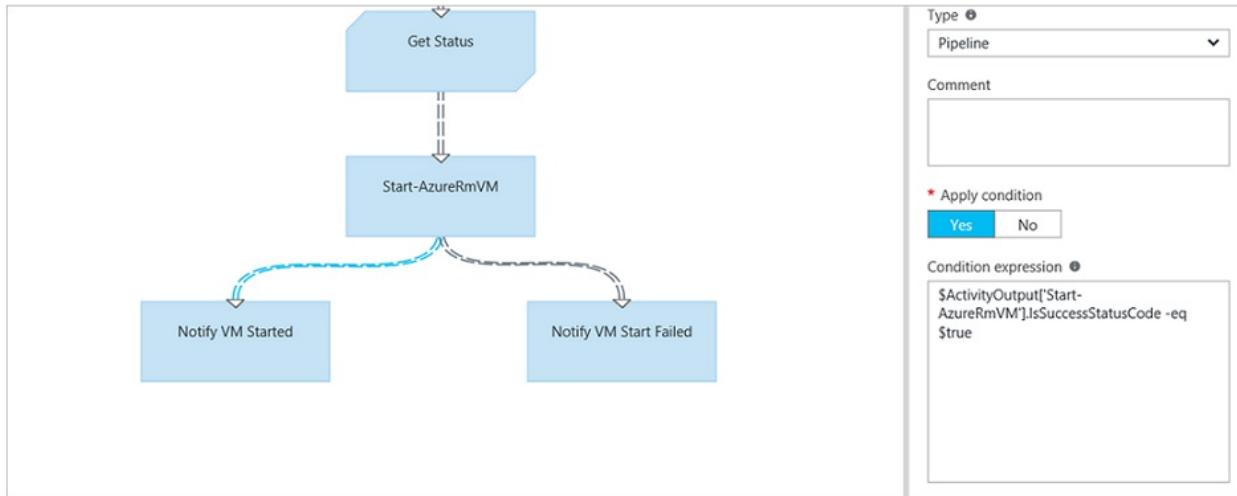
```

<#
Both VMName and ResourceGroupName runbook input parameters have values
#>
(
    (($VMName -ne $null) -and ($VMName.Length -gt 0))
) -and (
    (($ResourceGroupName -ne $null) -and ($ResourceGroupName.Length -gt 0))
)
  
```

When you use a conditional link, the data available from the source activity to other activities in that branch will

be filtered by the condition. If an activity is the source to multiple links, then the data available to activities in each branch will depend on the condition in the link connecting to that branch.

For example, the **Start-AzureRmVm** activity in the runbook below starts all virtual machines. It has two conditional links. The first conditional link uses the expression `$ActivityOutput['Start-AzureRmVM'].IsSuccessStatusCode -eq $true` to filter if the Start-AzureRmVm activity completed successfully. The second uses the expression `$ActivityOutput['Start-AzureRmVM'].IsSuccessStatusCode -ne $true` to filter if the Start-AzureRmVm activity failed to start the virtual machine.



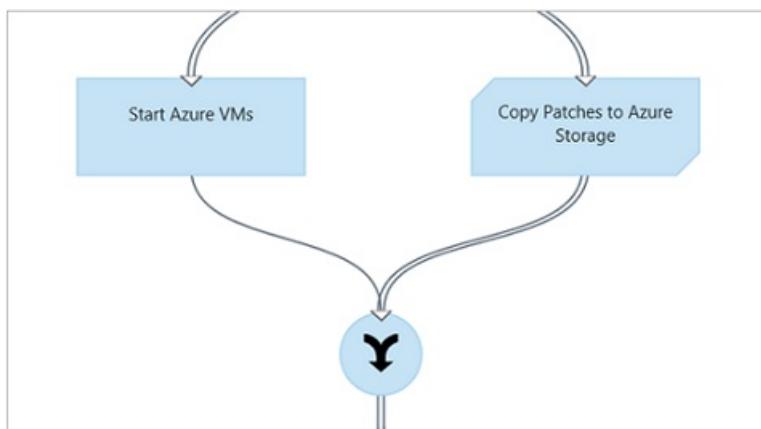
Any activity that follows the first link and uses the activity output from Get-AzureVM will only get the virtual machines that were started at the time that Get-AzureVM was run. Any activity that follows the second link will only get the the virtual machines that were stopped at the time that Get-AzureVM was run. Any activity following the third link will get all virtual machines regardless of their running state.

Junctions

A junction is a special activity that will wait until all incoming branches have completed. This allows you to run multiple activities in parallel and ensure that all have completed before moving on.

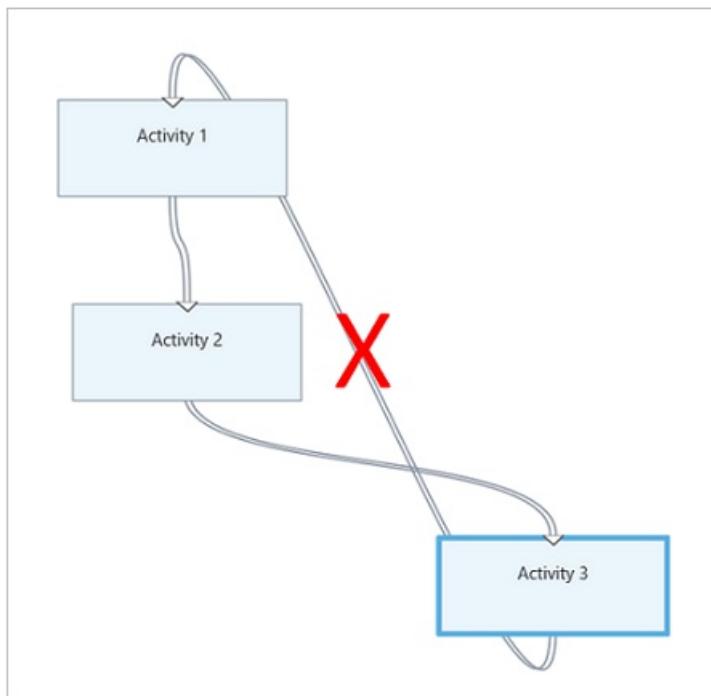
While a junction can have an unlimited number of incoming links, not more than one of those links can be a pipeline. The number of incoming sequence links is not constrained. You will be allowed to create the junction with multiple incoming pipeline links and save the runbook, but it will fail when it is run.

The example below is part of a runbook that starts a set of virtual machines while simultaneously downloading patches to be applied to those machines. A junction is used to ensure that both processes are completed before the runbook continues.



Cycles

A cycle is when a destination activity links back to its source activity or to another activity that eventually links back to its source. Cycles are currently not allowed in graphical authoring. If your runbook has a cycle, it will save properly but will receive an error when it runs.



Sharing data between activities

Any data that is output by an activity with an outgoing link is written to the *databus* for the runbook. Any activity in the runbook can use data on the databus to populate parameter values or include in script code. An activity can access the output of any previous activity in the workflow.

How the data is written to the databus depends on the type of link on the activity. For a **pipeline**, the data is output as multiple objects. For a **sequence** link, the data is output as an array. If there is only one value, it will be output as a single element array.

You can access data on the databus using one of two methods. First is using an **Activity Output** data source to populate a parameter of another activity. If the output is an object, you can specify a single property.

Data source

Activity output

Select data

- ▼ Get-Date
 - ▼ Output (DateTime)
 - ▶ Date (DateTime)
 - Day (Int32)
 - DayOfWeek (DayOfWeek)**
 - DayOfYear (Int32)
 - Hour (Int32)
 - Kind (DateTimeKind)
 - Millisecond (Int32)
 - Minute (Int32)
 - Month (Int32)

Selected activity name

Get-Date

Field path ⓘ

DayOfWeek

You can also retrieve the output of an activity in a **PowerShell Expression** data source or from a **Workflow Script** activity with an **ActivityOutput** variable. If the output is an object, you can specify a single property. **ActivityOutput** variables use the following syntax.

```
$ActivityOutput['Activity Label']
$ActivityOutput['Activity Label'].PropertyName
```

Checkpoints

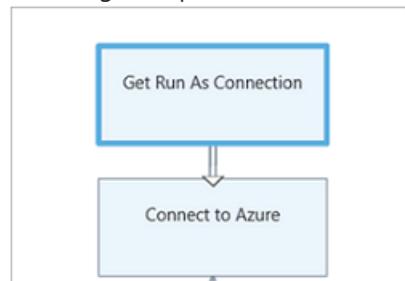
You can set [checkpoints](#) in a Graphical PowerShell Workflow runbook by selecting *Checkpoint runbook* on any activity. This causes a checkpoint to be set after the activity runs.



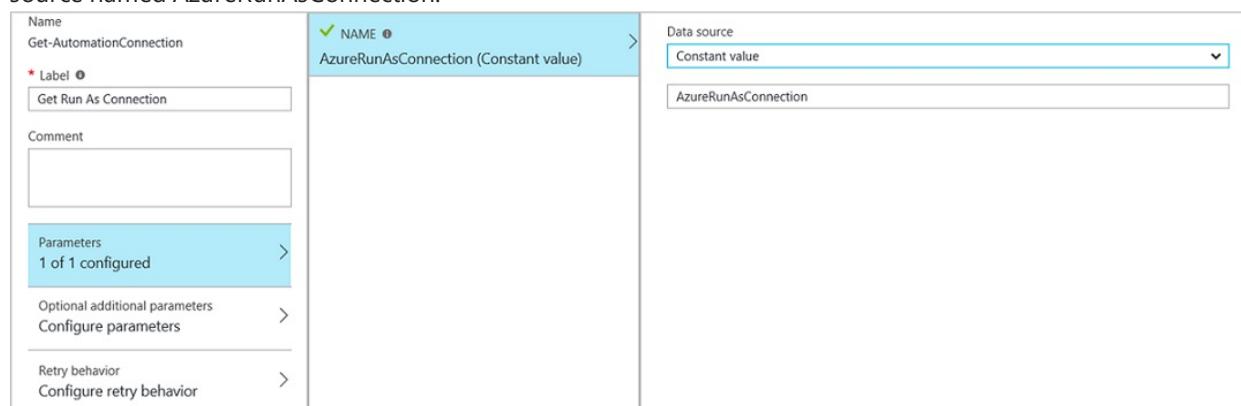
Checkpoints are only enabled in Graphical PowerShell Workflow runbooks, it is not available in Graphical runbooks. If the runbook uses Azure cmdlets, you should follow any checkpointed activity with an Add-AzureRMAccount in case the runbook is suspended and restarts from this checkpoint on a different worker.

Authenticating to Azure resources

Runbooks in Azure Automation that manage Azure resources will require authentication to Azure. The new [Run As account](#) feature (also referred to as a service principal) is the default method to access Azure Resource Manager resources in your subscription with Automation runbooks. You can add this functionality to a graphical runbook by adding the **AzureRunAsConnection** Connection asset, which is using the PowerShell [Get-AutomationConnection](#) cmdlet, and [Add-AzureRmAccount](#) cmdlet to the canvas. This is illustrated in the following example.



The Get Run As Connection activity (i.e. Get-AutomationConnection), is configured with a constant value data source named AzureRunAsConnection.



Name Get-AutomationConnection	✓ NAME <small>•</small> AzureRunAsConnection (Constant value)	Data source Constant value
* Label <small>•</small> Get Run As Connection		AzureRunAsConnection
Comment		
Parameters 1 of 1 configured	>	
Optional additional parameters Configure parameters	>	
Retry behavior Configure retry behavior	>	

The next activity, Add-AzureRmAccount, adds the authenticated Run As account for use in the runbook.

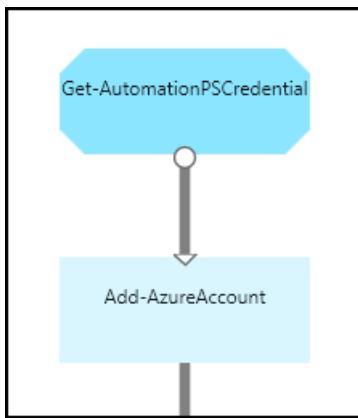
The screenshot shows the configuration of the 'Add-AzureRmAccount' activity. On the left, a graphical runbook snippet shows a 'Get Run As Connection' activity followed by a 'Connect to Azure' activity. On the right, the configuration pane for 'Add-AzureRmAccount' is displayed:

- Name:** Add-AzureRmAccount
- Label:** Connect to Azure
- Comment:** (empty)
- Parameters:**
 - APPLICATIONID**: true (Constant value)
 - CERTIFICATEHUMPRINT**: Get Run As Connection (Activity output)
 - ENVIRONMENT**: Not configured
 - ENVIRONMENTNAME**: Not configured
 - SERVICEPRINCIPAL**: true (Constant value)
 - TENANTID**: Get Run As Connection (Activity output)
- Optional additional parameters:** Configure parameters
- Retry behavior:** Configure retry behavior

For the parameters **APPLICATIONID**, **CERTIFICATEHUMPRINT**, and **TENANTID** you will need to specify the name of the property for the Field path because the activity outputs an object with multiple properties. Otherwise when you execute the runbook, it will fail attempting to authenticate. This is what you need at a minimum to authenticate your runbook with the Run As account.

To maintain backwards compatibility for subscribers who have created an Automation account using an [Azure AD User account](#) to manage Azure Service Management (ASM) or Azure Resource Manager resources, the method to authenticate is the Add-AzureAccount cmdlet with a [credential asset](#) that represents an Active Directory user with access to the Azure account.

You can add this functionality to a graphical runbook by adding a credential asset to the canvas followed by an Add-AzureAccount activity. Add-AzureAccount uses the credential activity for its input. This is illustrated in the following example.



You have to authenticate at the start of the runbook and after each checkpoint. This means adding an addition Add-AzureAccount activity after any Checkpoint-Workflow activity. You do not need an addition credential activity since you can use the same

The screenshot shows the configuration of the 'Get-AutomationPSCredential' activity. On the left, a graphical runbook snippet shows a 'Get-AutomationPSCredential' activity followed by an 'Add-AzureAccount' activity. On the right, the configuration pane for 'Get-AutomationPSCredential' is displayed:

- CREDENTIAL**: Get-AutomationPSCredential (Activity output)
- ENVIRONMENT**: Not configured
- PROFILE**: Not configured
- Activity output** (right pane):
 - UNSELECT**
 - ACTIVITY OUTPUT**: Get-AutomationPSCredential
 - POWERSHELL EXPRESSION**
- Field path**: (empty)

Runbook input and output

Runbook input

A runbook may require input either from a user when they start the runbook through the Azure portal or from another runbook if the current one is used as a child. For example, if you have a runbook that creates a virtual

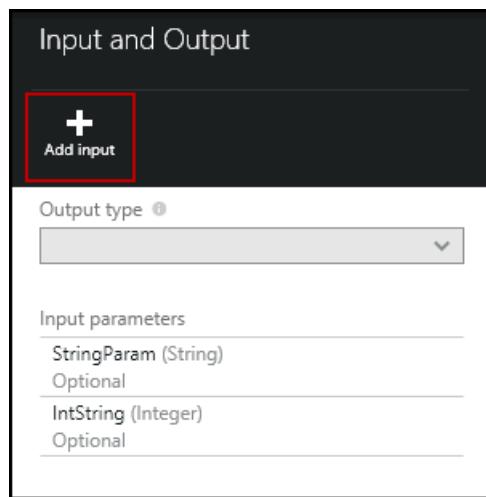
machine, you may need to provide information such as the name of the virtual machine and other properties each time you start the runbook.

You accept input for a runbook by defining one or more input parameters. You provide values for these parameters each time the runbook is started. When you start a runbook with the Azure portal, it will prompt you to provide values for each of the runbook's input parameters.

You can access input parameters for a runbook by clicking the **Input and output** button on the runbook toolbar.



This opens the **Input and Output** control where you can edit an existing input parameter or create a new one by clicking **Add input**.



Each input parameter is defined by the properties in the following table.

PROPERTY	DESCRIPTION
Name	The unique name of the parameter. This can only contain alpha numeric characters and cannot contain a space.
Description	An optional description for the input parameter.
Type	Data type expected for the parameter value. The Azure portal will provide an appropriate control for the data type for each parameter when prompting for input.
Mandatory	Specifies whether a value must be provided for the parameter. The runbook cannot be started if you do not provide a value for each mandatory parameter that does not have a default value defined.
Default Value	Specifies what value is used for the parameter if one is not provided. This can either be Null or a specific value.

Runbook output

Data created by any activity that does not have an outgoing link will be added to the [output of the runbook](#). The output is saved with the runbook job and is available to a parent runbook when the runbook is used as a child.

PowerShell expressions

One of the advantages of graphical authoring is providing you with the ability to build a runbook with minimal knowledge of PowerShell. Currently, you do need to know a bit of PowerShell though for populating certain [parameter values](#) and for setting [link conditions](#). This section provides a quick introduction to PowerShell expressions for those users who may not be familiar with it. Full details of PowerShell are available at [Scripting with Windows PowerShell](#).

PowerShell expression data source

You can use a PowerShell expression as a data source to populate the value of an [activity parameter](#) with the results of some PowerShell code. This could be a single line of code that performs some simple function or multiple lines that perform some complex logic. Any output from a command that is not assigned to a variable is output to the parameter value.

For example, the following command would output the current date.

```
Get-Date
```

The following commands build a string from the current date and assign it to a variable. The contents of the variable are then sent to the output

```
$string = "The current date is " + (Get-Date)  
$string
```

The following commands evaluate the current date and return a string indicating whether the current day is a weekend or weekday.

```
$date = Get-Date  
if (($date.DayOfWeek = "Saturday") -or ($date.DayOfWeek = "Sunday")) { "Weekend" }  
else { "Weekday" }
```

Activity output

To use the output from a previous activity in the runbook, use the \$ActivityOutput variable with the following syntax.

```
$ActivityOutput['Activity Label'].PropertyName
```

For example, you may have an activity with a property that requires the name of a virtual machine in which case you could use the following expression.

```
$ActivityOutput['Get-AzureVm'].Name
```

If the property that required the virtual machine object instead of just a property, then you would return the entire object using the following syntax.

```
$ActivityOutput['Get-AzureVm']
```

You can also use the output of an activity in a more complex expression such as the following that concatenates text to the virtual machine name.

```
"The computer name is " + $ActivityOutput['Get-AzureVm'].Name
```

Conditions

Use [comparison operators](#) to compare values or determine if a value matches a specified pattern. A comparison returns a value of either \$true or \$false.

For example, the following condition determines whether the virtual machine from an activity named *Get-AzureVM* is currently *stopped*.

```
$ActivityOutput["Get-AzureVM"].PowerState -eq "Stopped"
```

The following condition checks whether the same virtual machine is in any state other than *stopped*.

```
$ActivityOutput["Get-AzureVM"].PowerState -ne "Stopped"
```

You can join multiple conditions using a [logical operator](#) such as **-and** or **-or**. For example, the following condition checks whether the same virtual machine in the previous example is in a state of *stopped* or *stopping*.

```
($ActivityOutput["Get-AzureVM"].PowerState -eq "Stopped") -or ($ActivityOutput["Get-AzureVM"].PowerState -eq "Stopping")
```

Hashtables

[Hashtables](#) are name/value pairs that are useful for returning a set of values. Properties for certain activities may expect a hashtable instead of a simple value. You may also see a hashtable referred to as a dictionary.

You create a hashtable with the following syntax. A hashtable can contain any number of entries but each is defined by a name and value.

```
@{ <name> = <value>; [<name> = <value> ] ... }
```

For example, the following expression creates a hashtable to be used in the data source for an activity parameter that expected a hashtable with values for an internet search.

```
$query = "Azure Automation"
$count = 10
$h = @{'q'=$query; 'lr'='lang_ja'; 'count'=$Count}
$h
```

The following example uses output from an activity called *Get Twitter Connection* to populate a hashtable.

```
@{'ApiKey'=$ActivityOutput['Get Twitter Connection'].ConsumerAPIKey;
'ApiSecret'=$ActivityOutput['Get Twitter Connection'].ConsumerAPISecret;
'AccessToken'=$ActivityOutput['Get Twitter Connection'].AccessToken;
'AccessTokenSecret'=$ActivityOutput['Get Twitter Connection'].AccessTokenSecret}
```

Next Steps

- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)
- To get started with Graphical runbooks, see [My first graphical runbook](#)
- To know more about runbook types, their advantages and limitations, see [Azure Automation runbook types](#)

- To understand how to authenticate using the Automation Run As account, see [Configure Azure Run As Account](#)

Testing a runbook in Azure Automation

1/17/2017 • 1 min to read • [Edit on GitHub](#)

When you test a runbook, the [Draft version](#) is executed and any actions that it performs are completed. No job history is created, but the [Output](#) and [Warning and Error](#) streams are displayed in the Test output Pane. Messages to the [Verbose Stream](#) are displayed in the Output Pane only if the [\\$VerbosePreference variable](#) is set to Continue.

Even though the draft version is being run, the runbook still executes the workflow normally and performs any actions against resources in the environment. For this reason, you should only test runbooks at non-production resources.

The procedure to test each [type of runbook](#) is the same, and there is no difference in testing between the textual editor and the graphical editor in the Azure portal.

To test a runbook in the Azure portal

You can work with any [runbook type](#) in the Azure portal.

1. Open the Draft version of the runbook in either the [textual editor](#) or [graphical editor](#).
2. Click the **Test** button to open the Test blade.
3. If the runbook has parameters, they will be listed in the left pane where you can provide values to be used for the test.
4. If you want to run the test on a [Hybrid Runbook Worker](#), then change **Run Settings** to **Hybrid Worker** and select the name of the target group. Otherwise, keep the default **Azure** to run the test in the cloud.
5. Click the **Start** button to start the test.
6. If the runbook is [PowerShell Workflow](#) or [Graphical](#), then you can stop or suspend it while it is being tested with the buttons underneath the Output Pane. When you suspend the runbook, it completes the current activity before being suspended. Once the runbook is suspended, you can stop it or restart it.
7. Inspect the output from the runbook in the output pane.

Next Steps

- To learn how to create or import a runbook, see [Creating or importing a runbook in Azure Automation](#)
- To learn more about Graphical Authoring, see [Graphical authoring in Azure Automation](#)
- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)
- To learn more about configuring runbooks to return status messages and errors, including recommended practices, see [Runbook output and messages in Azure Automation](#)

Learning key Windows PowerShell Workflow concepts for Automation runbooks

1/23/2017 • 9 min to read • [Edit on GitHub](#)

Runbooks in Azure Automation are implemented as Windows PowerShell Workflows. A Windows PowerShell Workflow is similar to a Windows PowerShell script but has some significant differences that can be confusing to a new user. While this article is intended to help you write runbooks using PowerShell workflow, we recommend you write runbooks using PowerShell unless you need checkpoints. There are a number of syntax differences when authoring PowerShell Workflow runbooks and these differences require a bit more work to write effective workflows.

A workflow is a sequence of programmed, connected steps that perform long-running tasks or require the coordination of multiple steps across multiple devices or managed nodes. The benefits of a workflow over a normal script include the ability to simultaneously perform an action against multiple devices and the ability to automatically recover from failures. A Windows PowerShell Workflow is a Windows PowerShell script that leverages Windows Workflow Foundation. While the workflow is written with Windows PowerShell syntax and launched by Windows PowerShell, it is processed by Windows Workflow Foundation.

For complete details on the topics in this article, see [Getting Started with Windows PowerShell Workflow](#).

Basic structure of a workflow

The first step to converting a PowerShell script to a PowerShell workflow is enclosing it with the **Workflow** keyword. A workflow starts with the **Workflow** keyword followed by the body of the script enclosed in braces. The name of the workflow follows the **Workflow** keyword as shown in the following syntax.

```
Workflow Test-Workflow
{
    <Commands>
}
```

The name of the workflow must match the name of the Automation runbook. If the runbook is being imported, then the filename must match the workflow name and must end in .ps1.

To add parameters to the workflow, use the **Param** keyword just as you would to a script.

Code changes

PowerShell workflow code looks almost identical to PowerShell script code except for a few significant changes. The following sections describe changes that you will need to make to a PowerShell script for it to run in a workflow.

Activities

An activity is a specific task in a workflow. Just as a script is composed of one or more commands, a workflow is composed of one or more activities that are carried out in a sequence. Windows PowerShell Workflow automatically converts many of the Windows PowerShell cmdlets to activities when it runs a workflow. When you specify one of these cmdlets in your runbook, the corresponding activity is actually run by Windows Workflow Foundation. For those cmdlets without a corresponding activity, Windows PowerShell Workflow automatically runs the cmdlet within an [InlineScript](#) activity. There is a set of cmdlets that are excluded and cannot be used in a workflow unless you explicitly include them in an InlineScript block. For further details on these concepts, see

Using Activities in Script Workflows.

Workflow activities share a set of common parameters to configure their operation. For details about the workflow common parameters, see [about_WorkflowCommonParameters](#).

Positional parameters

You can't use positional parameters with activities and cmdlets in a workflow. All this means is that you must use parameter names.

For example, consider the following code that gets all running services.

```
Get-Service | Where-Object {$_.Status -eq "Running"}
```

If you try to run this same code in a workflow, you'll get a message like "Parameter set cannot be resolved using the specified named parameters." To correct this, simply provide the parameter name as in the following.

```
Workflow Get-RunningServices
{
    Get-Service | Where-Object -FilterScript {$_.Status -eq "Running"}
}
```

Deserialized objects

Objects in workflows are deserialized. This means that their properties are still available, but not their methods. For example, consider the following PowerShell code that stops a service using the Stop method of the Service object.

```
$Service = Get-Service -Name MyService
$Service.Stop()
```

If you try to run this in a workflow, you'll get an error saying "Method invocation is not supported in a Windows PowerShell Workflow".

One option is to wrap these two lines of code in an [InlineScript](#) block in which case \$Service would be a service object within the block.

```
Workflow Stop-Service
{
    InlineScript {
        $Service = Get-Service -Name MyService
        $Service.Stop()
    }
}
```

Another option is to use another cmdlet that performs the same functionality as the method, if one is available. In the case of our sample, the Stop-Service cmdlet provides the same functionality as the Stop method, and you could use the following for a workflow.

```
Workflow Stop-MyService
{
    $Service = Get-Service -Name MyService
    Stop-Service -Name $Service.Name
}
```

InlineScript

The **InlineScript** activity is useful when you need to run one or more commands as traditional PowerShell script instead of PowerShell workflow. While commands in a workflow are sent to Windows Workflow Foundation for processing, commands in an InlineScript block are processed by Windows PowerShell.

InlineScript uses the syntax shown below.

```
InlineScript
{
    <Script Block>
} <Common Parameters>
```

You can return output from an InlineScript by assigning the output to a variable. The following example stops a service and then outputs the service name.

```
Workflow Stop-MyService
{
    $Output = InlineScript {
        $Service = Get-Service -Name MyService
        $Service.Stop()
        $Service
    }

    $Output.Name
}
```

You can pass values into an InlineScript block, but you must use **\$Using** scope modifier. The following example is identical to the previous example except that the service name is provided by a variable.

```
Workflow Stop-MyService
{
    $serviceName = "MyService"

    $Output = InlineScript {
        $Service = Get-Service -Name $Using:serviceName
        $Service.Stop()
        $Service
    }

    $Output.Name
}
```

While InlineScript activities may be critical in certain workflows, they do not support workflow constructs and should only be used when necessary for the following reasons:

- You cannot use [checkpoints](#) inside of an InlineScript block. If a failure occurs within the block, it must be resumed from the beginning of the block.
- You cannot use [parallel execution](#) inside of an InlineScriptBlock.
- InlineScript affects scalability of the workflow since it holds the Windows PowerShell session for the entire length of the InlineScript block.

For further details on using InlineScript, see [Running Windows PowerShell Commands in a Workflow](#) and [about_InlineScript](#).

Parallel processing

One advantage of Windows PowerShell Workflows is the ability to perform a set of commands in parallel instead of sequentially as with a typical script.

You can use the **Parallel** keyword to create a script block with multiple commands that will run concurrently. This uses the syntax shown below. In this case, Activity1 and Activity2 will start at the same time. Activity3 will start only after both Activity1 and Activity2 have completed.

```
Parallel
{
    <Activity1>
    <Activity2>
}
<Activity3>
```

For example, consider the following PowerShell commands that copy multiple files to a network destination. These commands are run sequentially so that one file must finish copying before the next is started.

```
$Copy-Item -Path C:\LocalPath\file1.txt -Destination \\NetworkPath\file1.txt
$Copy-Item -Path C:\LocalPath\file2.txt -Destination \\NetworkPath\file2.txt
$Copy-Item -Path C:\LocalPath\file3.txt -Destination \\NetworkPath\file3.txt
```

The following workflow runs these same commands in parallel so that they all start copying at the same time. Only after they are all completely copied is the completion message displayed.

```
Workflow Copy-Files
{
    Parallel
    {
        $Copy-Item -Path "C:\LocalPath\file1.txt" -Destination "\\NetworkPath"
        $Copy-Item -Path "C:\LocalPath\file2.txt" -Destination "\\NetworkPath"
        $Copy-Item -Path "C:\LocalPath\file3.txt" -Destination "\\NetworkPath"
    }

    Write-Output "Files copied."
}
```

You can use the **ForEach -Parallel** construct to process commands for each item in a collection concurrently. The items in the collection are processed in parallel while the commands in the script block run sequentially. This uses the syntax shown below. In this case, Activity1 will start at the same time for all items in the collection. For each item, Activity2 will start after Activity1 is complete. Activity3 will start only after both Activity1 and Activity2 have completed for all items.

```
ForEach -Parallel ($<item> in $<collection>)
{
    <Activity1>
    <Activity2>
}
<Activity3>
```

The following example is similar to the previous example copying files in parallel. In this case, a message is displayed for each file after it copies. Only after they are all completely copied is the final completion message displayed.

```

Workflow Copy-Files
{
    $files = @("C:\LocalPath\File1.txt","C:\LocalPath\File2.txt","C:\LocalPath\File3.txt")

    ForEach -Parallel ($File in $Files)
    {
        $Copy-Item -Path $File -Destination \\NetworkPath
        Write-Output "$File copied."
    }

    Write-Output "All files copied."
}

```

NOTE

We do not recommend running child runbooks in parallel since this has been shown to give unreliable results. The output from the child runbook sometimes will not show up, and settings in one child runbook can affect the other parallel child runbooks

Checkpoints

A *checkpoint* is a snapshot of the current state of the workflow that includes the current value for variables and any output generated to that point. If a workflow ends in error or is suspended, then the next time it is run it will start from its last checkpoint instead of the start of the workflow. You can set a checkpoint in a workflow with the **Checkpoint-Workflow** activity.

In the following sample code, an exception occurs after Activity2 causing the workflow to end. When the workflow is run again, it starts by running Activity2 since this was just after the last checkpoint set.

```

<Activity1>
Checkpoint-Workflow
<Activity2>
<Exception>
<Activity3>

```

You should set checkpoints in a workflow after activities that may be prone to exception and should not be repeated if the workflow is resumed. For example, your workflow may create a virtual machine. You could set a checkpoint both before and after the commands to create the virtual machine. If the creation fails, then the commands would be repeated if the workflow is started again. If the workflow fails after the creation succeeds, then the virtual machine will not be created again when the workflow is resumed.

The following example copies multiple files to a network location and sets a checkpoint after each file. If the network location is lost, then the workflow will end in error. When it is started again, it will resume at the last checkpoint meaning that only the files that have already been copied will be skipped.

```

Workflow Copy-Files
{
    $files = @("C:\LocalPath\file1.txt","C:\LocalPath\file2.txt","C:\LocalPath\file3.txt")

    ForEach ($File in $Files)
    {
        $Copy-Item -Path $File -Destination \\NetworkPath
        Write-Output "$File copied."
        Checkpoint-Workflow
    }

    Write-Output "All files copied."
}

```

Because username credentials are not persisted after you call the [Suspend-Workflow](#) activity or after the last checkpoint, you need to set the credentials to null and then retrieve them again from the asset store after **Suspend-Workflow** or checkpoint is called. Otherwise, you may receive the following error message: *The workflow job cannot be resumed, either because persistence data could not be saved completely, or saved persistence data has been corrupted. You must restart the workflow.*

The following same code demonstrates how to handle this in your PowerShell Workflow runbooks.

```

workflow CreateTestVms
{
    $Cred = Get-AzureAutomationCredential -Name "MyCredential"
    $null = Add-AzureRmAccount -Credential $Cred

    $VmsToCreate = Get-AzureAutomationVariable -Name "VmsToCreate"

    foreach ($VmName in $VmsToCreate)
    {
        # Do work first to create the VM (code not shown)

        # Now add the VM
        New-AzureRmVm -VM $Vm -Location "WestUs" -ResourceGroupName "ResourceGroup01"

        # Checkpoint so that VM creation is not repeated if workflow suspends
        $Cred = $null
        Checkpoint-Workflow
        $Cred = Get-AzureAutomationCredential -Name "MyCredential"
        $null = Add-AzureRmAccount -Credential $Cred
    }
}

```

This is not required if you are authenticating using a Run As account configured with a service principal.

For more information about checkpoints, see [Adding Checkpoints to a Script Workflow](#).

Next steps

- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)

Child runbooks in Azure Automation

2/2/2017 • 5 min to read • [Edit on GitHub](#)

It is a best practice in Azure Automation to write reusable, modular runbooks with a discrete function that can be used by other runbooks. A parent runbook will often call one or more child runbooks to perform required functionality. There are two ways to call a child runbook, and each has distinct differences that you should understand so that you can determine which will be best for your different scenarios.

Invoking a child runbook using inline execution

To invoke a runbook inline from another runbook, you use the name of the runbook and provide values for its parameters exactly like you would use an activity or cmdlet. All runbooks in the same Automation account are available to all others to be used in this manner. The parent runbook will wait for the child runbook to complete before moving to the next line, and any output is returned directly to the parent.

When you invoke a runbook inline, it runs in the same job as the parent runbook. There will be no indication in the job history of the child runbook that it ran. Any exceptions and any stream output from the child runbook will be associated with the parent. This results in fewer jobs and makes them easier to track and to troubleshoot since any exceptions thrown by the child runbook and any of its stream output are associated with the parent job.

When a runbook is published, any child runbooks that it calls must already be published. This is because Azure Automation builds an association with any child runbooks when a runbook is compiled. If they aren't, the parent runbook will appear to publish properly, but will generate an exception when it's started. If this happens, you can republish the parent runbook in order to properly reference the child runbooks. You do not need to republish the parent runbook if any of the child runbooks are changed because the association will have already been created.

The parameters of a child runbook called inline can be any data type including complex objects, and there is no [JSON serialization](#) as there is when you start the runbook using the Azure Management Portal or with the `Start-AzureRmAutomationRunbook` cmdlet.

Runbook types

Which types can call each other:

- A [PowerShell runbook](#) and [Graphical runbooks](#) can call each other inline (both are PowerShell based).
- A [PowerShell Workflow runbook](#) and Graphical PowerShell Workflow runbooks can call each other inline (both are PowerShell Workflow based)
- The PowerShell types and the PowerShell Workflow types can't call each other inline, and must use `Start-AzureRmAutomationRunbook`.

When does publish order matter:

- The publish order of runbooks only matters for PowerShell Workflow and Graphical PowerShell Workflow runbooks.

When you call a Graphical or PowerShell Workflow child runbook using inline execution, you just use the name of the runbook. When you call a PowerShell child runbook, you must precede its name with `.\` to specify that the script is located in the local directory.

Example

The following example invokes a test child runbook that accepts three parameters, a complex object, an integer, and a boolean. The output of the child runbook is assigned to a variable. In this case, the child runbook is a PowerShell Workflow runbook

```
$vm = Get-AzureRmVM -ResourceGroupName "LabRG" -Name "MyVM"
$output = PSWF-ChildRunbook -VM $vm -RepeatCount 2 -Restart $true
```

Following is the same example using a PowerShell runbook as the child.

```
$vm = Get-AzureRmVM -ResourceGroupName "LabRG" -Name "MyVM"
$output = .\PS-ChildRunbook.ps1 -VM $vm -RepeatCount 2 -Restart $true
```

Starting a child runbook using cmdlet

You can use the [Start-AzureRmAutomationRunbook](#) cmdlet to start a runbook as described in [To start a runbook with Windows PowerShell](#). There are two modes of use for this cmdlet. In one mode, the cmdlet returns the job id as soon as the child job is created for the child runbook. In the other mode, which you enable by specifying the **-wait** parameter, the cmdlet will wait until the child job finishes and will return the output from the child runbook.

The job from a child runbook started with a cmdlet will run in a separate job from the parent runbook. This results in more jobs than invoking the runbook inline and makes them more difficult to track. The parent can start multiple child runbooks asynchronously without waiting for each to complete. For that same kind of parallel execution calling the child runbooks inline, the parent runbook would need to use the [parallel keyword](#).

Parameters for a child runbook started with a cmdlet are provided as a hashtable as described in [Runbook Parameters](#). Only simple data types can be used. If the runbook has a parameter with a complex data type, then it must be called inline.

Example

The following example starts a child runbook with parameters and then waits for it to complete using the `Start-AzureRmAutomationRunbook -wait` parameter. Once completed, its output is collected from the child runbook.

```
$params = @{"VMName"="MyVM"; "RepeatCount"=2; "Restart"=$true}
$joboutput = Start-AzureRmAutomationRunbook -AutomationAccountName "MyAutomationAccount" -Name "Test-ChildRunbook" -ResouceGroupName "LabRG" -Parameters $params -wait
```

Comparison of methods for calling a child runbook

The following table summarizes the differences between the two methods for calling a runbook from another runbook.

	INLINE	CMDLET
Job	Child runbooks run in the same job as the parent.	A separate job is created for the child runbook.
Execution	Parent runbook waits for the child runbook to complete before continuing.	Parent runbook continues immediately after child runbook is started or parent runbook waits for the child job to finish.
Output	Parent runbook can directly get output from child runbook.	Parent runbook must retrieve output from child runbook job or parent runbook can directly get output from child runbook.

	INLINE	CMDLET
Parameters	Values for the child runbook parameters are specified separately and can use any data type.	Values for the child runbook parameters must be combined into a single hashtable and can only include simple, array, and object data types that leverage JSON serialization.
Automation Account	Parent runbook can only use child runbook in the same automation account.	Parent runbook can use child runbook from any automation account from the same Azure subscription and even a different subscription if you have a connection to it.
Publishing	Child runbook must be published before parent runbook is published.	Child runbook must be published any time before parent runbook is started.

Next steps

- [Starting a runbook in Azure Automation](#)
- [Runbook output and messages in Azure Automation](#)

Runbook output and messages in Azure Automation

1/17/2017 • 12 min to read • [Edit on GitHub](#)

Most Azure Automation runbooks will have some form of output such as an error message to the user or a complex object intended to be consumed by another workflow. Windows PowerShell provides [multiple streams](#) to send output from a script or workflow. Azure Automation works with each of these streams differently, and you should follow best practices for how to use each when you are creating a runbook.

The following table provides a brief description of each of the streams and their behavior in the Azure Management Portal both when running a published runbook and when [testing a runbook](#). Further details on each stream are provided in subsequent sections.

STREAM	DESCRIPTION	PUBLISHED	TEST
Output	Objects intended to be consumed by other runbooks.	Written to the job history.	Displayed in the Test Output Pane.
Warning	Warning message intended for the user.	Written to the job history.	Displayed in the Test Output Pane.
Error	Error message intended for the user. Unlike an exception, the runbook continues after an error message by default.	Written to the job history.	Displayed in the Test Output Pane.
Verbose	Messages providing general or debugging information.	Written to job history only if verbose logging is turned on for the runbook.	Displayed in the Test Output pane only if \$VerbosePreference is set to Continue in the runbook.
Progress	Records automatically generated before and after each activity in the runbook. The runbook should not attempt to create its own progress records since they are intended for an interactive user.	Written to job history only if progress logging is turned on for the runbook.	Not displayed in the Test Output Pane.
Debug	Messages intended for an interactive user. Should not be used in runbooks.	Not written to job history.	Not written to Test Output Pane.

Output stream

The Output stream is intended for output of objects created by a script or workflow when it runs correctly. In Azure Automation, this stream is primarily used for objects intended to be consumed by [parent runbooks that call the current runbook](#). When you [call a runbook inline](#) from a parent runbook, it returns data from the output stream to the parent. You should only use the output stream to communicate general information back to the user if you know the runbook will never be called by another runbook. As a best practice, however, you should typically use the [Verbose Stream](#) to communicate general information to the user.

You can write data to the output stream using [Write-Output](#) or by putting the object on its own line in the runbook.

```
#The following lines both write an object to the output stream.  
Write-Output -InputObject $object  
$object
```

Output from a function

When you write to the output stream in a function that is included in your runbook, the output is passed back to the runbook. If the runbook assigns that output to a variable, then it is not written to the output stream. Writing to any other streams from within the function will write to the corresponding stream for the runbook.

Consider the following sample runbook.

```
Workflow Test-Runbook  
{  
    Write-Verbose "Verbose outside of function" -Verbose  
    Write-Output "Output outside of function"  
    $functionOutput = Test-Function  
    $functionOutput  
  
    Function Test-Function  
    {  
        Write-Verbose "Verbose inside of function" -Verbose  
        Write-Output "Output inside of function"  
    }  
}
```

The output stream for the runbook job would be:

```
Output inside of function  
Output outside of function
```

The verbose stream for the runbook job would be:

```
Verbose outside of function  
Verbose inside of function
```

Once you have published the runbook and before you start it, you must also turn on Verbose logging in the runbook settings in order to get the Verbose stream output.

Declaring output data type

A workflow can specify the data type of its output using the [OutputType attribute](#). This attribute has no effect during runtime, but it provides an indication to the runbook author at design time of the expected output of the runbook. As the toolset for runbooks continues to evolve, the importance of declaring output data types at design time will increase in importance. As a result, it is a best practice to include this declaration in any runbooks that you create.

Here is a list of example output types:

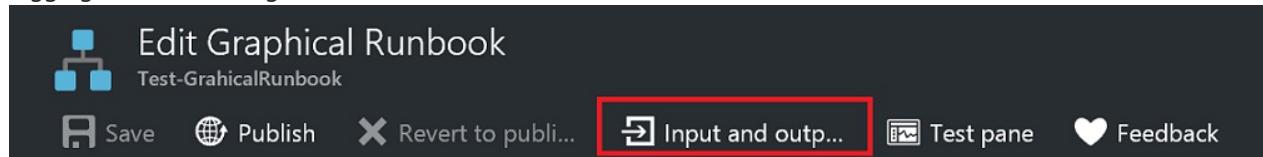
- System.String
- System.Int32
- System.Collections.Hashtable
- Microsoft.Azure.Commands.Compute.Models.PSVirtualMachine

The following sample runbook outputs a string object and includes a declaration of its output type. If your runbook outputs an array of a certain type, then you should still specify the type as opposed to an array of the type.

```
Workflow Test-Runbook
{
    [OutputType([string])]

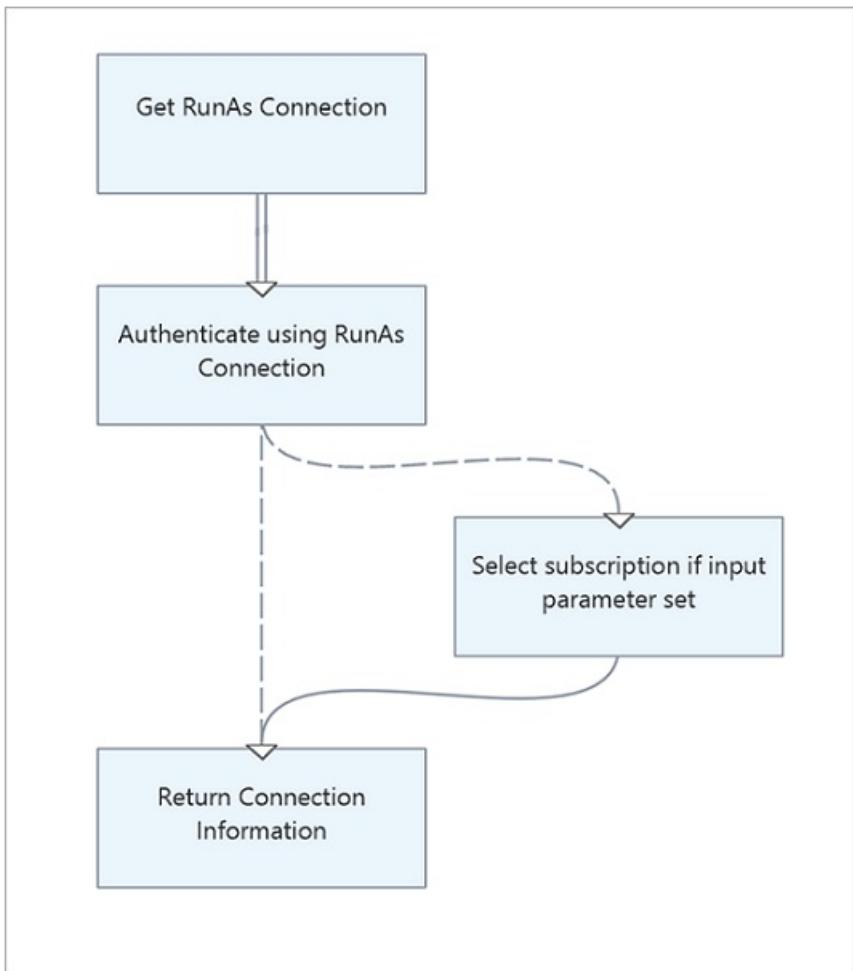
    $output = "This is some string output."
    Write-Output $output
}
```

To declare an output type in Graphical or Graphical PowerShell Workflow runbooks, you can select the **Input and Output** menu option and type in the name of the output type. We recommend you use the full .NET class name to make it easily identifiable when referencing it from a parent runbook. This exposes all the properties of that class to the data bus in the runbook and provides a lot of flexibility when using them for conditional logic, logging, and referencing as values for other activities in the runbook.

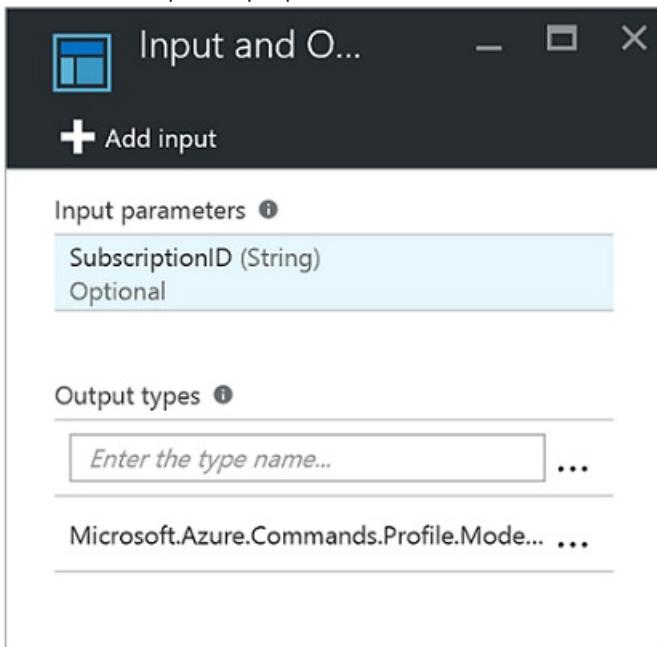


In the following example, we have two graphical runbooks to demonstrate this feature. If we apply the modular runbook design model, we have one runbook which serves as the *Authentication Runbook template* managing authentication with Azure using the Run As account. Our second runbook, which would normally perform the core logic to automate a given scenario, in this case is going to execute the *Authentication Runbook template* and display the results to your **Test** output pane. Under normal circumstances, we would have this runbook do something against a resource leveraging the output from the child runbook.

Here is the basic logic of the **AuthenticateTo-Azure** runbook.

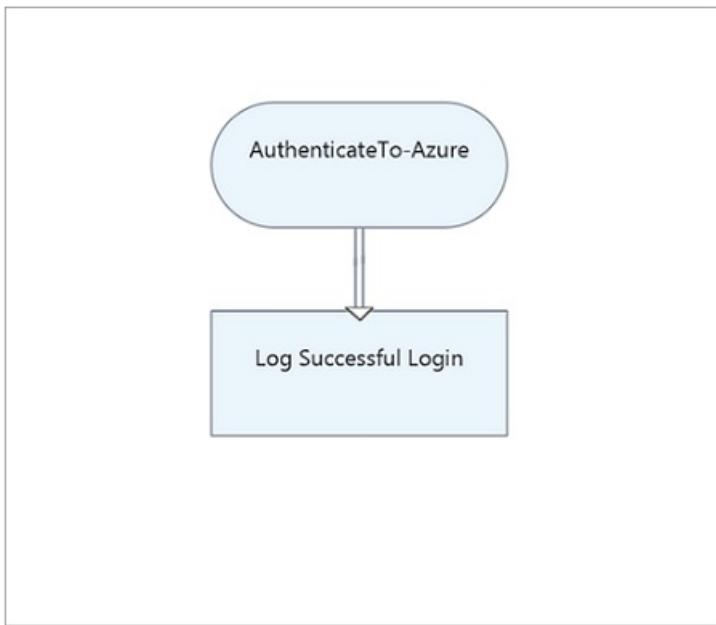


It includes the output type `Microsoft.Azure.Commands.Profile.Models.PSAzureContext`, which will return the authentication profile properties.



While this runbook is very straight forward, there is one configuration item to call out here. The last activity is executing the **Write-Output** cmdlet and writes the profile data to a `$_` variable using a PowerShell expression for the **InputObject** parameter, which is required for that cmdlet.

For the second runbook in this example, named `Test-ChildOutputType`, we simply have two activities.



The first activity calls the **AuthenticateTo-Azure** runbook and the second activity is running the **Write-Verbose** cmdlet with the **Data source of Activity output** and the value for **Field path** is **Context.Subscription.SubscriptionName**, which is specifying the context output from the **AuthenticateTo-Azure** runbook.

Activity Parameter Configuration		Parameter Value
Name Write-Verbose	✓ MESSAGE ⓘ AuthenticateTo-Azure (Activity output)	Data source Activity output
* Label ⓘ Log Successful Login		Select data
Comment		▼ AuthenticateTo-Azure
Parameters 1 of 1 configured		▼ Output (PSAzureContext)
Optional additional parameters Configure parameters		► Account (PSAzureRmAccount)
Retry behavior Configure retry behavior		► Environment (PSAzureEnvironment)
		► Subscription (PSAzureSubscription)
		CurrentStorageAccountName (String)
		SubscriptionId (String)
		SubscriptionName (String)
		TenantId (String)
		► Tenant (PSAzureTenant)
		Selected activity name AuthenticateTo-Azure
		Field path ⓘ Context.Subscription.SubscriptionName

The resulting output is the name of the subscription.

Test	
Test-ChildOutputType	Completed
Parameters	No input parameters
Run Settings	Run on Azure ⓘ
Microsoft Azure Internal Consumption	

One note about the behavior of the Output Type control. When you type a value in the Output Type field on the Input and Output properties blade, you have to click outside of the control after you type it, in order for your entry

to be recognized by the control.

Message streams

Unlike the output stream, message streams are intended to communicate information to the user. There are multiple message streams for different kinds of information, and each is handled differently by Azure Automation.

Warning and error streams

The Warning and Error streams are intended to log problems that occur in a runbook. They are written to the job history when a runbook is executed, and are included in the Test Output Pane in the Azure Management Portal when a runbook is tested. By default, the runbook will continue executing after a warning or error. You can specify that the runbook should be suspended on a warning or error by setting a [preference variable](#) in the runbook before creating the message. For example, to cause a runbook to suspend on an error as it would an exception, set **\$ErrorActionPreference** to Stop.

Create a warning or error message using the [Write-Warning](#) or [Write-Error](#) cmdlet. Activities may also write to these streams.

```
#The following lines create a warning message and then an error message that will suspend the runbook.

$ErrorActionPreference = "Stop"
Write-Warning -Message "This is a warning message."
Write-Error -Message "This is an error message that will stop the runbook because of the preference
variable."
```

Verbose stream

The Verbose message stream is for general information about the runbook operation. Since the [Debug Stream](#) is not available in a runbook, verbose messages should be used for debug information. By default, verbose messages from published runbooks will not be stored in the job history. To store verbose messages, configure published runbooks to Log Verbose Records on the Configure tab of the runbook in the Azure Management Portal. In most cases, you should keep the default setting of not logging verbose records for a runbook for performance reasons. Turn on this option only to troubleshoot or debug a runbook.

When [testing a runbook](#), verbose messages are not displayed even if the runbook is configured to log verbose records. To display verbose messages while [testing a runbook](#), you must set the **\$VerbosePreference** variable to Continue. With that variable set, verbose messages will be displayed in the Test Output Pane of the Azure portal.

Create a verbose message using the [Write-Verbose](#) cmdlet.

```
#The following line creates a verbose message.

Write-Verbose -Message "This is a verbose message."
```

Debug stream

The Debug stream is intended for use with an interactive user and should not be used in runbooks.

Progress records

If you configure a runbook to log progress records (on the Configure tab of the runbook in the Azure portal), then a record will be written to the job history before and after each activity is run. In most cases, you should keep the default setting of not logging progress records for a runbook in order to maximize performance. Turn on this option only to troubleshoot or debug a runbook. When testing a runbook, progress messages are not displayed even if the runbook is configured to log progress records.

The [Write-Progress](#) cmdlet is not valid in a runbook, since this is intended for use with an interactive user.

Preference variables

Windows PowerShell uses [preference variables](#) to determine how to respond to data sent to different output streams. You can set these variables in a runbook to control how it will respond to data sent into different streams.

The following table lists the preference variables that can be used in runbooks with their valid and default values. Note that this table only includes the values that are valid in a runbook. Additional values are valid for the preference variables when used in Windows PowerShell outside of Azure Automation.

VARIABLE	DEFAULT VALUE	VALID VALUES
WarningPreference	Continue	Stop Continue SilentlyContinue
ErrorActionPreference	Continue	Stop Continue SilentlyContinue
VerbosePreference	SilentlyContinue	Stop Continue SilentlyContinue

The following table lists the behavior for the preference variable values that are valid in runbooks.

VALUE	BEHAVIOR
Continue	Logs the message and continues executing the runbook.
SilentlyContinue	Continues executing the runbook without logging the message. This has the effect of ignoring the message.
Stop	Logs the message and suspends the runbook.

Retrieving runbook output and messages

Azure portal

You can view the details of a runbook job in the Azure portal from the Jobs tab of a runbook. The Summary of the job will display the input parameters and the [Output Stream](#) in addition to general information about the job and any exceptions if they occurred. The History will include messages from the [Output Stream](#) and [Warning and Error Streams](#) in addition to the [Verbose Stream](#) and [Progress Records](#) if the runbook is configured to log verbose and progress records.

Windows PowerShell

In Windows PowerShell, you can retrieve output and messages from a runbook using the [Get-AzureAutomationJobOutput](#) cmdlet. This cmdlet requires the ID of the job and has a parameter called Stream where you specify which stream to return. You can specify Any to return all streams for the job.

The following example starts a sample runbook and then waits for it to complete. Once completed, its output stream is collected from the job.

```

$job = Start-AzureRmAutomationRunbook -ResourceGroupName "ResourceGroup01" ` 
-AutomationAccountName "MyAutomationAccount" -Name "Test-Runbook"

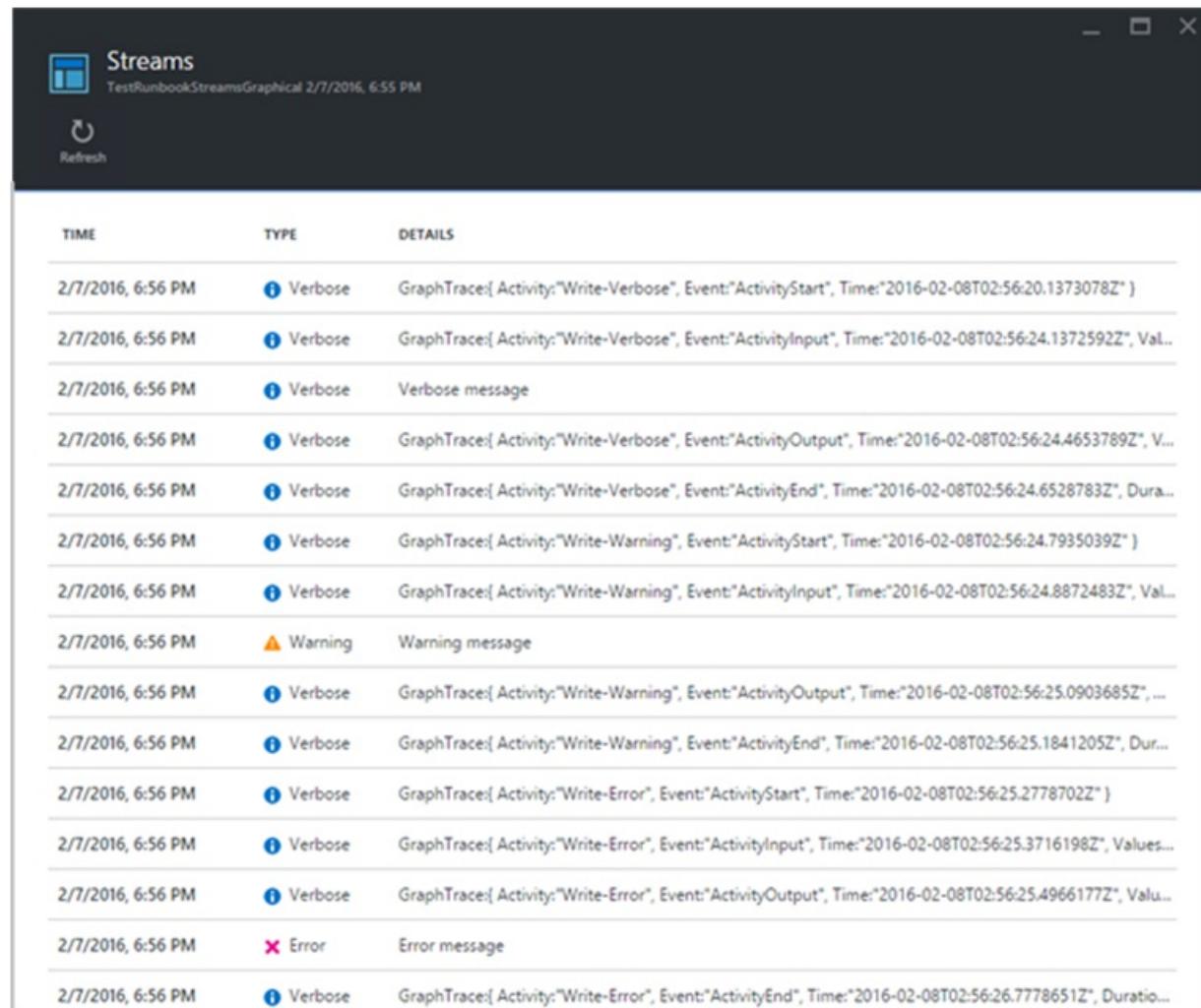
$doLoop = $true
While ($doLoop) {
    $job = Get-AzureRmAutomationJob -ResourceGroupName "ResourceGroup01" ` 
    -AutomationAccountName "MyAutomationAccount" -Id $job.JobId
    $status = $job.Status
    $doLoop = (($status -ne "Completed") -and ($status -ne "Failed") -and ($status -ne "Suspended") -and 
    ($status -ne "Stopped"))
}

Get-AzureRmAutomationJobOutput -ResourceGroupName "ResourceGroup01" ` 
-AutomationAccountName "MyAutomationAccount" -Id $job.JobId -Stream Output

```

Graphical Authoring

For graphical runbooks, extra logging is available in the form of activity-level tracing. There are two levels of tracing: Basic and Detailed. In Basic tracing, you can see the start and end time of each activity in the runbook plus information related to any activity retries, such as number of attempts and start time of the activity. In Detailed tracing, you get Basic tracing plus input and output data for each activity. Note that currently the trace records are written using the verbose stream, so you must enable Verbose logging when you enable tracing. For graphical runbooks with tracing enabled there is no need to log progress records, because the Basic tracing serves the same purpose and is more informative.



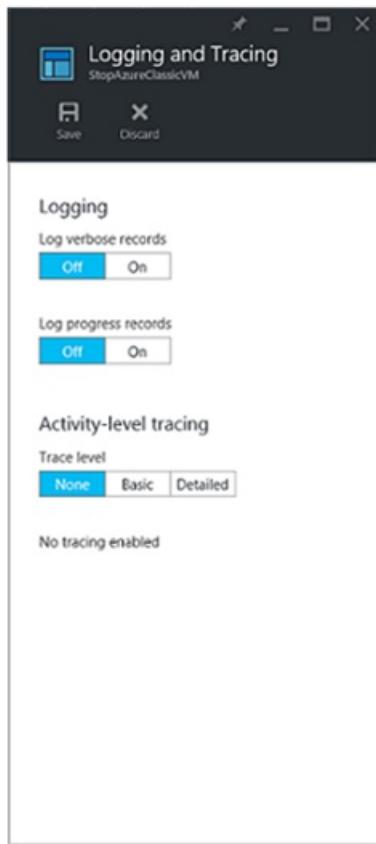
TIME	TYPE	DETAILS
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Verbose", Event:"ActivityStart", Time:"2016-02-08T02:56:20.1373078Z" }
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Verbose", Event:"ActivityInput", Time:"2016-02-08T02:56:24.1372592Z", Val...
2/7/2016, 6:56 PM	Verbose	Verbose message
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Verbose", Event:"ActivityOutput", Time:"2016-02-08T02:56:24.4653789Z", V...
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Verbose", Event:"ActivityEnd", Time:"2016-02-08T02:56:24.6528783Z", Dura...
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Warning", Event:"ActivityStart", Time:"2016-02-08T02:56:24.7935039Z" }
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Warning", Event:"ActivityInput", Time:"2016-02-08T02:56:24.8872483Z", Val...
2/7/2016, 6:56 PM	Warning	Warning message
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Warning", Event:"ActivityOutput", Time:"2016-02-08T02:56:25.0903685Z", ...
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Warning", Event:"ActivityEnd", Time:"2016-02-08T02:56:25.1841205Z", Dur...
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Error", Event:"ActivityStart", Time:"2016-02-08T02:56:25.2778702Z" }
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Error", Event:"ActivityInput", Time:"2016-02-08T02:56:25.3716198Z", Values...
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Error", Event:"ActivityOutput", Time:"2016-02-08T02:56:25.4966177Z", Valu...
2/7/2016, 6:56 PM	Error	Error message
2/7/2016, 6:56 PM	Verbose	GraphTrace{ Activity:"Write-Error", Event:"ActivityEnd", Time:"2016-02-08T02:56:26.7778651Z", Duratio...

You can see from the above screenshot that when you enable Verbose logging and tracing for Graphical runbooks, much more information is available in the production Job Streams view. This extra information can be essential for troubleshooting production problems with a runbook, and therefore you should only enable it for that purpose and not as a general practice.

The Trace records can be especially numerous. With Graphical runbook tracing you can get two to four records per activity depending on whether you have configured Basic or Detailed tracing. Unless you need this information to track the progress of a runbook for troubleshooting, you might want to keep Tracing turned off.

To enable activity-level tracing, perform the following steps.

1. In the Azure Portal, open your Automation account.
2. Click on the **Runbooks** tile to open the list of runbooks.
3. On the Runbooks blade, click to select a graphical runbook from your list of runbooks.
4. On the Settings blade for the selected runbook, click **Logging and Tracing**.
5. On the Logging and Tracing blade, under Log verbose records, click **On** to enable verbose logging and under Activity-level tracing, change the trace level to **Basic** or **Detailed** based on the level of tracing you require.



Microsoft Operations Management Suite (OMS) Log Analytics

Automation can send runbook job status and job streams to your Microsoft Operations Management Suite (OMS) Log Analytics workspace. With Log Analytics you can,

- Get insight on your Automation jobs
- Trigger an email or alert based on your runbook job status (e.g. failed or suspended)
- Write advanced queries across your job streams
- Correlate jobs across Automation accounts
- Visualize your job history over time

For more information on how to configure integration with Log Analytics to collect, correlate and act on job data, see [Forward job status and job streams from Automation to Log Analytics \(OMS\)](#).

Next steps

- To learn more about runbook execution, how to monitor runbook jobs, and other technical details, see [Track a runbook job](#)

- To understand how to design and use child runbooks, see [Child runbooks in Azure Automation](#)

Source control integration in Azure Automation

1/17/2017 • 6 min to read • [Edit on GitHub](#)

Source control integration allows you to associate runbooks in your Automation account to a GitHub source control repository. Source control allows you to easily collaborate with your team, track changes, and roll back to earlier versions of your runbooks. For example, source control allows you to sync different branches in source control to your development, test or production Automation accounts, making it easy to promote code that has been tested in your development environment to your production Automation account.

Source control allows you to push code from Azure Automation to source control or pull your runbooks from source control to Azure Automation. This article describes how to set up source control in your Azure Automation environment. We will start by configuring Azure Automation to access your GitHub repository and walk through different operations that can be done using source control integration.

NOTE

Source control supports pulling and pushing [PowerShell Workflow runbooks](#) as well as [PowerShell runbooks](#). [Graphical runbooks](#) are not yet supported.

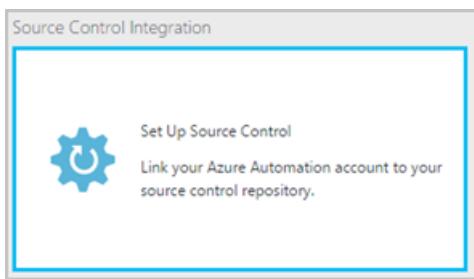
There are two simple steps required to configure source control for your Automation account, and only one if you already have a GitHub account. They are:

Step 1 – Create a GitHub repository

If you already have a GitHub account and a repository that you want to link to Azure Automation, then login to your existing account and start from step 2 below. Otherwise, navigate to [GitHub](#), sign up for a new account and [create a new repository](#).

Step 2 – Set up source control in Azure Automation

- From the Automation Account blade in the Azure portal, click **Set Up Source Control**.

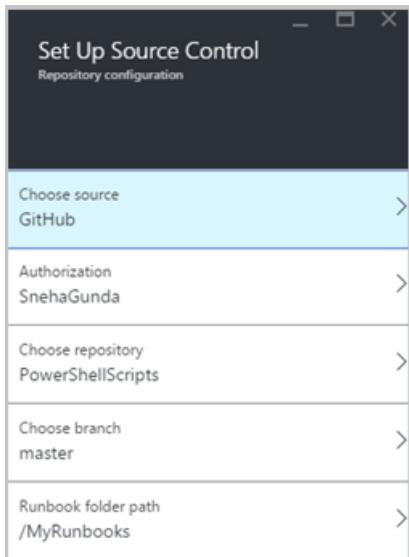


- The **Source Control** blade opens, where you can configure your GitHub account details. Below is the list of parameters to configure:

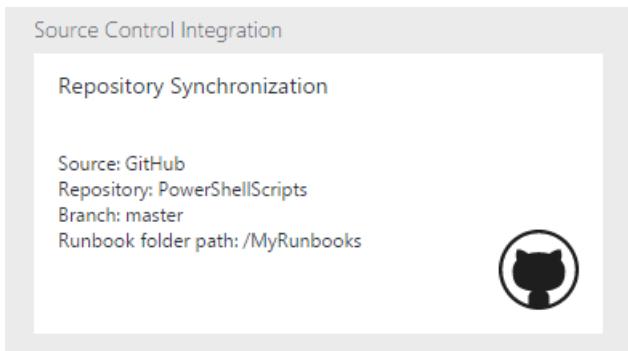
PARAMETER	DESCRIPTION
Choose Source	Select the source. Currently, only GitHub is supported.

PARAMETER	DESCRIPTION
Authorization	Click the Authorize button to grant Azure Automation access to your GitHub repository. If you are already logged in to your GitHub account in a different window, then the credentials of that account are used. Once authorization is successful, the blade will show your GitHub username under Authorization Property .
Choose repository	Select a GitHub repository from the list of available repositories.
Choose branch	Select a branch from the list of available branches. Only the master branch is shown if you haven't created any branches.
Runbook folder path	The runbook folder path specifies the path in the GitHub repository from which you want to push or pull your code. It must be entered in the format /foldername/subfoldername . Only runbooks in the runbook folder path will be synced to your Automation account. Runbooks in the subfolders of the runbook folder path will NOT be synced. Use / to sync all the runbooks under the repository.

3. For example, if you have a repository named **PowerShellScripts** that contains a folder named **RootFolder**, which contains a folder named **SubFolder**. You can use the following strings to sync each folder level:
 - a. To sync runbooks from **repository**, runbook folder path is /
 - b. To sync runbooks from **RootFolder**, runbook folder path is /RootFolder
 - c. To sync runbooks from **SubFolder**, runbook folder path is /RootFolder/SubFolder.
4. After you configure the parameters, they are displayed on the **Set Up Source Control** blade.



5. Once you click OK, source control integration is now configured for your Automation account and should be updated with your GitHub information. You can now click on this part to view all of your source control sync job history.



6. After you set up source control, the following Automation resources will be created in your Automation account:

Two **variable assets** are created.

- The variable **Microsoft.Azure.Automation.SourceControl.Connection** contains the values of the connection string, as shown below.

PARAMETER	VALUE
Name	Microsoft.Azure.Automation.SourceControl.Connection
Type	String
Value	{"Branch":<Your branch name>,"RunbookFolderPath":<Runbook folder path>,"ProviderType":<has a value 1 for GitHub>,"Repository":<Name of your repository>,"Username":<Your GitHub user name>}

- The variable **Microsoft.Azure.Automation.SourceControl.OAuthToken**, contains the secure encrypted value of your OAuthToken.

PARAMETER	VALUE
Name	Microsoft.Azure.Automation.SourceControl.OAuthToke n
Type	Unknown(Encrypted)
Value	<Encrypted OAuthToken>

The screenshot shows the 'Variables' blade in the Azure portal. It lists two variables:

NAME	TYPE	VALUE	LAST MODIFIED
Microsoft.Azure.Automation.SourceControl.Connection	String	{"Branch":"master","RunbookFolderPath":"/MyRunbooks","ProviderType":1,"Repository":"PowerShellScripts","Username":<Your GitHub user name>}	10/27/2015, 7:31 PM
Microsoft.Azure.Automation.SourceControl.OAuthToken	Unknown (encrypted)	*****	10/27/2015, 7:29 PM

- Automation Source Control** is added as an authorized application to your GitHub account. To view the application: From your GitHub home page, navigate to your **profile > Settings > Applications**. This application allows Azure Automation to sync your GitHub repository to an Automation account.

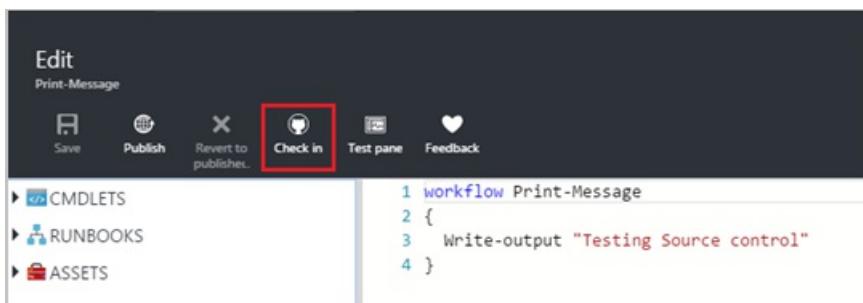
The screenshot shows the GitHub 'Personal settings' page under the 'Authorized applications' tab. A red box highlights the 'Applications' link in the sidebar. The main area lists two authorized applications: 'Automation Source Control' (last used within the last 2 weeks) and 'Azure Open Source Portal' (last used within the last 2 months). The 'Automation Source Control' card has a red border around it. On the right side, there's a sidebar with links like 'Your profile', 'Your stars', 'Explore', 'Integrations', 'Help', 'Settings' (which is also highlighted with a red box), and 'Sign out'. A 'Revoke' button is located at the bottom right of the application list.

Using Source Control in Automation

Check-in a runbook from Azure Automation to source control

Runbook check-in allows you to push the changes you have made to a runbook in Azure Automation into your source control repository. Below are the steps to check-in a runbook:

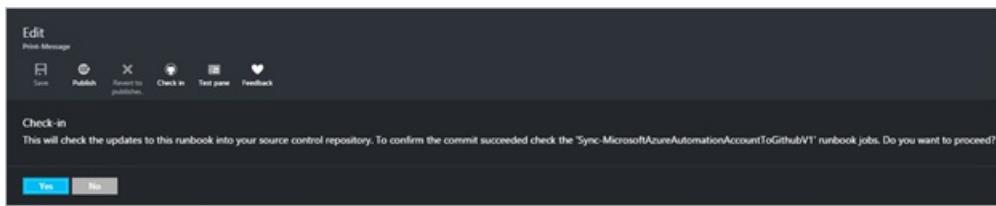
1. From your Automation Account, [create a new textual runbook](#), or [edit an existing, textual runbook](#). This runbook can be either a PowerShell Workflow or a PowerShell script runbook.
2. After you edit your runbook, save it and click **check-in** from the **Edit** blade.



NOTE

Check-in from Azure Automation will overwrite the code that currently exists in your source control. The Git equivalent command line instruction to check-in is **git add + git commit + git push**

3. When you click **check-in**, you will be prompted with a confirmation message, click yes to continue.



4. Check-in starts the source control runbook: **Sync-MicrosoftAzureAutomationAccountToGitHubV1**. This runbook connects to GitHub and pushes changes from Azure Automation to your repository. To view the check-in job history, go back to the **Source Control Integration** tab and click to open the Repository Synchronization blade. This blade shows all of your source control jobs. Select the job you want to view and click to view the details.

Repository Synchronization			
Jobs			
STATUS	RUNBOOK	CREATED	LAST UPDATED
✓ Completed	Sync-MicrosoftAzureAutomationAccountToGit...	10/27/2015, 9:14 PM	10/27/2015, 9:16 PM

NOTE

Source control runbooks are special Automation runbooks that you cannot view or edit. While they will not show up on your runbook list, you will see sync jobs showing up on your jobs list.

5. The name of the modified runbook is sent as an input parameter to the check-in runbook. You can [view the job details](#) by expanding runbook in **Repository Synchronization** blade.

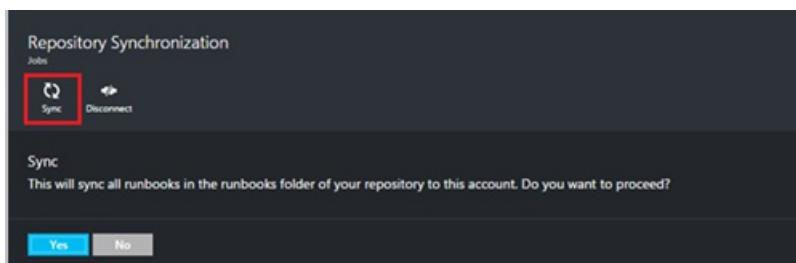
NAME	INPUT VALUE
RUNBOOKNAME	Print-Message

6. Refresh your GitHub repository once the job completes to view the changes. There should be a commit in your repository with a commit message: **Updated Runbook Name in Azure Automation**.

Sync runbooks from source control to Azure Automation

The sync button on the Repository Synchronization blade allows you to pull all the runbooks in the runbook folder path of your repository to your Automation account. The same repository can be synced to more than one Automation account. Below are the steps to sync a runbook:

1. From the Automation account where you set up source control, open the **Source Control Integration/Repository Synchronization blade** and click **Sync** then you will be prompted with a confirmation message, click **Yes** to continue.



2. Sync starts the runbook: **Sync-MicrosoftAzureAutomationAccountFromGitHubV1**. This runbook connects to GitHub and pulls the changes from your repository to Azure Automation. You should see a new job on the **Repository Synchronization** blade for this action. To view details about the sync job, click to open the job details blade.

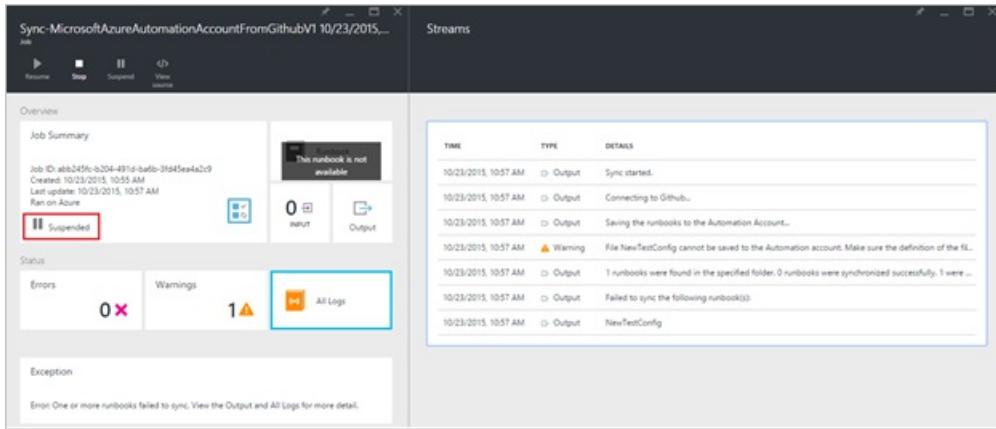
Repository Synchronization			
Jobs			
STATUS	RUNBOOK	CREATED	LAST UPDATED
✓ Completed	Sync-MicrosoftAzureAutomationAccountFrom...	10/27/2015, 10:15 PM	10/27/2015, 10:16 PM

NOTE

A sync from source control overwrites the draft version of the runbooks that currently exist in your Automation account for **ALL** runbooks that are currently in source control. The Git equivalent command line instruction to sync is **git pull**

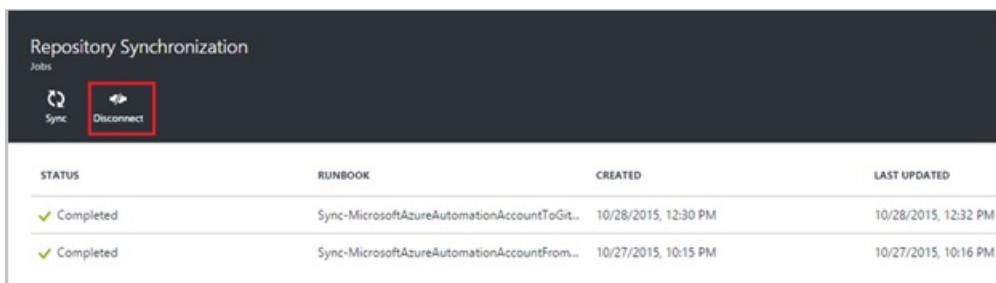
Troubleshooting source control problems

If there are any errors with a check-in or sync job, the job status should be suspended and you can view more details about the error in the job blade. The **All Logs** part will show you all the PowerShell streams associated with that job. This will provide you with the details needed to help you fix any problems with your check-in or sync. It will also show you the sequence of actions that occurred while syncing or checking-in a runbook.



Disconnecting source control

To disconnect from your GitHub account, open the Repository Synchronization blade and click **Disconnect**. Once you disconnect source control, runbooks that were synced earlier will still remain in your Automation account but the Repository Synchronization blade will not be enabled.



Next steps

For more information about source control integration, see the following resources:

- [Azure Automation: Source Control Integration in Azure Automation](#)
- [Vote for your favorite source control system](#)
- [Azure Automation: Integrating Runbook Source Control using Visual Studio Team Services](#)

Starting a runbook in Azure Automation

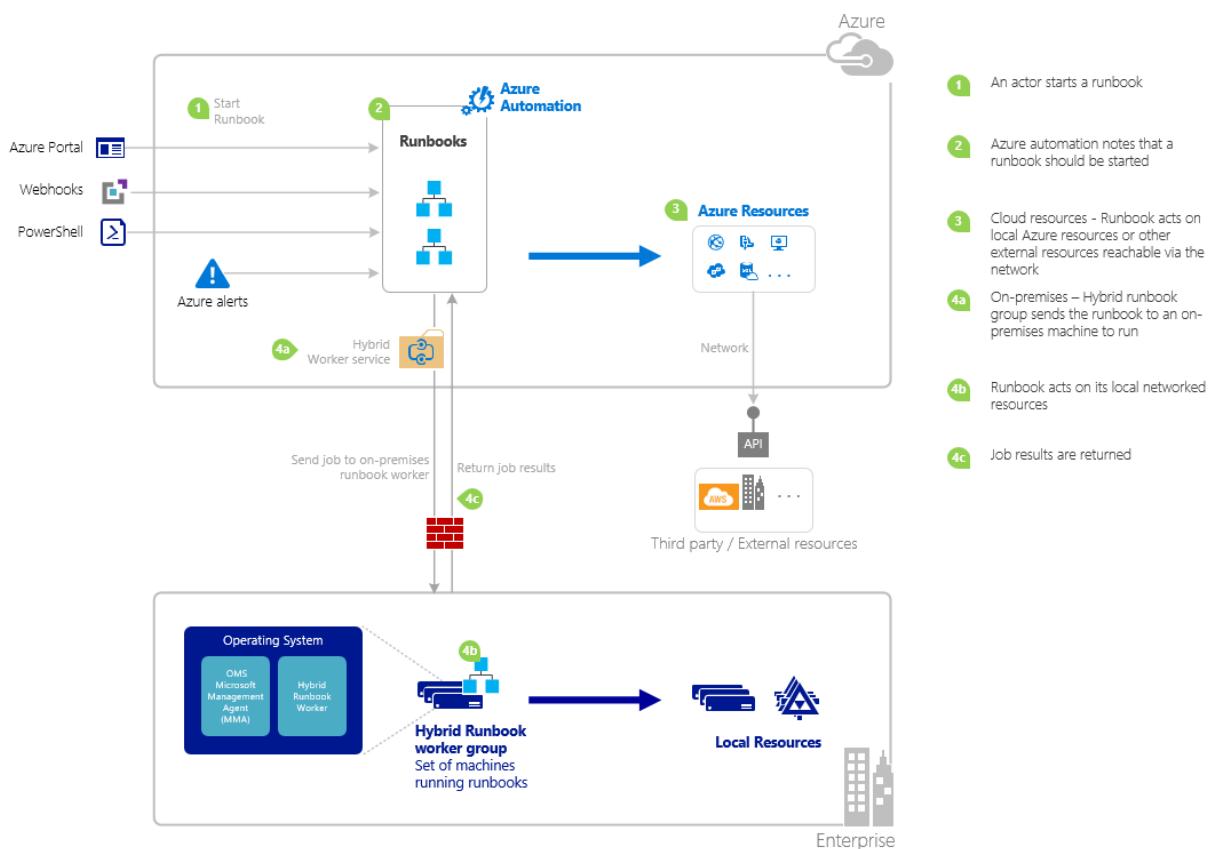
1/17/2017 • 5 min to read • [Edit on GitHub](#)

The following table will help you determine the method to start a runbook in Azure Automation that is most suitable to your particular scenario. This article includes details on starting a runbook with the Azure portal and with Windows PowerShell. Details on the other methods are provided in other documentation that you can access from the links below.

METHOD	CHARACTERISTICS
Azure Portal	<ul style="list-style-type: none">• Simplest method with interactive user interface.• Form to provide simple parameter values.• Easily track job state.• Access authenticated with Azure logon.
Windows PowerShell	<ul style="list-style-type: none">• Call from command line with Windows PowerShell cmdlets.• Can be included in automated solution with multiple steps.• Request is authenticated with certificate or OAuth user principal / service principal.• Provide simple and complex parameter values.• Track job state.• Client required to support PowerShell cmdlets.
Azure Automation API	<ul style="list-style-type: none">• Most flexible method but also most complex.• Call from any custom code that can make HTTP requests.• Request authenticated with certificate, or OAuth user principal / service principal.• Provide simple and complex parameter values.• Track job state.
Webhooks	<ul style="list-style-type: none">• Start runbook from single HTTP request.• Authenticated with security token in URL.• Client cannot override parameter values specified when webhook created. Runbook can define single parameter that is populated with the HTTP request details.• No ability to track job state through webhook URL.
Respond to Azure Alert	<ul style="list-style-type: none">• Start a runbook in response to Azure alert.• Configure webhook for runbook and link to alert.• Authenticated with security token in URL.• Currently supports alert on Metrics only.
Schedule	<ul style="list-style-type: none">• Automatically start runbook on hourly, daily, or weekly schedule.• Manipulate schedule through Azure portal, PowerShell cmdlets, or Azure API.• Provide parameter values to be used with schedule.
From Another Runbook	<ul style="list-style-type: none">• Use a runbook as an activity in another runbook.• Useful for functionality used by multiple runbooks.• Provide parameter values to child runbook and use output in parent runbook.

The following image illustrates detailed step-by-step process in the life cycle of a runbook. It includes different ways a runbook is started in Azure Automation, components required for Hybrid Runbook Worker to execute

Azure Automation runbooks and interactions between different components. To learn about executing Automation runbooks in your datacenter, refer to [hybrid runbook workers](#)



Starting a runbook with the Azure portal

1. In the Azure portal, select **Automation** and then click the name of an automation account.
2. Select the **Runbooks** tab.
3. Select a runbook, and then click **Start**.
4. If the runbook has parameters, you will be prompted to provide values with a text box for each parameter. See [Runbook Parameters](#) below for further details on parameters.
5. Either select **View Job** next to the **Starting** runbook message or select the **Jobs** tab for the runbook to view the runbook job's status.

Starting a runbook with the Azure portal

1. From your automation account, click the **Runbooks** part to open the **Runbooks** blade.
2. Click a runbook to open its **Runbook** blade.
3. Click **Start**.
4. If the runbook has no parameters, you will be prompted to confirm whether you want to start it. If the runbook has parameters, the **Start Runbook** blade will be opened so you can provide parameter values. See [Runbook Parameters](#) below for further details on parameters.
5. The **Job** blade is opened so that you can track the job's status.

Starting a runbook with Windows PowerShell

You can use the [Start-AzureRmAutomationRunbook](#) to start a runbook with Windows PowerShell. The following sample code starts a runbook called Test-Runbook.

```
Start-AzureRmAutomationRunbook -AutomationAccountName "MyAutomationAccount" -Name "Test-Runbook" -  
ResourceGroupName "ResourceGroup01"
```

Start-AzureRmAutomationRunbook returns a job object that you can use to track its status once the runbook is started. You can then use this job object with [Get-AzureRmAutomationJob](#) to determine the status of the job and [Get-AzureRmAutomationJobOutput](#) to get its output. The following sample code starts a runbook called Test-Runbook, waits until it has completed, and then displays its output.

```
$runbookName = "Test-Runbook"  
$ResourceGroup = "ResourceGroup01"  
$AutomationAcct = "MyAutomationAccount"  
  
$job = Start-AzureRmAutomationRunbook -AutomationAccountName $AutomationAcct -Name $runbookName -  
ResourceGroupName $ResourceGroup  
  
$doLoop = $true  
While ($doLoop) {  
    $job = Get-AzureRmAutomationJob -AutomationAccountName $AutomationAcct -Id $job.JobId -ResourceGroupName  
$ResourceGroup  
    $status = $job.Status  
    $doLoop = (($status -ne "Completed") -and ($status -ne "Failed") -and ($status -ne "Suspended") -and  
($status -ne "Stopped"))  
}  
  
Get-AzureRmAutomationJobOutput -AutomationAccountName $AutomationAcct -Id $job.JobId -ResourceGroupName  
$ResourceGroup -Stream Output
```

If the runbook requires parameters, then you must provide them as a [hashtable](#) where the key of the hashtable matches the parameter name and the value is the parameter value. The following example shows how to start a runbook with two string parameters named FirstName and LastName, an integer named RepeatCount, and a boolean parameter named Show. For additional information on parameters, see [Runbook Parameters](#) below.

```
$params = @{"FirstName"="Joe";"LastName"="Smith";"RepeatCount"=2;"Show"=$true}  
Start-AzureRmAutomationRunbook -AutomationAccountName "MyAutomationAccount" -Name "Test-Runbook" -  
ResourceGroupName "ResourceGroup01" -Parameters $params
```

Runbook parameters

When you start a runbook from the Azure Portal or Windows PowerShell, the instruction is sent through the Azure Automation web service. This service does not support parameters with complex data types. If you need to provide a value for a complex parameter, then you must call it inline from another runbook as described in [Child runbooks in Azure Automation](#).

The Azure Automation web service will provide special functionality for parameters using certain data types as described in the following sections.

Named values

If the parameter is data type [object], then you can use the following JSON format to send it a list of named values: `{Name1:'Value1', Name2:'Value2', Name3:'Value3'}`. These values must be simple types. The runbook will receive the parameter as a [PSCustomObject](#) with properties that correspond to each named value.

Consider the following test runbook that accepts a parameter called user.

```

Workflow Test-Parameters
{
    param (
        [Parameter(Mandatory=$true)][object]$user
    )
    $userObject = $user | ConvertFrom-JSON
    if ($userObject.Show) {
        foreach ($i in 1..$userObject.RepeatCount) {
            $userObject.FirstName
            $userObject.LastName
        }
    }
}

```

The following text could be used for the user parameter.

```
{FirstName:'Joe',LastName:'Smith',RepeatCount:'2',Show:'True'}
```

This results in the following output.

```

Joe
Smith
Joe
Smith

```

Arrays

If the parameter is an array such as [array] or [string[]], then you can use the following JSON format to send it a list of values: `[Value1,Value2,Value3]`. These values must be simple types.

Consider the following test runbook that accepts a parameter called `user`.

```

Workflow Test-Parameters
{
    param (
        [Parameter(Mandatory=$true)][array]$user
    )
    if ($user[3]) {
        foreach ($i in 1..$user[2]) {
            $user[0]
            $user[1]
        }
    }
}

```

The following text could be used for the user parameter.

```
["Joe","Smith",2,true]
```

This results in the following output.

```

Joe
Smith
Joe
Smith

```

Credentials

If the parameter is data type **PSCredential**, then you can provide the name of an Azure Automation [credential asset](#). The runbook will retrieve the credential with the name that you specify.

Consider the following test runbook that accepts a parameter called credential.

```
Workflow Test-Parameters
{
    param (
        [Parameter(Mandatory=$true)][PSCredential]$credential
    )
    $credential.UserName
}
```

The following text could be used for the user parameter assuming that there was a credential asset called *My Credential*.

```
My Credential
```

Assuming the username in the credential was *jsmith*, this results in the following output.

```
jsmith
```

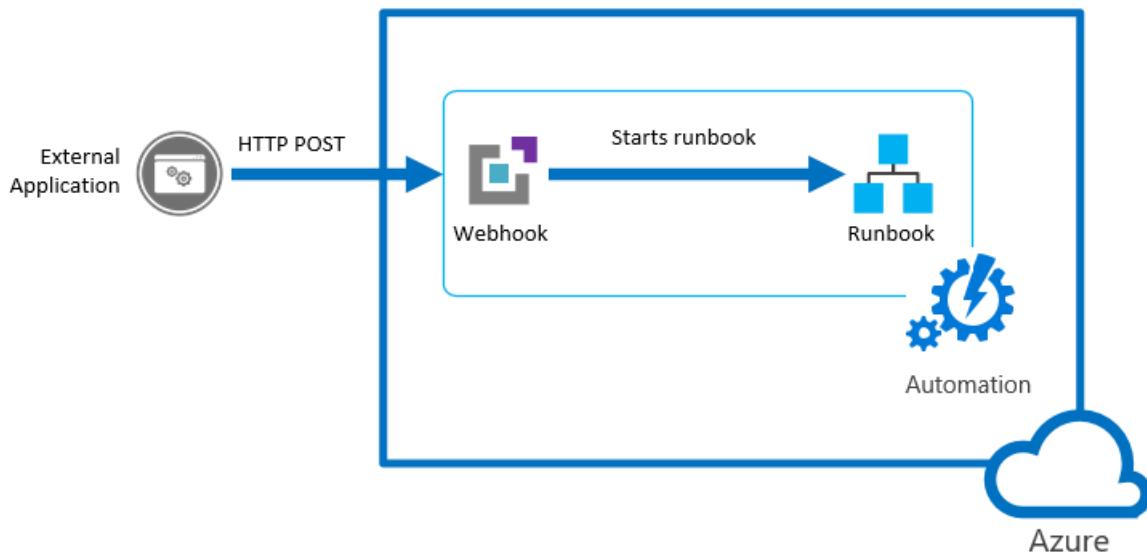
Next steps

- The runbook architecture in current article provides a high-level overview of runbooks managing resources in Azure and on-premise with the Hybrid Runbook Worker. To learn about executing Automation runbooks in your datacenter, refer to [Hybrid Runbook Workers](#).
- To learn more about the creating modular runbooks to be used by other runbooks for specific or common functions, refer to [Child Runbooks](#).

Starting an Azure Automation runbook with a webhook

2/22/2017 • 12 min to read • [Edit on GitHub](#)

A *webhook* allows you to start a particular runbook in Azure Automation through a single HTTP request. This allows external services such as Visual Studio Team Services, GitHub, Microsoft Operations Management Suite Log Analytics, or custom applications to start runbooks without implementing a full solution using the Azure Automation API.



You can compare webhooks to other methods of starting a runbook in [Starting a runbook in Azure Automation](#)

Details of a webhook

The following table describes the properties that you must configure for a webhook.

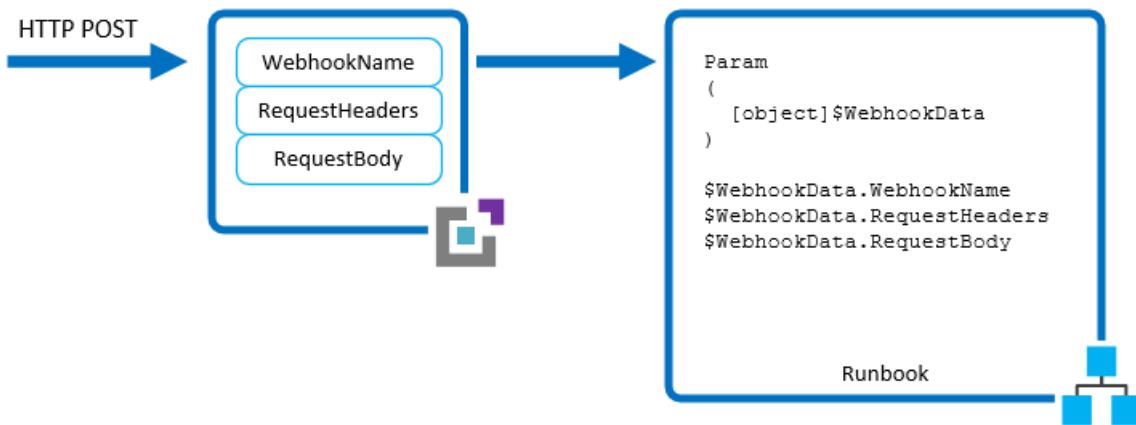
PROPERTY	DESCRIPTION
Name	You can provide any name you want for a webhook since this is not exposed to the client. It is only used for you to identify the runbook in Azure Automation. As a best practice, you should give the webhook a name related to the client that will use it.
URL	The URL of the webhook is the unique address that a client calls with an HTTP POST to start the runbook linked to the webhook. It is automatically generated when you create the webhook. You cannot specify a custom URL. The URL contains a security token that allows the runbook to be invoked by a third party system with no further authentication. For this reason, it should be treated like a password. For security reasons, you can only view the URL in the Azure portal at the time the webhook is created. You should note the URL in a secure location for future use.

PROPERTY	DESCRIPTION
Expiration date	Like a certificate, each webhook has an expiration date at which time it can no longer be used. This expiration date can be modified after the webhook is created.
Enabled	A webhook is enabled by default when it is created. If you set it to Disabled, then no client will be able to use it. You can set the Enabled property when you create the webhook or anytime once it is created.

Parameters

A webhook can define values for runbook parameters that are used when the runbook is started by that webhook. The webhook must include values for any mandatory parameters of the runbook and may include values for optional parameters. A parameter value configured to a webhook can be modified even after creating the webhook. Multiple webhooks linked to a single runbook can each use different parameter values.

When a client starts a runbook using a webhook, it cannot override the parameter values defined in the webhook. To receive data from the client, the runbook can accept a single parameter called **\$WebhookData** of type [object] that will contain data that the client includes in the POST request.



The **\$WebhookData** object will have the following properties:

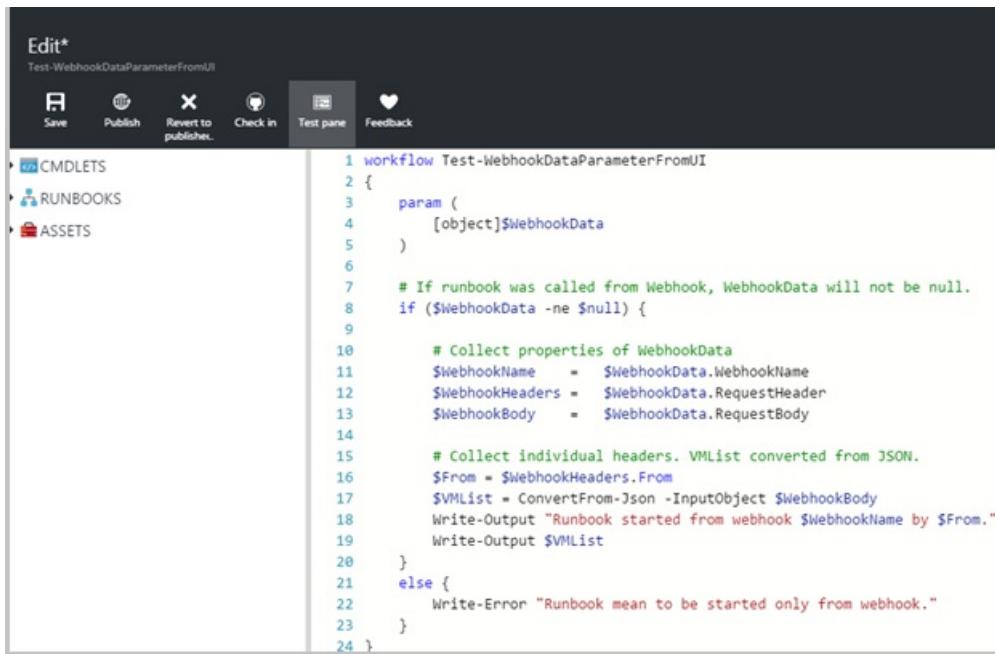
PROPERTY	DESCRIPTION
WebhookName	The name of the webhook.
RequestHeader	Hash table containing the headers of the incoming POST request.
RequestBody	The body of the incoming POST request. This will retain any formatting such as string, JSON, XML, or form encoded data. The runbook must be written to work with the data format that is expected.

There is no configuration of the webhook required to support the **\$WebhookData** parameter, and the runbook is not required to accept it. If the runbook does not define the parameter, then any details of the request sent from the client is ignored.

If you specify a value for **\$WebhookData** when you create the webhook, that value will be overridden when the

webhook starts the runbook with the data from the client POST request, even if the client does not include any data in the request body. If you start a runbook that has \$WebhookData using a method other than a webhook, you can provide a value for \$WebhookData that will be recognized by the runbook. This value should be an object with the same [properties](#) as \$WebhookData so that the runbook can properly work with it as if it was working with actual WebhookData passed by a webhook.

For example, if you are starting the following runbook from the Azure Portal and want to pass some sample WebhookData for testing, since WebhookData is an object, it should be passed as JSON in the UI.



The screenshot shows the Azure Runbook Editor interface. The title bar says "Edit* Test-WebhookDataParameterFromUI". Below the title bar are buttons for Save, Publish, Revert to publisher, Check in, Test pane (which is selected), and Feedback. On the left, there's a sidebar with "CMDLETS", "RUNBOOKS", and "ASSETS" sections. The main area contains the following PowerShell script:

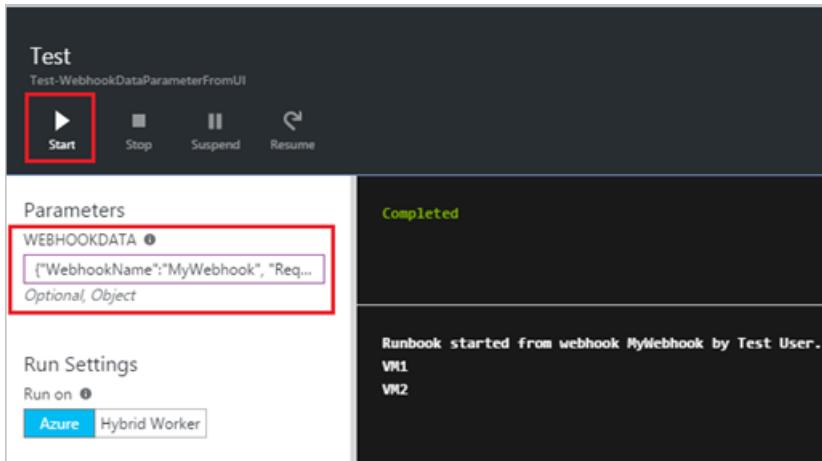
```
1 workflow Test-WebhookDataParameterFromUI
2 {
3     param (
4         [object]$WebhookData
5     )
6
7     # If runbook was called from Webhook, WebhookData will not be null.
8     if ($WebhookData -ne $null) {
9
10        # Collect properties of WebhookData
11        $WebhookName = $WebhookData.WebhookName
12        $WebhookHeaders = $WebhookData.RequestHeader
13        $WebhookBody = $WebhookData.RequestBody
14
15        # Collect individual headers. VMList converted from JSON.
16        $From = $WebhookHeaders.From
17        $VMList = ConvertFrom-Json -InputObject $WebhookBody
18        Write-Output "Runbook started from webhook $WebhookName by $From."
19        Write-Output $VMList
20    }
21    else {
22        Write-Error "Runbook mean to be started only from webhook."
23    }
24 }
```

For the above runbook, if you have the following properties for the WebhookData parameter:

1. WebhookName: *MyWebhook*
2. RequestHeader: *From=Test User*
3. RequestBody: *[“VM1”, “VM2”]*

Then you would pass the following JSON value in the UI for the WebhookData parameter:

- {"WebhookName": "MyWebhook", "RequestHeader": {"From": "Test User"}, "RequestBody": ["VM1", "VM2"]}



NOTE

The values of all input parameters are logged with the runbook job. This means that any input provided by the client in the webhook request will be logged and available to anyone with access to the automation job. For this reason, you should be cautious about including sensitive information in webhook calls.

Security

The security of a webhook relies on the privacy of its URL which contains a security token that allows it to be invoked. Azure Automation does not perform any authentication on the request as long as it is made to the correct URL. For this reason, webhooks should not be used for runbooks that perform highly sensitive functions without using an alternate means of validating the request.

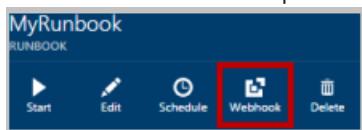
You can include logic within the runbook to determine that it was called by a webhook by checking the **WebhookName** property of the \$WebhookData parameter. The runbook could perform further validation by looking for particular information in the **RequestHeader** or **RequestBody** properties.

Another strategy is to have the runbook perform some validation of an external condition when it received a webhook request. For example, consider a runbook that is called by GitHub whenever there is a new commit to a GitHub repository. The runbook might connect to GitHub to validate that a new commit had actually just occurred before continuing.

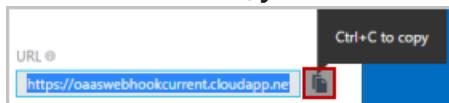
Creating a webhook

Use the following procedure to create a new webhook linked to a runbook in the Azure portal.

1. From the **Runbooks blade** in the Azure portal, click the runbook that the webhook will start to view its detail blade.
2. Click **Webhook** at the top of the blade to open the **Add Webhook** blade.



3. Click **Create new webhook** to open the **Create webhook blade**.
4. Specify a **Name**, **Expiration Date** for the webhook and whether it should be enabled. See [Details of a webhook](#) for more information about these properties.
5. Click the copy icon and press Ctrl+C to copy the URL of the webhook. Then record it in a safe place. **Once you create the webhook, you cannot retrieve the URL again.**



6. Click **Parameters** to provide values for the runbook parameters. If the runbook has mandatory parameters, then you will not be able to create the webhook unless values are provided.
7. Click **Create** to create the webhook.

Using a webhook

To use a webhook after it has been created, your client application must issue an HTTP POST with the URL for the webhook. The syntax of the webhook will be in the following format.

```
http://<Webhook Server>/token?=<Token Value>
```

The client will receive one of the following return codes from the POST request.

CODE	TEXT	DESCRIPTION
202	Accepted	The request was accepted, and the runbook was successfully queued.

Code	Text	Description
400	Bad Request	The request was not accepted for one of the following reasons. <ul style="list-style-type: none"> • The webhook has expired. • The webhook is disabled. • The token in the URL is invalid.
404	Not Found	The request was not accepted for one of the following reasons. <ul style="list-style-type: none"> • The webhook was not found. • The runbook was not found. • The account was not found.
500	Internal Server Error	The URL was valid, but an error occurred. Please resubmit the request.

Assuming the request is successful, the webhook response contains the job id in JSON format as follows. It will contain a single job id, but the JSON format allows for potential future enhancements.

```
{"JobIds":["<JobId>"]}
```

The client cannot determine when the runbook job completes or its completion status from the webhook. It can determine this information using the job id with another method such as [Windows PowerShell](#) or the [Azure Automation API](#).

Example

The following example uses Windows PowerShell to start a runbook with a webhook. Note that any language that can make an HTTP request can use a webhook; Windows PowerShell is just used here as an example.

The runbook is expecting a list of virtual machines formatted in JSON in the body of the request. We also are including information about who is starting the runbook and the date and time it is being started in the header of the request.

```
$uri = "https://s1events.azure-automation.net/webhooks?
token=8ud0d5rSo%2fvHWpYbk1w%3c8s0Gr0KJZ9Nr7zqcS%2bIqr4c%3d"
$headers = @{"From"="user@contoso.com";"Date"="05/28/2015 15:47:00"}

$vms  = @(
    @{ Name="vm01";ServiceName="vm01"},
    @{ Name="vm02";ServiceName="vm02"}
)
$body = ConvertTo-Json -InputObject $vms

$response = Invoke-RestMethod -Method Post -Uri $uri -Headers $headers -Body $body
$jobid = ConvertFrom-Json $response
```

The following image shows the header information (using a [Fiddler](#) trace) from this request. This includes standard headers of an HTTP request in addition to the custom Date and From headers that we added. Each of these values is available to the runbook in the **RequestHeaders** property of **WebhookData**.

```

Request Headers
POST /webhooks?token=8ud0dSrSo%2vHWpYbklW%3c8s0GrOKJZ9Nr7zqcS%2bIQr4c%3d HTTP/1.1
Client
User-Agent: Mozilla/5.0 (Windows NT; Windows NT 6.3; en-US) WindowsPowerShell/5.0.10018.0
Entity
Content-Length: 200
Content-Type: application/x-www-form-urlencoded
Miscellaneous
Date: Thu, 28 May 2015 22:47:00 GMT
From: user@contoso.com
Transport
Connection: Keep-Alive
Host: oaswebhookcurrent.cloudapp.net

```

The following image shows the body of the request (using a [Fiddler](#) trace) that is available to the runbook in the **RequestBody** property of **WebhookData**. This is formatted as JSON because that was the format that was included in the body of the request.

```
{
  "Name": "vm01",
  "ServiceName": "vm01"
},
{
  "Name": "vm02",
  "ServiceName": "vm02"
}
```

The following image shows the request being sent from Windows PowerShell and the resulting response. The job id is extracted from the response and converted to a string.

```

PS C:\> $response = Invoke-WebRequest -Method Post -Uri $uri -Headers $headers -Body $body
PS C:\> $response

StatusCode      : 202
StatusDescription : Accepted
Content         : {"JobIds":["53d64dcd-43c5-4a03-9345-4d6e24a9512f"]}
RawContent      : HTTP/1.1 202 Accepted
                  Pragma: no-cache
                  Strict-Transport-Security: max-age=31536000; includeSubDomains
                  Content-Length: 0
                  Cache-Control: no-cache
                  Date: Sat, 30 May 2015 00:04:14 GMT
                  Expires: -1
                  S...
Headers          : {[Pragma, no-cache], [strict-Transport-Security, max-age=31536000; includeSubDomains],
                   [Content-Length, 0], [Cache-Control, no-cache]...}
RawContentLength : 0

PS C:\> $jobid = (ConvertFrom-Json ($response.Content)).JobIds[0]
PS C:\> $jobid
53d64dcd-43c5-4a03-9345-4d6e24a9512f
PS C:\>
```

The following sample runbook accepts the previous example request and starts the virtual machines specified in the request body.

```

workflow Test-StartVirtualMachinesFromWebhook
{
    param (
        [object]$WebhookData
    )

    # If runbook was called from Webhook, WebhookData will not be null.
    if ($WebhookData -ne $null) {

        # Collect properties of WebhookData
        $WebhookName      =      $WebhookData.WebhookName
        $WebhookHeaders   =      $WebhookData.RequestHeader
        $WebhookBody      =      $WebhookData.RequestBody

        # Collect individual headers. VMList converted from JSON.
        $From = $WebhookHeaders.From
        $VMList = ConvertFrom-Json -InputObject $WebhookBody
        Write-Output "Runbook started from webhook $WebhookName by $From."

        # Authenticate to Azure resources
        $Cred = Get-AutomationPSCredential -Name 'MyAzureCredential'
        Add-AzureAccount -Credential $Cred

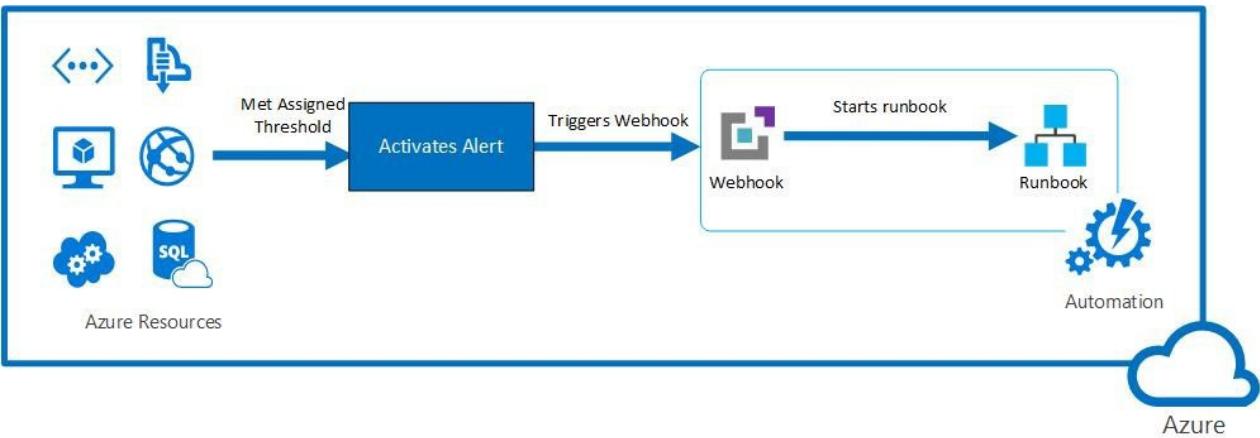
        # Start each virtual machine
        foreach ($VM in $VMList)
        {
            $VMName = $VM.Name
            Write-Output "Starting $VMName"
            Start-AzureVM -Name $VM.Name -ServiceName $VM.ServiceName
        }
    }
    else {
        Write-Error "Runbook mean to be started only from webhook."
    }
}

```

Starting runbooks in response to Azure alerts

Webhook-enabled runbooks can be used to react to [Azure alerts](#). Resources in Azure can be monitored by collecting the statistics like performance, availability and usage with the help of Azure alerts. You can receive an alert based on monitoring metrics or events for your Azure resources, currently Automation Accounts support only metrics. When the value of a specified metric exceeds the threshold assigned or if the configured event is triggered then a notification is sent to the service admin or co-admins to resolve the alert, for more information on metrics and events please refer to [Azure alerts](#).

Besides using Azure alerts as a notification system, you can also kick off runbooks in response to alerts. Azure Automation provides the capability to run webhook-enabled runbooks with Azure alerts. When a metric exceeds the configured threshold value then the alert rule becomes active and triggers the automation webhook which in turn executes the runbook.



Alert context

Consider an Azure resource such as a virtual machine, CPU utilization of this machine is one of the key performance metric. If the CPU utilization is 100% or more than a certain amount for long period of time, you might want to restart the virtual machine to fix the problem. This can be solved by configuring an alert rule to the virtual machine and this rule takes CPU percentage as its metric. CPU percentage here is just taken as an example but there are many other metrics that you can configure to your Azure resources and restarting the virtual machine is an action that is taken to fix this issue, you can configure the runbook to take other actions.

When this the alert rule becomes active and triggers the webhook-enabled runbook, it sends the alert context to the runbook. **Alert context** contains details including **SubscriptionID**, **ResourceGroupName**, **ResourceName**, **ResourceType**, **ResourceId** and **Timestamp** which are required for the runbook to identify the resource on which it will be taking action. Alert context is embedded in the body part of the **WebhookData** object sent to the runbook and it can be accessed with **Webhook.RequestBody** property

Example

Create an Azure virtual machine in your subscription and associate an [alert to monitor CPU percentage metric](#). While creating the alert make sure you populate the webhook field with the URL of the webhook which was generated while creating the webhook.

The following sample runbook is triggered when the alert rule becomes active and it collects the Alert context parameters which are required for the runbook to identify the resource on which it will be taking action.

```

workflow Invoke-RunbookUsingAlerts
{
    param (
        [object]$WebhookData
    )

    # If runbook was called from Webhook, WebhookData will not be null.
    if ($WebhookData -ne $null) {
        # Collect properties of WebhookData.
        $WebhookName = $WebhookData.WebhookName
        $WebhookBody = $WebhookData.RequestBody
        $WebhookHeaders = $WebhookData.RequestHeader

        # Outputs information on the webhook name that called This
        Write-Output "This runbook was started from webhook $WebhookName."

        # Obtain the WebhookBody containing the AlertContext
        $WebhookBody = (ConvertFrom-Json -InputObject $WebhookBody)
        Write-Output "`nWEBHOOK BODY"
        Write-Output "====="
        Write-Output $WebhookBody

        # Obtain the AlertContext
        $AlertContext = [object]$WebhookBody.context

        # Some selected AlertContext information
        Write-Output "`nALERT CONTEXT DATA"
        Write-Output "====="
        Write-Output $AlertContext.name
        Write-Output $AlertContext.subscriptionId
        Write-Output $AlertContext.resourceGroupName
        Write-Output $AlertContext.resourceName
        Write-Output $AlertContext.resourceType
        Write-Output $AlertContext.resourceId
        Write-Output $AlertContext.timestamp

        # Act on the AlertContext data, in our case restarting the VM.
        # Authenticate to your Azure subscription using Organization ID to be able to restart that Virtual
Machine.
        $cred = Get-AutomationPSCredential -Name "MyAzureCredential"
        Add-AzureAccount -Credential $cred
        Select-AzureSubscription -subscriptionName "Visual Studio Ultimate with MSDN"

        #Check the status property of the VM
        Write-Output "Status of VM before taking action"
        Get-AzureVM -Name $AlertContext.resourceName -ServiceName $AlertContext.resourceName
        Write-Output "Restarting VM"

        # Restart the VM by passing VM name and Service name which are same in this case
        Restart-AzureVM -ServiceName $AlertContext.resourceName -Name $AlertContext.resourceName
        Write-Output "Status of VM after alert is active and takes action"
        Get-AzureVM -Name $AlertContext.resourceName -ServiceName $AlertContext.resourceName
    }
    else
    {
        Write-Error "This runbook is meant to only be started from a webhook."
    }
}

```

Next steps

- For details on different ways to start a runbook, see [Starting a Runbook](#).
- For information on viewing the Status of a Runbook Job, refer to [Runbook execution in Azure Automation](#).

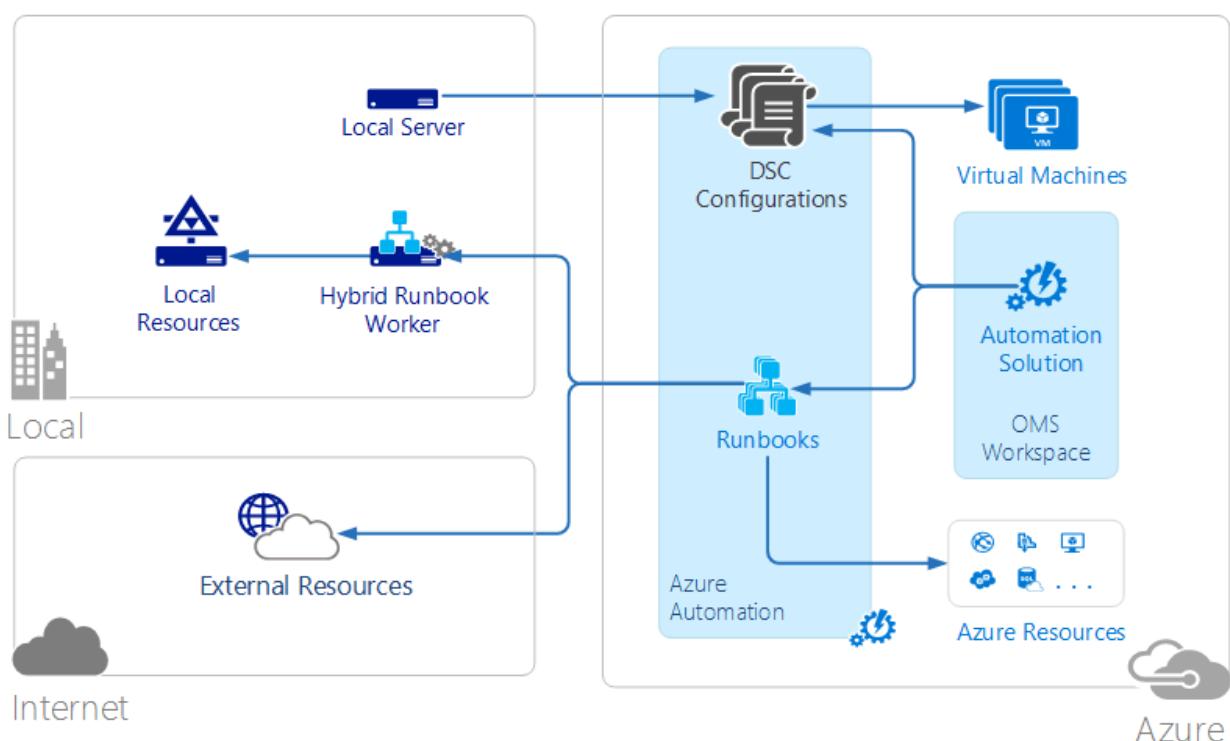
- To learn how to use Azure Automation to take action on Azure Alerts, see [Remediate Azure VM Alerts with Automation Runbooks](#).
- To learn how to invoke a runbook from an OMS Log Analytics alert, see [Runbook Actions with Log Analytics Alerts](#).

Automate resources in your data center with Hybrid Runbook Worker

2/28/2017 • 16 min to read • [Edit on GitHub](#)

Runbooks in Azure Automation cannot access resources in your local data center since they run in the Azure cloud. The Hybrid Runbook Worker feature of Azure Automation allows you to run runbooks on machines located in your data center in order to manage local resources. The runbooks are stored and managed in Azure Automation and then delivered to one or more on-premises machines.

This functionality is illustrated in the following image.



You can designate one or more computers in your data center to act as a Hybrid Runbook Worker and run runbooks from Azure Automation. Each worker requires the Microsoft Management Agent with a connection to Microsoft Operations Management Suite and the Azure Automation runbook environment. Operations Management Suite is only used to install and maintain the management agent and to monitor the functionality of the worker. The delivery of runbooks and the instruction to run them are performed by Azure Automation.

There are no inbound firewall requirements to support Hybrid Runbook Workers. The agent on the local computer initiates all communication with Azure Automation in the cloud. When a runbook is started, Azure Automation creates an instruction that is retrieved by agent. The agent then pulls down the runbook and any parameters before running it. It will also retrieve any **assets** that are used by the runbook from Azure Automation.

NOTE

In order to manage the configuration of your servers supporting the Hybrid Runbook Worker role with Desired State Configuration (DSC), you will need to add them as DSC nodes. For further information about onboarding them for management with DSC, see [Onboarding machines for management by Azure Automation DSC](#).

If you enable the [Update Management solution](#), any Windows computer connected to your OMS workspace will automatically be configured as a Hybrid Runbook Worker in order to support runbooks that are included in this solution. However, it is not registered with any Hybrid Worker groups you may already have defined in your Automation account. The computer can be added to a Hybrid Runbook Worker group in your Automation account to support Automation runbooks as long as you are using the same account for both the solution and Hybrid Runbook Worker group membership. This functionality has been added to version 7.2.12024.0 of the Hybrid Runbook Worker.

Hybrid Runbook Worker groups

Each Hybrid Runbook Worker is a member of a Hybrid Runbook Worker group that you specify when you install the agent. A group can include a single agent, but you can install multiple agents in a group for high availability.

When you start a runbook on a Hybrid Runbook Worker, you specify the group that it will run on. The members of the group will determine which worker will service the request. You cannot specify a particular worker.

Hybrid Runbook Worker requirements

You must designate at least one on-premises computer to run hybrid runbook jobs. This computer must have the following:

- Windows Server 2012 or later
- Windows PowerShell 4.0 or later. We recommend installing Windows PowerShell 5.0 on the computer(s) for increased reliability. You can download the new version from the [Microsoft Download Center](#)
- Minimum of two cores and 4 GB of RAM

Consider the following recommendations for hybrid workers:

- Designate multiple hybrid workers in each group for high availability.
- Hybrid workers can coexist with Service Management Automation or System Center Orchestrator runbook servers.
- Consider using a computer physically located in or near the region of your Automation account since the job data is sent back to Azure Automation when a job completes.

Configure proxy and firewall settings

For the on-premise Hybrid Runbook Worker to connect to and register with the Microsoft Operations Management Suite (OMS) service, it must have access to the port number and the URLs described below. This is in addition to the [ports and URLs required for the Microsoft Monitoring Agent](#) to connect to OMS. If you use a proxy server for communication between the agent and the OMS service, you'll need to ensure that the appropriate resources are accessible. If you use a firewall to restrict access to the Internet, you need to configure your firewall to permit access.

The information below list the port and URLs that are required for the Hybrid Runbook Worker to communicate with Automation.

- Port: Only TCP 443 is required for outbound Internet access
- Global URL: *.azure-automation.net

If you have an Automation account defined for a specific region and you want to restrict communication with that regional datacenter, the following table provides the DNS record for each region.

REGION	DNS RECORD
South Central US	scus-jobruntimedata-prod-su1.azure-automation.net
East US 2	eus2-jobruntimedata-prod-su1.azure-automation.net
West Europe	we-jobruntimedata-prod-su1.azure-automation.net
North Europe	ne-jobruntimedata-prod-su1.azure-automation.net
Canada Central	cc-jobruntimedata-prod-su1.azure-automation.net
South East Asia	sea-jobruntimedata-prod-su1.azure-automation.net
Central India	cid-jobruntimedata-prod-su1.azure-automation.net
Japan East	jpe-jobruntimedata-prod-su1.azure-automation.net
Australia South East	ase-jobruntimedata-prod-su1.azure-automation.net

Installing Hybrid Runbook Worker

There are two methods described below on how to install and configure Hybrid Runbook Worker. The first method is a PowerShell script which automates all steps required to configure the Windows computer, and this is the suggested approach as it streamlines the entire deployment process for you. The second method is following a step-by-step procedure to manually install and configure the role.

Automated deployment

Perform the following steps to automate the installation and configuration of the Hybrid Worker role.

1. Download the *New-OnPremiseHybridWorker.ps1* script from the [PowerShell Gallery](#) directly from the computer running the Hybrid Runbook Worker role or from another computer in your environment and copy it to the worker.

The *New-OnPremiseHybridWorker.ps1* script requires the following parameters during execution:

- *AutomationAccountName* (mandatory) - the name of your Automation account.
- *ResourceGroupName* (mandatory) - the name of the resource group associated with your Automation account.
- *HybridGroupName* (mandatory) - the name of a Hybrid Runbook Worker group that you will specify as a target for the runbooks supporting this scenario
- *SubscriptionID* (mandatory) - the Azure Subscription Id that your Automation account is in
- *WorkspaceName* (optional) - the OMS workspace name. If you do not have an OMS workspace, the script will create and configure one.

NOTE

Currently the only Automation regions supported for integration with OMS are - **Australia Southeast, East US 2, Southeast Asia, and West Europe**. If your Automation account is not in one of those regions, the script will create the OMS workspace but it will warn you that it cannot link them together.

2. On your computer, start **Windows PowerShell** from the **Start** screen in Administrator mode.

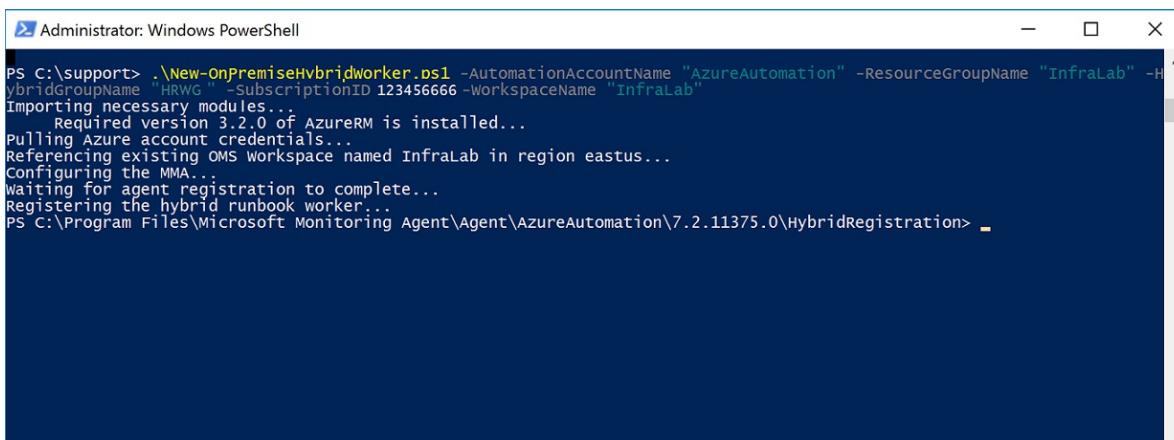
3. From the PowerShell command-line shell, navigate to the folder which contains the script you downloaded and execute it changing the values for parameters `-AutomationAccountName`, `-ResourceGroupName`, `-HybridGroupName`, `-SubscriptionId`, and `-WorkspaceName`.

NOTE

You will be prompted to authenticate with Azure after you execute the script. You **must** log in with an account that is a member of the Subscription Admins role and co-administrator of the subscription.

```
.\New-OnPremiseHybridWorker.ps1 -AutomationAccountName <NameofAutomationAccount> `  
-ResourceGroupName <NameofOResourceGroup> -HybridGroupName <NameofHRWGroup> `  
-SubscriptionId <AzureSubscriptionId> -WorkspaceName <NameofOMSWorkspace>
```

4. You will be prompted to agree to install **NuGet** and you will be prompted to authenticate with your Azure credentials.



```
Administrator: Windows PowerShell  
PS C:\support> .\New-OnPremiseHybridWorker.ps1 -AutomationAccountName "AzureAutomation" -ResourceGroupName "InfraLab" -HybridGroupName "HRWG" -SubscriptionID 123456666 -WorkspaceName "InfraLab"  
Importing necessary modules...  
Required version 3.2.0 of AzureRM is installed...  
Pulling Azure account credentials...  
Referencing existing OMS Workspace named InfraLab in region eastus...  
Configuring the MMA...  
Waiting for agent registration to complete...  
Registering the hybrid runbook worker...  
PS C:\Program Files\Microsoft Monitoring Agent\Agent\AzureAutomation\7.2.11375.0\HybridRegistration>
```

5. After the script is complete, the Hybrid Worker Groups blade will show the new group and number of members or if an existing group, the number of members will be incremented accordingly. You can select the group from the list on the on the **Hybrid Worker Groups** blade and select the **Hybrid Workers** tile. On the **Hybrid Workers** blade, you will see each member of the group listed.

Manual deployment

Perform the first two steps once for your Automation environment and then repeat the remaining steps for each worker computer.

1. Create Operations Management Suite workspace

If you do not already have an Operations Management Suite workspace, then create one using instructions at [Set up your workspace](#). You can use an existing workspace if you already have one.

2. Add Automation solution to Operations Management Suite workspace

Solutions add functionality to Operations Management Suite. The Automation solution adds functionality for Azure Automation including support for Hybrid Runbook Worker. When you add the solution to your workspace, it will automatically push down worker components to the agent computer that you will install in the next step.

Follow the instructions at [To add a solution using the Solutions Gallery](#) to add the **Automation** solution to your Operations Management Suite workspace.

3. Install the Microsoft Monitoring Agent

The Microsoft Monitoring Agent connects computers to Operations Management Suite. When you install the agent on your on-premises computer and connect it to your workspace, it will automatically download the components required for Hybrid Runbook Worker.

Follow the instructions at [Connect Windows computers to Log Analytics](#) to install the agent on the on-premises

computer. You can repeat this process for multiple computers to add multiple workers to your environment.

When the agent has successfully connected to Operations Management Suite, it will be listed on the **Connected Sources** tab of the Operations Management Suite **Settings** pane. You can verify that the agent has correctly downloaded the Automation solution when it has a folder called **AzureAutomationFiles** in C:\Program Files\Microsoft Monitoring Agent\Agent. To confirm the version of the Hybrid Runbook Worker, you can navigate to C:\Program Files\Microsoft Monitoring Agent\Agent\AzureAutomation\ and note the \version sub-folder.

4. Install the runbook environment and connect to Azure Automation

When you add an agent to Operations Management Suite, the Automation solution pushes down the **HybridRegistration** PowerShell module which contains the **Add-HybridRunbookWorker** cmdlet. You use this cmdlet to install the runbook environment on the computer and register it with Azure Automation.

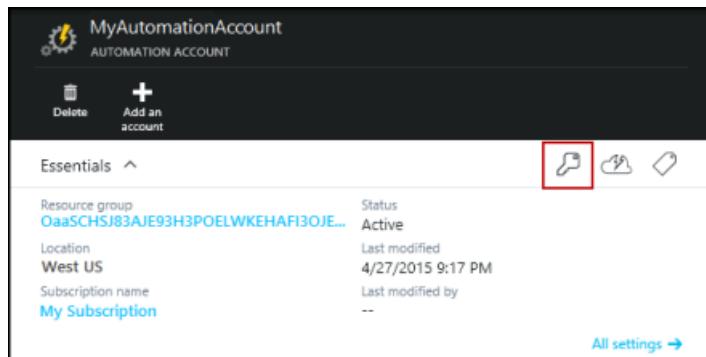
Open a PowerShell session in Administrator mode and run the following commands to import the module.

```
cd "C:\Program Files\Microsoft Monitoring Agent\Agent\AzureAutomation\<version>\HybridRegistration"
Import-Module HybridRegistration.psd1
```

Then run the **Add-HybridRunbookWorker** cmdlet using the following syntax:

```
Add-HybridRunbookWorker -Name <String> -EndPoint <Url> -Token <String>
```

You can get the information required for this cmdlet from the **Manage Keys** blade in the Azure portal. Open this blade by selecting the **Keys** option from the **Settings** blade from your Automation account.



- **Name** is the name of the Hybrid Runbook Worker Group. If this group already exists in the automation account, then the current computer is added to it. If it does not already exist, then it is added.
- **EndPoint** is the **URL** field in the **Manage Keys** blade.
- **Token** is the **Primary Access Key** in the **Manage Keys** blade.

Use the **-Verbose** switch with **Add-HybridRunbookWorker** to receive detailed information about the installation.

5. Install PowerShell modules

Runbooks can use any of the activities and cmdlets defined in the modules installed in your Azure Automation environment. These modules are not automatically deployed to on-premises computers though, so you must install them manually. The exception is the Azure module which is installed by default providing access to cmdlets for all Azure services and activities for Azure Automation.

Since the primary purpose of the Hybrid Runbook Worker feature is to manage local resources, you will most likely need to install the modules that support these resources. You can refer to [Installing Modules](#) for information on installing Windows PowerShell modules.

Removing Hybrid Runbook Worker

You can remove one or more Hybrid Runbook Workers from a group or you can remove the group, depending on your requirements. To remove a Hybrid Runbook Worker from an on-premises computer, perform the following steps.

1. In the Azure portal navigate to your Automation account.
2. From the **Settings** blade, select **Keys** and note the values for field **URL** and **Primary Access Key**. You will need this information for the next step.
3. Open a PowerShell session in Administrator mode and run the following command -
`Remove-HybridRunbookWorker -url <URL> -key <PrimaryAccessKey>`. Use the **-Verbose** switch for a detailed log of the removal process.

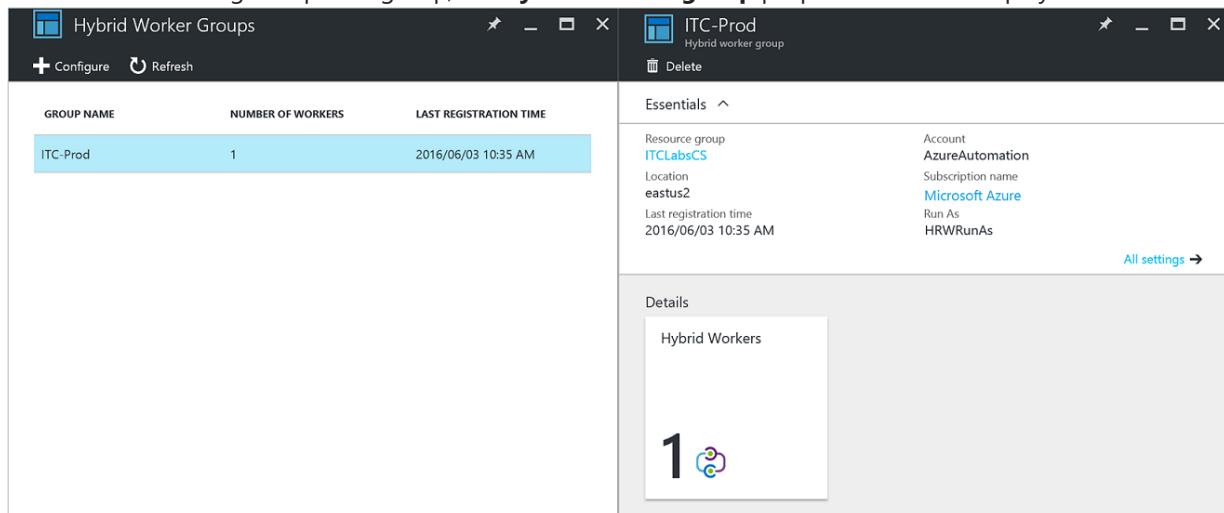
NOTE

This does not remove the Microsoft Monitoring Agent from the computer, only the functionality and configuration of the Hybrid Runbook Worker role.

Remove Hybrid Worker groups

To remove a group, you first need to remove the Hybrid Runbook Worker from every computer that is a member of the group using the procedure shown earlier, and then you perform the following steps to remove the group.

1. Open the Automation account in the Azure portal.
2. Select the **Hybrid Worker Groups** tile and in the **Hybrid Worker Groups** blade, select the group you wish to delete. After selecting the specific group, the **Hybrid worker group** properties blade is displayed.



3. On the properties blade for the selected group, click **Delete**. A message will appear asking you to confirm this action, and select **Yes** if you are sure you want to proceed.

Delete Hybrid Worker Group

The hybrid worker group ITC-Prod will be deleted and you will not be able to run jobs on any of the workers in this group. Before you delete the worker group, unregister any servers you have defined as hybrid runbook workers by running the Remove-HybridRunbookWorker cmdlet. If you no longer have access to the runbook workers, deleting the hybrid worker group will remove the association between the workers in this group and your account. Are you sure you want to continue?

Yes

No

This process can take several seconds to complete and you can track its progress under **Notifications** from

the menu.

Starting runbooks on Hybrid Runbook Worker

[Starting a Runbook in Azure Automation](#) describes different methods for starting a runbook. Hybrid Runbook Worker adds a **RunOn** option where you can specify the name of a Hybrid Runbook Worker Group. If a group is specified, then the runbook is retrieved and run by of the workers in that group. If this option is not specified, then it is run in Azure Automation as normal.

When you start a runbook in the Azure portal, you will be presented with a **Run on** option where you can select **Azure** or **Hybrid Worker**. If you select **Hybrid Worker**, then you can select the group from a dropdown.

Use the **RunOn** parameter You could use the following command to start a runbook named Test-Runbook on a Hybrid Runbook Worker Group named MyHybridGroup using Windows PowerShell.

```
Start-AzureRmAutomationRunbook -AutomationAccountName "MyAutomationAccount" -Name "Test-Runbook" -RunOn  
"MyHybridGroup"
```

NOTE

The **RunOn** parameter was added to the **Start-AzureAutomationRunbook** cmdlet in version 0.9.1 of Microsoft Azure PowerShell. You should [download the latest version](#) if you have an earlier one installed. You only need to install this version on a workstation where you will be starting the runbook from Windows PowerShell. You do not need to install it on the worker computer unless you intend to start runbooks from that computer. You cannot currently start a runbook on a Hybrid Runbook Worker from another runbook since this would require the latest version of Azure Powershell to be installed in your Automation account. The latest version will be automatically updated in Azure Automation and automatically pushed down to the workers soon.

Runbook permissions

Runbooks running on a Hybrid Runbook Worker cannot use the same method that is typically used for runbooks authenticating to Azure resources, since they will be accessing resources outside of Azure. The runbook can either provide its own authentication to local resources, or you can specify a RunAs account to provide a user context for all runbooks.

Runbook authentication

By default, runbooks will run in the context of the local System account on the on-premises computer, so they must provide their own authentication to resources that they will access.

You can use [Credential](#) and [Certificate](#) assets in your runbook with cmdlets that allow you to specify credentials so you can authenticate to different resources. The following example shows a portion of a runbook that restarts a computer. It retrieves credentials from a credential asset and the name of the computer from a variable asset and then uses these values with the Restart-Computer cmdlet.

```
$Cred = Get-AzureRmAutomationCredential -ResourceGroupName "ResourceGroup01" -Name "MyCredential"  
$Computer = Get-AzureRmAutomationVariable -ResourceGroupName "ResourceGroup01" -Name "ComputerName"  
  
Restart-Computer -ComputerName $Computer -Credential $Cred
```

You can also leverage [InlineScript](#) which will allow you to run blocks of code on another computer with credentials specified by the [PSCredential common parameter](#).

RunAs account

Instead of having runbooks provide their own authentication to local resources, you can specify a **RunAs** account

for a Hybrid worker group. You specify a [credential asset](#) that has access to local resources, and all runbooks will run under these credentials when running on a Hybrid Runbook Worker in the group.

The user name for the credential must be in one of the following formats:

- domain\username
- username@domain
- username (for accounts local to the on-premises computer)

Use the following procedure to specify a RunAs account for a Hybrid worker group:

1. Create a [credential asset](#) with access to local resources.
2. Open the Automation account in the Azure portal.
3. Select the **Hybrid Worker Groups** tile, and then select the group.
4. Select **All settings** and then **Hybrid worker group settings**.
5. Change **Run As** from **Default** to **Custom**.
6. Select the credential and click **Save**.

Creating runbooks for Hybrid Runbook Worker

There is no difference in the structure of runbooks that run in Azure Automation and those that run on a Hybrid Runbook Worker. Runbooks that you use with each will most likely differ significantly though since runbooks for Hybrid Runbook Worker will typically manage local resources in your data center while runbooks in Azure Automation typically manage resources in the Azure cloud.

You can edit a runbook for Hybrid Runbook Worker in Azure Automation, but you may have difficulties if you try to test the runbook in the editor. The PowerShell modules that access the local resources may not be installed in your Azure Automation environment in which case, the test would fail. If you do install the required modules, then the runbook will run, but it will not be able to access local resources for a complete test.

Troubleshooting runbooks on Hybrid Runbook Worker

[Runbook output and messages](#) are sent to Azure Automation from hybrid workers just like runbook jobs run in the cloud. You can also enable the Verbose and Progress streams the same way you would for other runbooks.

Logs are stored locally on each hybrid worker at C:\ProgramData\Microsoft\System Center\Orchestrator\7.2\SMA\Sandboxes.

If your runbooks are not completing successfully and the job summary shows a status of **Suspended**, please review the troubleshooting article [Hybrid Runbook Worker: A runbook job terminates with a status of Suspended](#).

Relationship to Service Management Automation

[Service Management Automation \(SMA\)](#) allows you to run the same runbooks that are supported by Azure Automation in your local data center. SMA is generally deployed together with Windows Azure Pack, as Windows Azure Pack contains a graphical interface for SMA management. Unlike Azure Automation, SMA requires a local installation that includes web servers to host the API, a database to contain runbooks and SMA configuration, and Runbook Workers to execute runbook jobs. Azure Automation provides these services in the cloud and only requires you to maintain the Hybrid Runbook Workers in your local environment.

If you are an existing SMA user, you can move your runbooks to Azure Automation to be used with Hybrid Runbook Worker with no changes, assuming that they perform their own authentication to resources as described in [Creating runbooks for Hybrid Runbook Worker](#). Runbooks in SMA run in the context of the service account on the worker server which may provide that authentication for the runbooks.

You can use the following criteria to determine whether Azure Automation with Hybrid Runbook Worker or

Service Management Automation is more appropriate for your requirements.

- SMA requires a local installation of its underlying components that are connected to Windows Azure Pack if a graphical management interface is required. More local resources will be needed with higher maintenance costs than Azure Automation, which only needs an agent installed on local runbook workers. The agents are managed by Operations Management Suite, further decreasing your maintenance costs.
- Azure Automation stores its runbooks in the cloud and delivers them to on-premises Hybrid Runbook Workers. If your security policy does not allow this behavior, then you should use SMA.
- SMA is included with System Center; and therefore, requires a System Center 2012 R2 license. Azure Automation is based on a tiered subscription model.
- Azure Automation has advanced features such as graphical runbooks that are not available in SMA.

Next steps

- To learn more about the different methods that can be used to start a runbook, see [Starting a Runbook in Azure Automation](#).
- To understand the different procedures for working with PowerShell and PowerShell Workflow runbooks in Azure Automation using the textual editor, see [Editing a Runbook in Azure Automation](#)

Runbook input parameters

1/17/2017 • 10 min to read • [Edit on GitHub](#)

Runbook input parameters increase the flexibility of runbooks by allowing you to pass data to it when it is started. The parameters allow the runbook actions to be targeted for specific scenarios and environments. In this article, we will walk you through different scenarios where input parameters are used in runbooks.

Configure input parameters

Input parameters can be configured in PowerShell, PowerShell Workflow, and graphical runbooks. A runbook can have multiple parameters with different data types, or no parameters at all. Input parameters can be mandatory or optional, and you can assign a default value for optional parameters. You can assign values to the input parameters for a runbook when you start it through one of the available methods. These methods include starting a runbook from the portal or a web service. You can also start one as a child runbook that is called inline in another runbook.

Configure input parameters in PowerShell and PowerShell Workflow runbooks

PowerShell and [PowerShell Workflow runbooks](#) in Azure Automation support input parameters that are defined through the following attributes.

PROPERTY	DESCRIPTION
Type	Required. The data type expected for the parameter value. Any .NET type is valid.
Name	Required. The name of the parameter. This must be unique within the runbook, and can contain only letters, numbers, or underscore characters. It must start with a letter.
Mandatory	Optional. Specifies whether a value must be provided for the parameter. If you set this to \$true , then a value must be provided when the runbook is started. If you set this to \$false , then a value is optional.
Default value	Optional. Specifies a value that will be used for the parameter if a value is not passed in when the runbook is started. A default value can be set for any parameter and will automatically make the parameter optional regardless of the Mandatory setting.

Windows PowerShell supports more attributes of input parameters than those listed here, like validation, aliases, and parameter sets. However, Azure Automation currently supports only the input parameters listed above.

A parameter definition in PowerShell Workflow runbooks has the following general form, where multiple parameters are separated by commas.

```

Param
(
    [Parameter (Mandatory= $true/$false)]
    [Type] Name1 = <Default value>,

    [Parameter (Mandatory= $true/$false)]
    [Type] Name2 = <Default value>
)

```

NOTE

When you're defining parameters, if you don't specify the **Mandatory** attribute, then by default, the parameter is considered optional. Also, if you set a default value for a parameter in PowerShell Workflow runbooks, it will be treated by PowerShell as an optional parameter, regardless of the **Mandatory** attribute value.

As an example, let's configure the input parameters for a PowerShell Workflow runbook that outputs details about virtual machines, either a single VM or all VMs within a resource group. This runbook has two parameters as shown in the following screenshot: the name of virtual machine and the name of the resource group.

```

Edit PowerShell Workflow Runbook
ListAllVMs-PSW
Save Publish Revert to public... Check in Test pane Feedback
CMDLETS RUNBOOKS ASSETS
1 workflow ListAllVMs-PSW
2 {
3     Param(
4         [Parameter(Mandatory = $false)]
5         [String] $VMName,
6         [Parameter(Mandatory = $false)]
7         [String] $ResourceGroupName = "InfraLab"
8     )
9     $connectionName = "AzureRunAsConnection"
10    $subId = Get-AutomationVariable -Name 'SubscriptionId'
11    try
12    {
13        # Get the connection "AzureRunAsConnection"
14        $servicePrincipalConnection = Get-AutomationConnection -Name $connectionName
15
16        "Logging in to Azure..."
17        Add-AzureRmAccount `
18            -ServicePrincipal `
19            -TenantId $servicePrincipalConnection.TenantId `
20            -ApplicationId $servicePrincipalConnection.ApplicationId `
21            -CertificateThumbprint $servicePrincipalConnection.CertificateThumbprint
22        "Setting context to a specific subscription"
23        Set-AzureRmContext -SubscriptionId $subId
24    }
25    catch {
26        if (!$servicePrincipalConnection)
27        {
28            $errorMessage = "Connection $connectionName not found."
29            throw $errorMessage
30        }
31        else{
32            Write-Error -Message $_.Exception
33            throw $_.Exception
34        }
35    }
36    IF ($VMName)
37    {
38        Get-AzureRmVM -Name $VMName -ResourceGroup $resourceGroupName
39    }
39    else
40    {
41        Get-AzureRmVM -ResourceGroup $resourceGroupName
42    }
43 }

```

In this parameter definition, the parameters **\$VMName** and **\$resourceGroupName** are simple parameters of type string. However, PowerShell and PowerShell Workflow runbooks support all simple types and complex types, such as **object** or **PSCredential** for input parameters.

If your runbook has an object type input parameter, then use a PowerShell hashtable with (name,value) pairs to pass in a value. For example, if you have the following parameter in a runbook:

```

[Parameter (Mandatory = $true)]
[object] $FullName

```

Then you can pass the following value to the parameter:

```
@{"FirstName"="Joe";"MiddleName"="Bob";"LastName"="Smith"}
```

Configure input parameters in graphical runbooks

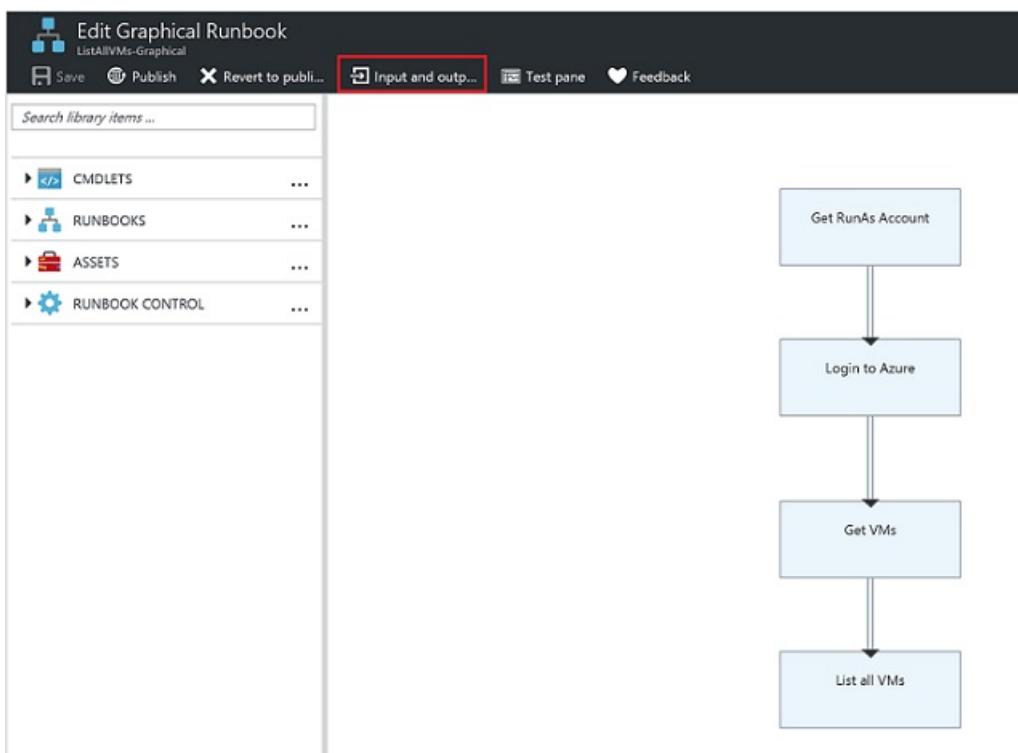
To [configure a graphical runbook](#) with input parameters, let's create a graphical runbook that outputs details about virtual machines, either a single VM or all VMs within a resource group. Configuring a runbook consists of two major activities, as described below.

Authenticate Runbooks with Azure Run As account to authenticate with Azure.

Get-AzureRmVm to get the properties of a virtual machines.

You can use the **Write-Output** activity to output the names of virtual machines. The activity **Get-AzureRmVm** accepts two parameters, the **virtual machine name** and the **resource group name**. Since these parameters could require different values each time you start the runbook, you can add input parameters to your runbook. Here are the steps to add input parameters:

1. Select the graphical runbook from the **Runbooks** blade and then click **Edit** it.
2. From the runbook editor, click **Input and output** to open the **Input and output** blade.



3. The **Input and output** blade displays a list of input parameters that are defined for the runbook. On this blade, you can either add a new input parameter or edit the configuration of an existing input parameter. To add a new parameter for the runbook, click **Add input** to open the **Runbook input parameter** blade. There, you can configure the following parameters:

PROPERTY	DESCRIPTION
Name	Required. The name of the parameter. This must be unique within the runbook, and can contain only letters, numbers, or underscore characters. It must start with a letter.

PROPERTY	DESCRIPTION
Description	Optional. Description about the purpose of input parameter.
Type	Optional. The data type that's expected for the parameter value. Supported parameter types are String , Int32 , Int64 , Decimal , Boolean , DateTime , and Object . If a data type is not selected, it defaults to String .
Mandatory	Optional. Specifies whether a value must be provided for the parameter. If you choose yes , then a value must be provided when the runbook is started. If you choose no , then a value is not required when the runbook is started, and a default value may be set.
Default Value	Optional. Specifies a value that will be used for the parameter if a value is not passed in when the runbook is started. A default value can be set for a parameter that's not mandatory. To set a default value, choose Custom . This value is used unless another value is provided when the runbook is started. Choose None if you don't want to provide any default value.

Runbook Input Param...

* Name ⓘ
resourceGroupName ✓

Description ⓘ

Type ⓘ
String

Mandatory ⓘ
Yes No

Default value ⓘ
None Custom

Custom default value
WSSC1

4. Create two parameters with the following properties that will be used by the **Get-AzureRmVm** activity:

- **Parameter1:**

- Name - VMName
- Type - String
- Mandatory - No

- **Parameter2:**

- Name - resourceGroupName
- Type - String
- Mandatory - No
- Default value - Custom
- Custom default value - <Name of the resource group that contains the virtual machines>

5. Once you add the parameters, click **OK**. You can now view them in the **Input and output blade**. Click **OK** again, and then click **Save** and **Publish** your runbook.

Assign values to input parameters in runbooks

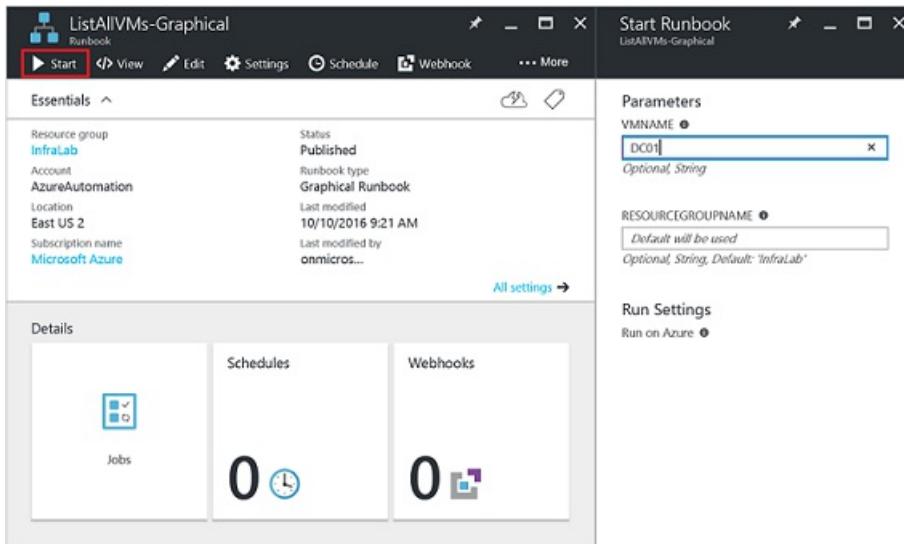
You can pass values to input parameters in runbooks in the following scenarios.

Start a runbook and assign parameters

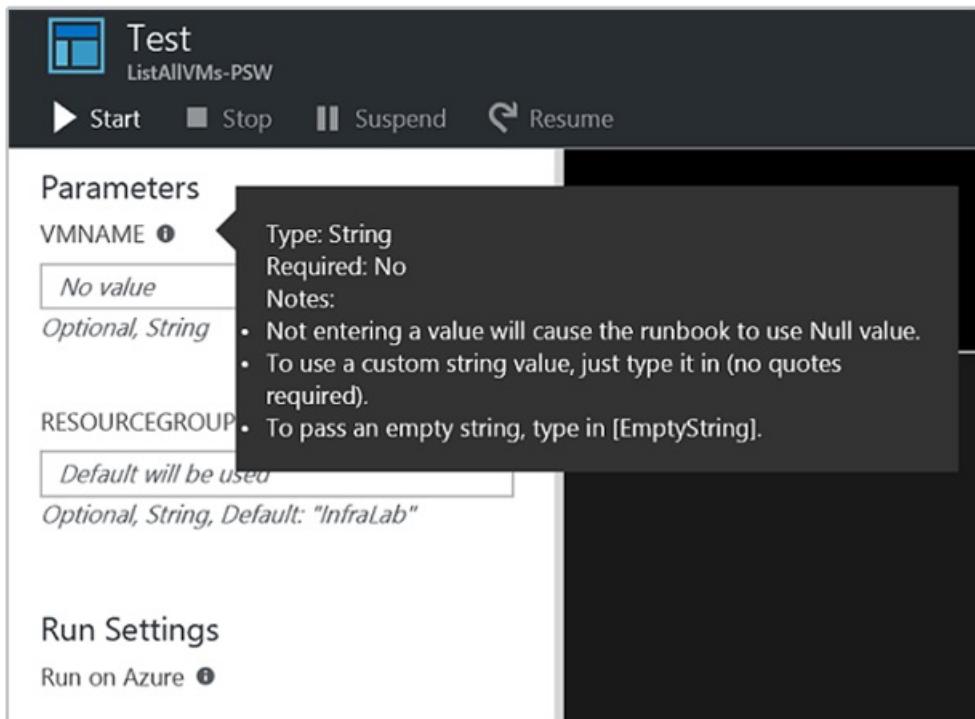
A runbook can be started many ways: through the Azure portal, with a webhook, with PowerShell cmdlets, with the REST API, or with the SDK. Below we discuss different methods for starting a runbook and assigning parameters.

Start a published runbook by using the Azure portal and assign parameters

When you [start the runbook](#), the **Start Runbook** blade opens and you can configure values for the parameters that you just created.



In the label beneath the input box, you can see the attributes that have been set for the parameter. Attributes include mandatory or optional, type, and default value. In the help balloon next to the parameter name, you can see all the key information you need to make decisions about parameter input values. This information includes whether a parameter is mandatory or optional. It also includes the type and default value (if any), and other helpful notes.



NOTE

String type parameters support **Empty** String values. Entering **[EmptyString]** in the input parameter box will pass an empty string to the parameter. Also, String type parameters don't support **Null** values being passed. If you don't pass any value to the String parameter, then PowerShell will interpret it as null.

Start a published runbook by using PowerShell cmdlets and assign parameters

- **Azure Resource Manager cmdlets:** You can start an Automation runbook that was created in a resource group by using [Start-AzureRmAutomationRunbook](#).

Example:

```
$params = @{"VMName"="WSVMClassic";"resourceGroupName"="WSVMClassicSG"}  
  
Start-AzureRmAutomationRunbook -AutomationAccountName "TestAutomation" -Name "Get-AzureVMGraphical" -  
ResourceGroupName $resourceGroupName -Parameters $params
```

- **Azure Service Management cmdlets:** You can start an automation runbook that was created in a default resource group by using [Start-AzureAutomationRunbook](#).

Example:

```
$params = @{"VMName"="WSVMClassic"; "ServiceName"="WSVMClassicSG"}  
  
Start-AzureAutomationRunbook -AutomationAccountName "TestAutomation" -Name "Get-AzureVMGraphical" -  
Parameters $params
```

NOTE

When you start a runbook by using PowerShell cmdlets, a default parameter, **MicrosoftApplicationManagementStartedBy** is created with the value **PowerShell**. You can view this parameter in the **Job details** blade.

Start a runbook by using an SDK and assign parameters

- **Azure Resource Manager method:** You can start a runbook by using the SDK of a programming language.

Below is a C# code snippet for starting a runbook in your Automation account. You can view all the code at our [GitHub repository](#).

```
public Job StartRunbook(string runbookName, IDictionary<string, string> parameters = null)
{
    var response = AutomationClient.Jobs.Create(resourceGroupName, automationAccount, new
JobCreateParameters
{
    Properties = new JobCreateProperties
    {
        Runbook = new RunbookAssociationProperty
        {
            Name = runbookName
        },
        Parameters = parameters
    }
});
return response.Job;
}
```

- **Azure Service Management method:** You can start a runbook by using the SDK of a programming language. Below is a C# code snippet for starting a runbook in your Automation account. You can view all the code at our [GitHub repository](#).

```
public Job StartRunbook(string runbookName, IDictionary<string, string> parameters = null)
{
    var response = AutomationClient.Jobs.Create(automationAccount, new JobCreateParameters
{
    Properties = new JobCreateProperties
    {
        Runbook = new RunbookAssociationProperty
        {
            Name = runbookName
        },
        Parameters = parameters
    }
});
return response.Job;
}
```

To start this method, create a dictionary to store the runbook parameters, **VMName** and **resourceGroupName**, and their values. Then start the runbook. Below is the C# code snippet for calling the method that's defined above.

```
IDictionary<string, string> RunbookParameters = new Dictionary<string, string>();

// Add parameters to the dictionary.
RunbookParameters.Add("VMName", "WSVMClassic");
RunbookParameters.Add("resourceGroupName", "WSSC1");

//Call the StartRunbook method with parameters
StartRunbook("Get-AzureVMGraphical", RunbookParameters);
```

Start a runbook by using the REST API and assign parameters

A runbook job can be created and started with the Azure Automation REST API by using the **PUT** method with the following request URI.

```
https://management.core.windows.net/<subscription-id>/cloudServices/<cloud-service-name>/resources/automation/~/automationAccounts/<automation-account-name>/jobs/<job-id>?api-version=2014-12-08`
```

In the request URI, replace the following parameters:

- **subscription-id:** Your Azure subscription ID.
- **cloud-service-name:** The name of the cloud service to which the request should be sent.
- **automation-account-name:** The name of your automation account that's hosted within the specified cloud service.
- **job-id:** The GUID for the job. GUIDs in PowerShell can be created by using the **[GUID]::NewGuid().ToString()** command.

In order to pass parameters to the runbook job, use the request body. It takes the following two properties provided in JSON format:

- **Runbook name:** Required. The name of the runbook for the job to start.
- **Runbook parameters:** Optional. A dictionary of the parameter list in (name, value) format where name should be of String type and value can be any valid JSON value.

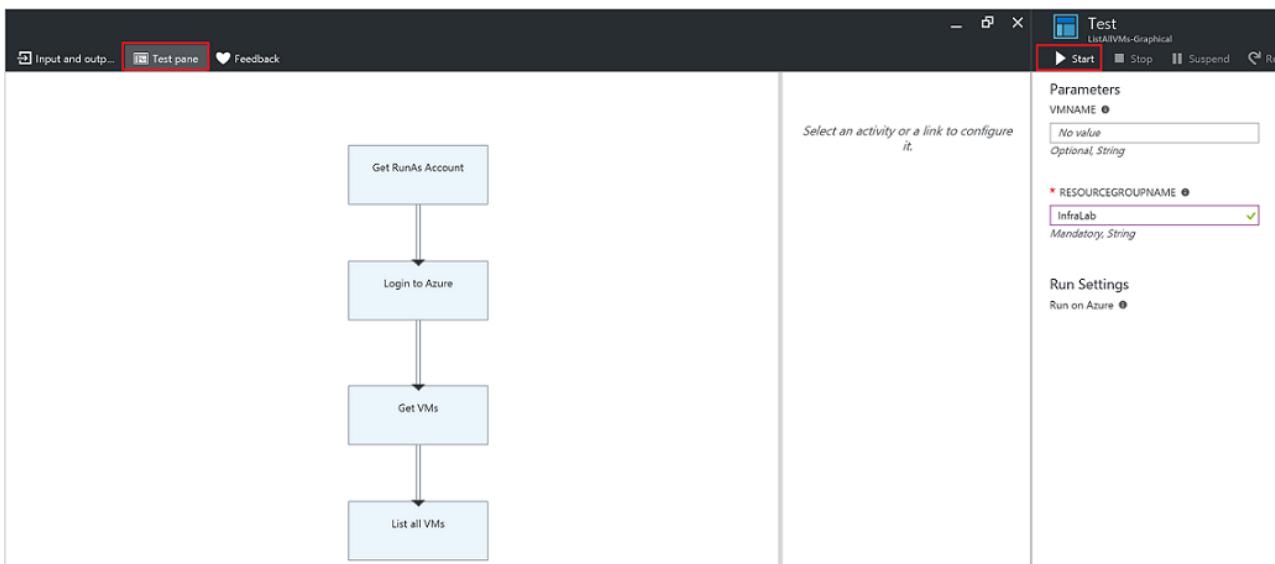
If you want to start the **Get-AzureVMTextual** runbook that was created earlier with **VMName** and **resourceGroupName** as parameters, use the following JSON format for the request body.

```
{  
  "properties":{  
    "runbook":{  
      "name":"Get-AzureVMTextual"},  
    "parameters":{  
      "VMName":"WSVMClassic",  
      "resourceGroupName":"WSCS1"}  
  }  
}
```

A HTTP status code 201 is returned if the job is successfully created. For more information on response headers and the response body, refer to the article about how to [create a runbook job by using the REST API](#).

Test a runbook and assign parameters

When you [test the draft version of your runbook](#) by using the test option, the **Test** blade opens and you can configure values for the parameters that you just created.



Link a schedule to a runbook and assign parameters

You can [link a schedule](#) to your runbook so that the runbook starts at a specific time. You assign input parameters when you create the schedule, and the runbook will use these values when it is started by the schedule. You can't save the schedule until all mandatory parameter values are provided.

Create a webhook for a runbook and assign parameters

You can create a [webhook](#) for your runbook and configure runbook input parameters. You can't save the webhook until all mandatory parameter values are provided.

When you execute a runbook by using a webhook, the predefined input parameter **Webhookdata** is sent, along with the input parameters that you defined. You can click to expand the **WebhookData** parameter for more details.

NAME	INPUT VALUE
RESOURCEGROUPNAME	WSVM1Classic
VMNAME	WSVM1Classic
WEBHOOKDATA	{"WebhookName": "Webhook1", "RequestBo..."}

Next steps

- For more information on runbook input and output, see [Azure Automation: runbook input, output, and nested runbooks](#).
- For details about different ways to start a runbook, see [Starting a runbook](#).
- To edit a textual runbook, refer to [Editing textual runbooks](#).
- To edit a graphical runbook, refer to [Graphical authoring in Azure Automation](#).

Error handling in Azure Automation graphical runbooks

2/2/2017 • 3 min to read • [Edit on GitHub](#)

A key runbook design principal to consider is identifying different issues that a runbook might experience. These issues can include success, expected error states, and unexpected error conditions.

Runbooks should include error handling. To validate the output of an activity or handle an error with graphical runbooks, you could use a Windows PowerShell code activity, define conditional logic on the output link of the activity, or apply another method.

Often, if there is a non-terminating error that occurs with a runbook activity, any activity that follows is processed regardless of the error. The error is likely to generate an exception, but the next activity is still allowed to run. This is the way that PowerShell is designed to handle errors.

The types of PowerShell errors that can occur during execution are terminating or non-terminating. The differences between terminating and non-terminating errors are as follows:

- **Terminating error:** A serious error during execution that halts the command (or script execution) completely. Examples include non-existent cmdlets, syntax errors that prevent a cmdlet from running, or other fatal errors.
- **Non-terminating error:** A non-serious error that allows execution to continue despite the failure. Examples include operational errors such as file not found errors and permissions problems.

Azure Automation graphical runbooks have been improved with the capability to include error handling. You can now turn exceptions into non-terminating errors and create error links between activities. This process allows a runbook author to catch errors and manage realized or unexpected conditions.

When to use error handling

Whenever there is a critical activity that throws an error or exception, it's important to prevent the next activity in your runbook from processing and to handle the error appropriately. This is especially critical when your runbooks are supporting a business or service operations process.

For each activity that can produce an error, the runbook author can add an error link pointing to any other activity. The destination activity can be of any type, including code activities, invoking a cmdlet, invoking another runbook, and so on.

In addition, the destination activity can also have outgoing links. These links can be regular links or error links. This means the runbook author can implement complex error-handling logic without resorting to a code activity. The recommended practice is to create a dedicated error-handling runbook with common functionality, but it's not mandatory. Error-handling logic in a PowerShell code activity it isn't the only option.

For example, consider a runbook that tries to start a VM and install an application on it. If the VM doesn't start correctly, it performs two actions:

1. It sends a notification about this problem.
2. It starts another runbook that automatically provisions a new VM instead.

One solution is to have an error link pointing to an activity that handles step one. For example, you could connect the **Write-Warning** cmdlet to an activity for step two, such as the **Start-AzureRmAutomationRunbook** cmdlet.

You could also generalize this behavior for use in many runbooks by putting these two activities in a separate error handling runbook and following the guidance suggested earlier. Before calling this error-handling runbook, you could construct a custom message from the data in the original runbook, and then pass it as a parameter to the error-handling runbook.

How to use error handling

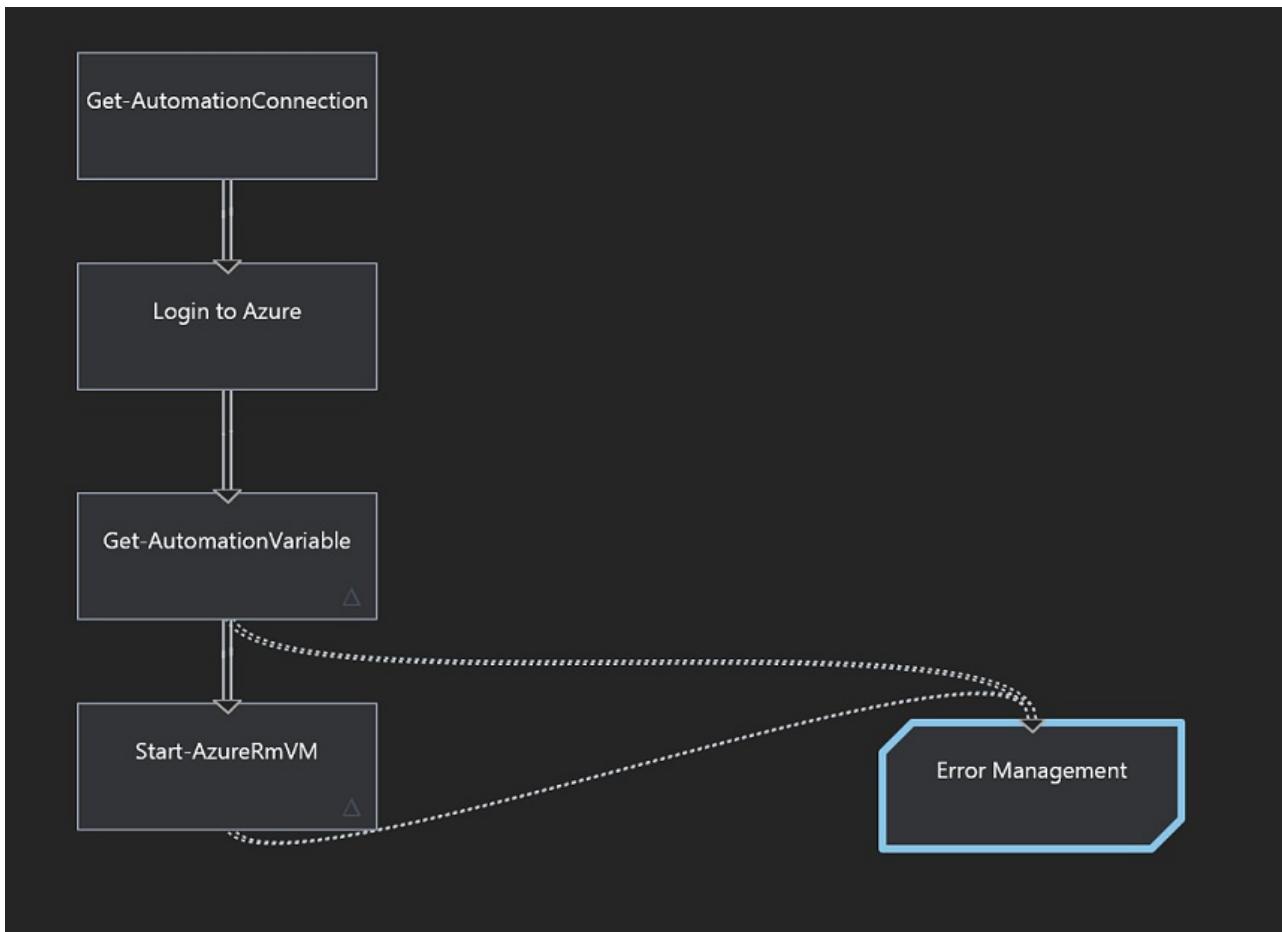
Each activity has a configuration setting that turns exceptions into non-terminating errors. By default, this setting is disabled. We recommend that you enable this setting on any activity where you want to handle errors.

By enabling this configuration, you are assuring that both terminating and non-terminating errors in the activity are handled as non-terminating errors, and can be handled with an error link.

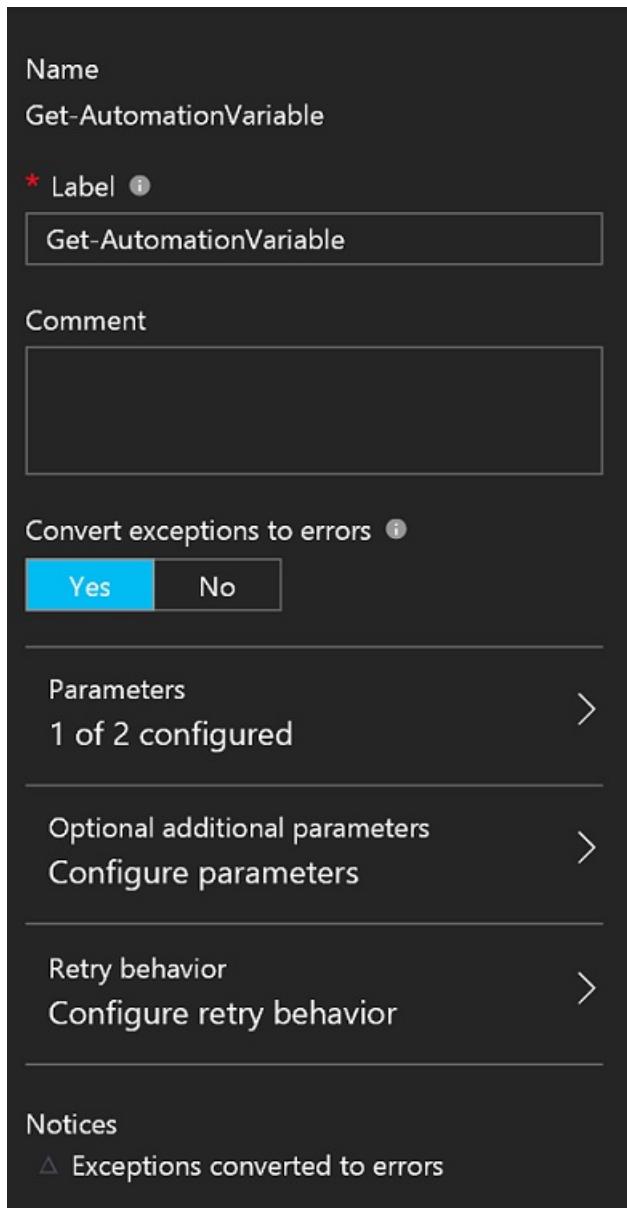
After configuring this setting, you create an activity that handles the error. If an activity produces any error, then the outgoing error links are followed, and the regular links are not, even if the activity produces regular output as well.



In the following example, a runbook retrieves a variable that contains the computer name of a virtual machine. It then attempts to start the virtual machine with the next activity.



The **Get-AutomationVariable** activity and **Start-AzureRmVm** are configured to convert exceptions to errors. If there are problems getting the variable or starting the VM, then errors are generated.



Error links flow from these activities to a single **error management** activity (a code activity). This activity is configured with a simple PowerShell expression that uses the *Throw* keyword to stop processing, along with `$Error.Exception.Message` to get the message that describes the current exception.

PowerShell code ⓘ

```
Throw "Error occurred with an activity." +" Error returned is: " + $Error.Exception.Message|
```

Next steps

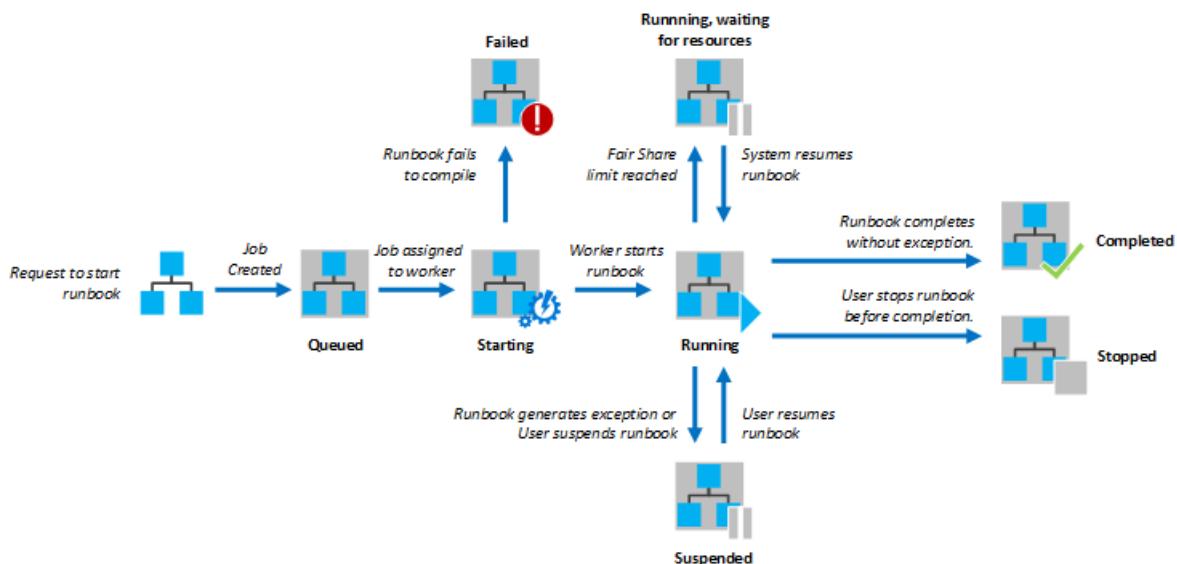
- To learn more about links and link types in graphical runbooks, see [Graphical authoring in Azure Automation](#).
- To learn more about runbook execution, how to monitor runbook jobs, and other technical details, see [Track a runbook job](#).

Runbook execution in Azure Automation

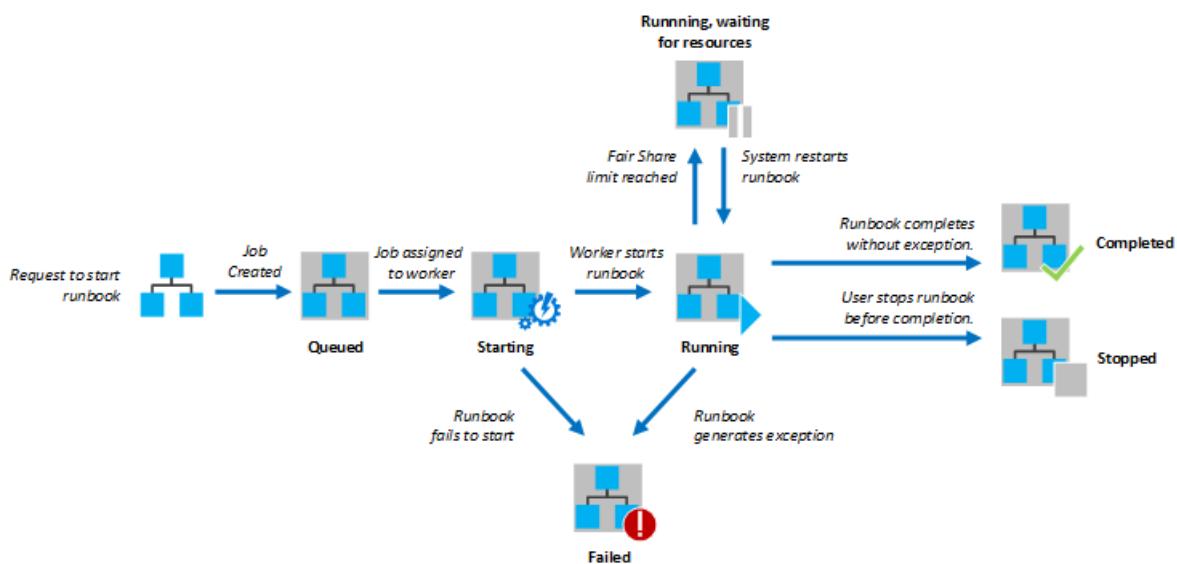
1/17/2017 • 6 min to read • [Edit on GitHub](#)

When you start a runbook in Azure Automation, a job is created. A job is a single execution instance of a runbook. An Azure Automation worker is assigned to run each job. While workers are shared by multiple Azure accounts, jobs from different Automation accounts are isolated from one another. You do not have control over which worker will service the request for your job. A single runbook can have multiple jobs running at one time. When you view the list of runbooks in the Azure portal, it will list the status of the last job that was started for each runbook. You can view the list of jobs for each runbook in order to track the status of each. For a description of the different job statuses, see [Job Statuses](#).

The following diagram shows the lifecycle of a runbook job for [Graphical runbooks](#) and [PowerShell Workflow runbooks](#).



The following diagram shows the lifecycle of a runbook job for [PowerShell runbooks](#).



Your jobs will have access to your Azure resources by making a connection to your Azure subscription. They will only have access to resources in your data center if those resources are accessible from the public cloud.

Job statuses

The following table describes the different statuses that are possible for a job.

STATUS	DESCRIPTION
Completed	The job completed successfully.
Failed	For Graphical and PowerShell Workflow runbooks , the runbook failed to compile. For PowerShell Script runbooks , the runbook failed to start or the job encountered an exception.
Failed, waiting for resources	The job failed because it reached the fair share limit three times and started from the same checkpoint or from the start of the runbook each time.
Queued	The job is waiting for resources on an Automation worker to come available so that it can be started.
Starting	The job has been assigned to a worker, and the system is in the process of starting it.
Resuming	The system is in the process of resuming the job after it was suspended.
Running	The job is running.
Running, waiting for resources	The job has been unloaded because it reached the fair share limit. It will resume shortly from its last checkpoint.
Stopped	The job was stopped by the user before it was completed.
Stopping	The system is in the process of stopping the job.
Suspended	The job was suspended by the user, by the system, or by a command in the runbook. A job that is suspended can be started again and will resume from its last checkpoint or from the beginning of the runbook if it has no checkpoints. The runbook will only be suspended by the system in the case of an exception. By default, ErrorActionPreference is set to Continue meaning that the job will keep running on an error. If this preference variable is set to Stop then the job will suspend on an error. Applies to Graphical and PowerShell Workflow runbooks only.
Suspending	The system is attempting to suspend the job at the request of the user. The runbook must reach its next checkpoint before it can be suspended. If it has already passed its last checkpoint, then it will complete before it can be suspended. Applies to Graphical and PowerShell Workflow runbooks only.

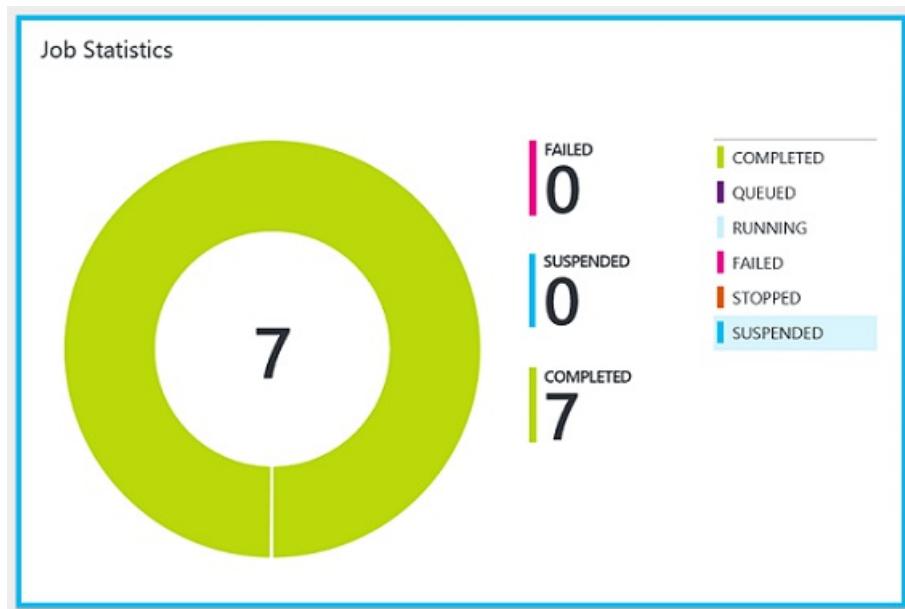
Viewing job status from the Azure portal

You can view a summarized status of all runbook jobs or drill into details of a specific runbook job in the Azure portal or by configuring integration with your Microsoft Operations Management Suite (OMS) Log Analytics workspace to forward runbook job status and job streams. For more information about integrating with OMS

Log Analytics, see [Forward job status and job streams from Automation to Log Analytics \(OMS\)](#).

Automation runbook jobs summary

From the Automation account blade, you can see a summary of all of the runbook jobs for a selected Automation account under **Job Statistics** tile.



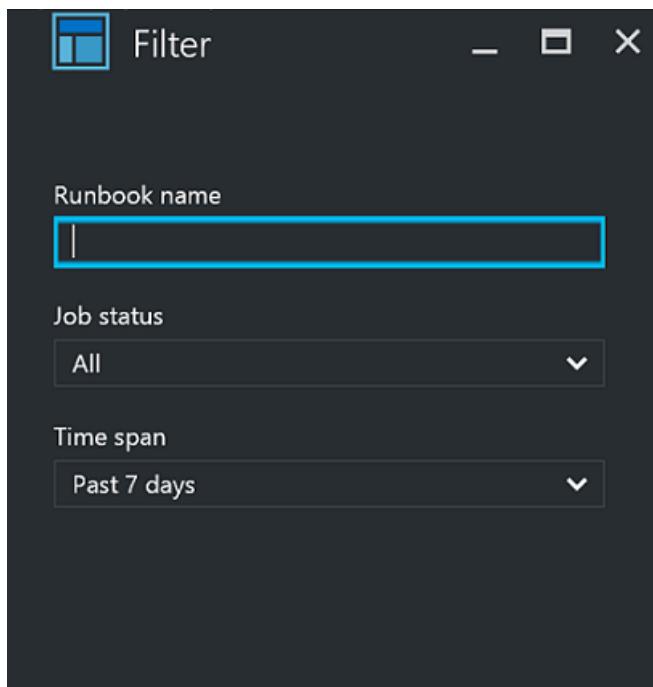
This tile displays a count and graphical representation of the job status for all jobs executed.

Clicking on the tile will present the **Jobs** blade, which includes a summarized list of all jobs executed, with status, job execution start and completion times.

The 'Jobs' blade is displayed in a window. The title bar says 'Jobs'. Below it is a 'Filter Jobs' button. The main area is a table with the following columns: STATUS, RUNBOOK, CREATED, and LAST UPDATED. The table lists seven completed jobs, each with a green checkmark icon in the STATUS column. All jobs are associated with the runbook 'Stop-AllVMs' and were created on 11/1/2016 at 6:00 PM, with the last update occurring between 6:08 PM and 6:12 PM.

STATUS	RUNBOOK	CREATED	LAST UPDATED
✓ Completed	Stop-AllVMs	11/1/2016 6:00 PM	11/1/2016 6:08 PM
✓ Completed	Stop-AllVMs	10/31/2016 6:00 PM	10/31/2016 6:12 PM
✓ Completed	Stop-AllVMs	10/30/2016 6:00 PM	10/30/2016 6:08 PM
✓ Completed	Stop-AllVMs	10/29/2016 6:00 PM	10/29/2016 6:09 PM
✓ Completed	Stop-AllVMs	10/28/2016 6:00 PM	10/28/2016 6:08 PM
✓ Completed	Stop-AllVMs	10/27/2016 6:00 PM	10/27/2016 6:08 PM
✓ Completed	Stop-AllVMs	10/26/2016 6:00 PM	10/26/2016 6:12 PM

You can filter the list of jobs by selecting **Filter jobs** on the **Jobs** blade and filter on a specific runbook, job status, or from the drop-down list, the date/time range to search within.



Alternatively, you can view job summary details for a specific runbook by selecting that runbook from the **Runbooks** blade in your Automation account, and then select the **Jobs** tile. This will present the **Jobs** blade, and from there you can click on the job record to view its detail and output.

A screenshot of the Azure Management Portal's 'Job' blade. The title bar shows 'Stop-AIIVMs 11/1/2016 6:00 PM' and the 'Job' tab is selected. Below the title are buttons for Resume, Stop, Suspend, and View source. The main area is divided into sections: 'Overview' (Job Summary: Job ID edf36252-6cc9-4c59-92d7-dcaeb2285cac, Created 11/1/2016 6:00 PM, Last updated 11/1/2016 6:08 PM, Ran on Azure, Completed), 'Status' (Errors: 1, Warnings: 0, All Logs), and 'Exception' (None).

Errors	Warnings	All Logs
1 ✖	0 ⚠	[button]

Job Summary

You can view a list of all of the jobs that have been created for a particular runbook and their most recent status. You can filter this list by job status and the range of dates for the last change to the job. Click on the name of a job to view its detailed information and its output. The detailed view of the job includes the values for the runbook parameters that were provided to that job.

You can use the following steps to view the jobs for a runbook.

1. In the Azure Management Portal, select **Automation** and then click the name of an automation account.

2. Click the name of a runbook.
3. Select the **Jobs** tab.
4. Click on the **Job Created** column for a job to view its detail and output.

Retrieving job status using Windows PowerShell

You can use the [Get-AzureRmAutomationJob](#) to retrieve the jobs created for a runbook and the details of a particular job. If you start a runbook with Windows PowerShell using [Start-AzureRmAutomationRunbook](#), then it will return the resulting job. Use [Get-AzureRmAutomationJobOutput](#) to get a job's output.

The following sample commands retrieve the last job for a sample runbook and displays its status, the values provided for the runbook parameters, and the output from the job.

```
$job = (Get-AzureRmAutomationJob -AutomationAccountName "MyAutomationAccount" ` 
    -RunbookName "Test-Runbook" -ResourceGroupName "ResourceGroup01" | sort LastModifiedDate -desc)[0]
$job.Status
$job.JobParameters
Get-AzureRmAutomationJobOutput -ResourceGroupName "ResourceGroup01" ` 
    -AutomationAccountName "MyAutomationAcct" -Id $job.JobId -Stream Output
```

Fair share

In order to share resources among all runbooks in the cloud, Azure Automation will temporarily unload any job after it has been running for 3 hours. [Graphical](#) and [PowerShell Workflow](#) runbooks will be resumed from their last [checkpoint](#). During this time, the job will show a status of Running, Waiting for Resources. If the runbook has no checkpoints or the job had not reached the first checkpoint before being unloaded, then it will restart from the beginning. [PowerShell](#) runbooks are always restarted from the beginning since they don't support checkpoints.

If the runbook restarts from the same checkpoint or from the beginning of the runbook three consecutive times, it will be terminated with a status of Failed, waiting for resources. This is to protect from runbooks running indefinitely without completing, as they are not able to make it to the next checkpoint without being unloaded again. In this case, you will receive the following exception with the failure.

The job cannot continue running because it was repeatedly evicted from the same checkpoint. Please make sure your Runbook does not perform lengthy operations without persisting its state.

When you create a runbook, you should ensure that the time to run any activities between two checkpoints will not exceed 3 hours. You may need to add checkpoints to your runbook to ensure that it does not reach this 3 hour limit or break up long running operations. For example, your runbook might perform a reindex on a large SQL database. If this single operation does not complete within the fair share limit, then the job will be unloaded and restarted from the beginning. In this case, you should break up the reindex operation into multiple steps, such as reindexing one table at a time, and then insert a checkpoint after each operation so that the job could resume after the last operation to complete.

Next steps

- To learn more about the different methods that can be used to start a runbook in Azure Automation, see [Starting a runbook in Azure Automation](#)

Runbook settings

1/17/2017 • 2 min to read • [Edit on GitHub](#)

Each runbook in Azure Automation has multiple settings that help it to be identified and to change its logging behavior. Each of these settings is described below followed by procedures on how to modify them.

Settings

Name and description

You cannot change the name of a runbook after it has been created. The Description is optional and can be up to 512 characters.

Tags

Tags allow you to assign distinct words and phrases to help identify a runbook. For example, when you submit a runbook to the [PowerShell Gallery](#), you specify particular tags to identify which categories the runbook should be listed in. You can specify multiple tags for a runbook by separating them with commas.

Logging

By default, Verbose and Progress records are not written to job history. You can change the settings for a particular runbook to log these records. For more information on these records, see [Runbook Output and Messages](#).

Changing runbook settings

Changing runbook settings with the Azure portal

You can change settings for a runbook in the Azure portal from the **Settings** blade for the runbook.

1. In the Azure portal, select **Automation** and then click the name of an automation account.
2. Select the **Runbooks** tab.
3. Click the name of a runbook and you are directed to the settings blade for the runbook. From here you can specify or modify tags, the runbook description, configure logging and tracing settings, and access support tools to help you solve problems.

Changing runbook settings with Windows PowerShell

You can use the `Set-AzureRmAutomationRunbook` cmdlet to change the settings for a runbook. If you want to specify multiple tags, you can either provide an array or a single string with comma delimited values to the Tags parameter. You can get the current tags with the `Get-AzureRmAutomationRunbook`.

The following sample commands show how to set the properties for a runbook. This sample adds three tags to the existing tags and specifies that verbose records should be logged.

```
$automationAccountName = "MyAutomationAccount"
$runbookName = "Sample-TestRunbook"
$tags = (Get-AzureRmAutomationRunbook -ResourceGroupName "ResourceGroup01" ` 
-AutomationAccountName $automationAccountName -Name $runbookName).Tags
$tags += "Tag1,Tag2,Tag3"
Set-AzureRmAutomationRunbook -ResourceGroupName "ResourceGroup01" ` 
-AutomationAccountName $automationAccountName -Name $runbookName -LogVerbose $true -Tags $tags
```

Next steps

- To learn how to create and retrieve output and error messages from runbooks, see [Runbook Output and Messages](#)
- To understand how to add a runbook that was already developed by the community or other source, or to create your own runbook see [Creating or Importing a Runbook](#)

Managing Azure Automation data

1/17/2017 • 3 min to read • [Edit on GitHub](#)

This article contains multiple topics for managing an Azure Automation environment.

Data retention

When you delete a resource in Azure Automation, it is retained for 90 days for auditing purposes before being removed permanently. You can't see or use the resource during this time. This policy also applies to resources that belong to an automation account that is deleted.

Azure Automation automatically deletes and permanently removes jobs older than 90 days.

The following table summarizes the retention policy for different resources.

DATA	POLICY
Accounts	Permanently removed 90 days after the account is deleted by a user.
Assets	Permanently removed 90 days after the asset is deleted by a user, or 90 days after the account that holds the asset is deleted by a user.
Modules	Permanently removed 90 days after the module is deleted by a user, or 90 days after the account that holds the module is deleted by a user.
Runbooks	Permanently removed 90 days after the resource is deleted by a user, or 90 days after the account that holds the resource is deleted by a user.
Jobs	Deleted and permanently removed 90 days after last being modified. This could be after the job completes, is stopped, or is suspended.
Node Configurations/MOF Files	Old node configuration is permanently removed 90 days after a new node configuration is generated.
DSC Nodes	Permanently removed 90 days after the node is unregistered from Automation Account using Azure portal or the Unregister-AzureRMAutomationDscNode cmdlet in Windows PowerShell. Nodes are also permanently removed 90 days after the account that holds the node is deleted by a user.
Node Reports	Permanently removed 90 days after a new report is generated for that node

The retention policy applies to all users and currently cannot be customized.

Backing up Azure Automation

When you delete an automation account in Microsoft Azure, all objects in the account are deleted including

runbooks, modules, configurations, settings, jobs, and assets. The objects cannot be recovered after the account is deleted. You can use the following information to backup the contents of your automation account before deleting it.

Runbooks

You can export your runbooks to script files using either the Azure Management Portal or the [Get-AzureAutomationRunbookDefinition](#) cmdlet in Windows PowerShell. These script files can be imported into another automation account as discussed in [Creating or Importing a Runbook](#).

Integration modules

You cannot export integration modules from Azure Automation. You must ensure that they are available outside of the automation account.

Assets

You cannot export [assets](#) from Azure Automation. Using the Azure Management Portal, you must note the details of variables, credentials, certificates, connections, and schedules. You must then manually create any assets that are used by runbooks that you import into another automation.

You can use [Azure cmdlets](#) to retrieve details of unencrypted assets and either save them for future reference or create equivalent assets in another automation account.

You cannot retrieve the value for encrypted variables or the password field of credentials using cmdlets. If you don't know these values, then you can retrieve them from a runbook using the [Get-AutomationVariable](#) and [Get-AutomationPSCredential](#) activities.

You cannot export certificates from Azure Automation. You must ensure that any certificates are available outside of Azure.

DSC configurations

You can export your configurations to script files using either the Azure Management Portal or the [Export-AzureRmAutomationDscConfiguration](#) cmdlet in Windows PowerShell. These configurations can be imported and used in another automation account.

Geo-replication in Azure Automation

Geo-replication, standard in Azure Automation accounts, backs up account data to a different geographical region for redundancy. You can choose a primary region when setting up your account, and then a secondary region is assigned to it automatically. The secondary data, copied from the primary region, is continuously updated in case of data loss.

The following table shows the available primary and secondary region pairings.

PRIMARY	SECONDARY
South Central US	North Central US
US East 2	Central US
West Europe	North Europe
South East Asia	East Asia
Japan East	Japan West

In the unlikely event that a primary region data is lost, Microsoft attempts to recover it. If the primary data cannot

be recovered, then geo-failover is performed and the affected customers will be notified about this through their subscription.

Calling an Azure Automation runbook from an OMS Log Analytics alert

2/9/2017 • 4 min to read • [Edit on GitHub](#)

When an alert is configured in Log Analytics to create an alert record if results match a particular criteria, such as a prolonged spike in processor utilization or a particular application process critical to the functionality of a business application fails and writes a corresponding event in the Windows event log, that alert can automatically run an Automation runbook in an attempt to auto-remediate the issue.

There are two options to call a runbook when configuring the alert. Specifically,

1. Using a Webhook.
 - This is the only option available if your OMS workspace is not linked to an Automation account.
 - If you already have an Automation account linked to an OMS workspace, this option is still available.
2. Select a runbook directly.
 - This option is available only when the OMS workspace is linked to an Automation account.

Calling a runbook using a webhook

A webhook allows you to start a particular runbook in Azure Automation through a single HTTP request. Before configuring the [Log Analytics alert](#) to call the runbook using a webhook as an alert action, you will need to first create a webhook for the runbook that will be called using this method. Review and follow the steps in the [create a webhook](#) article and remember to record the webhook URL so that you can reference it while configuring the alert rule.

Calling a runbook directly

If you have the Automation & Control offering installed and configured in your OMS workspace, when configuring the Runbook actions option for the alert, you can view all runbooks from the **Select a runbook** dropdown list and select the specific runbook you want to run in response to the alert. The selected runbook can run in a workspace in the Azure cloud or on a hybrid runbook worker. When the alert is created using the runbook option, a webhook will be created for the runbook. You can see the webhook if you go to the Automation account and navigate to the webhook blade of the selected runbook. If you delete the alert, the webhook is not deleted, but the user can delete the webhook manually. It is not a problem if the webhook is not deleted, it is just an orphaned item that will eventually need to be deleted in order to maintain an organized Automation account.

Characteristics of a runbook (for both options)

Both methods for calling the runbook from the Log Analytics alert have different behavior characteristics that need to be understood before you configure your alert rules.

- You must have a runbook input parameter called **WebhookData** that is **Object** type. It can be mandatory or optional. The alert passes the search results to the runbook using this input parameter.

```
param
(
    [Parameter (Mandatory=$true)]
    [object] $WebhookData
)
```

- You must have code to convert the WebhookData to a PowerShell object.

```
$SearchResults = (ConvertFrom-Json $WebhookData.RequestBody).SearchResults.value
```

\$SearchResults will be an array of objects; each object contains the fields with values from one search result

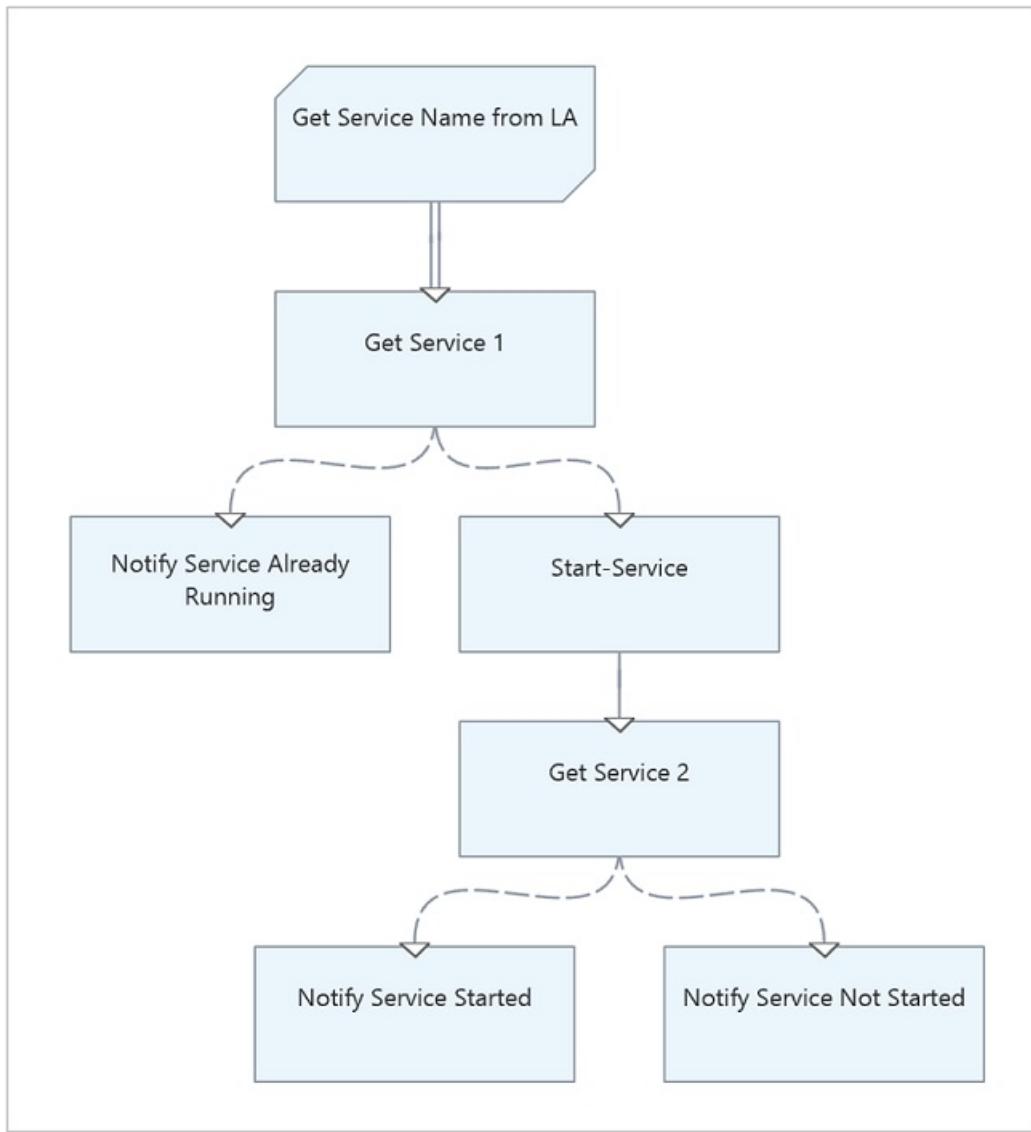
WebhookData inconsistencies between the webhook option and runbook option

- When configuring an alert to call a Webhook, enter a webhook URL you created for a runbook, and click the **Test Webhook** button. The resulting WebhookData sent to the runbook does not contain either *.SearchResult* or *.SearchResults*.
- If you save that alert, when the alert triggers and calls the webhook, the WebhookData sent to the runbook contains *.SearchResult*.
- If you create an alert, and configure it to call a runbook (which also creates a webhook), when the alert triggers it sends WebhookData to the runbook that contains *.SearchResults*.

Thus in the code example above, you will need to get *.SearchResult* if the alert calls a webhook, and will need to get *.SearchResults* if the alert calls a runbook directly.

Example walkthrough

We will demonstrate how this works by using the following example graphical runbook, which starts a Windows service.



The runbook has one input parameter of type **Object** that is called **WebhookData** and includes the webhook data passed from the alert containing `.SearchResults`.

The screenshot shows the 'Runbook Input Par...' configuration page. It includes fields for Name (WebhookData), Description, Type (Object), Mandatory (Yes selected), and Default value (Mandatory parameters can't have a default value).

For this example, in Log Analytics we created two custom fields, *SvcDisplayName_CF* and *SvcState_CF*, to extract the service display name and the state of the service (i.e. running or stopped) from the event written to the System event log. We then create an alert rule with the following search query:

Type=Event SvcDisplayName_CF="Print Spooler" SvcState_CF="stopped" so that we can detect when the Print Spooler service is stopped on the Windows system. It can be any service of interest, but for this example we are referencing one of the pre-existing services that are included with the Windows OS. The alert action is configured to execute our runbook used in this example and run on the Hybrid Runbook Worker, which are enabled on the target systems.

The runbook code activity **Get Service Name from LA** will convert the JSON-formatted string into an object type and filter on the item *SvcDisplayName_CF* to extract the display name of the Windows service and pass this onto the next activity which will verify the service is stopped before attempting to restart it. *SvcDisplayName_CF* is a [custom field](#) created in Log Analytics to demonstrate this example.

```
$SearchResults = (ConvertFrom-Json $WebhookData.RequestBody).SearchResults.value  
$SearchResults.SvcDisplayName_CF
```

When the service stops, the alert rule in Log Analytics will detect a match and trigger the runbook and send the alert context to the runbook. The runbook will take action to verify the service is stopped, and if so attempt to restart the service and verify it started correctly and output the results.

Alternatively if you don't have your Automation account linked to your OMS workspace, you would configure the alert rule with a webhook action to trigger the runbook and configure the runbook to convert the JSON-formatted string and filter on *.SearchResult* following the guidance mentioned earlier.

Next steps

- To learn more about alerts in Log Analytics and how to create one, see [Alerts in Log Analytics](#).

- To understand how to trigger runbooks using a webhook, see [Azure Automation webhooks](#).

Azure Automation DSC Overview

2/3/2017 • 8 min to read • [Edit on GitHub](#)

What is Azure Automation DSC?

Deploying and maintaining the desired state of your servers and application resources can be tedious and error prone. With Azure Automation Desired State Configuration (DSC), you can consistently deploy, reliably monitor, and automatically update the desired state of all your IT resources, at scale from the cloud. Built on PowerShell DSC, Automation DSC can align machine configuration with a specific state across physical and virtual machines (VMs), using Windows or Linux, and in the cloud or on-premises. You can enable continuous IT services delivery with consistent control and manage rapid change across your heterogeneous hybrid IT environment with ease.

Azure Automation DSC builds on top of the fundamentals introduced in PowerShell DSC to provide an even easier configuration management experience. Azure Automation DSC brings the same management layer to [PowerShell Desired State Configuration](#) as Azure Automation offers for PowerShell scripting today.

Azure Automation DSC allows you to [author and manage PowerShell Desired State Configurations](#), import [DSC Resources](#), and generate DSC Node Configurations (MOF documents), all in the cloud. These DSC items are placed on the Azure Automation [DSC pull server](#) so that target nodes automatically receive configurations, conform to the desired state, and report back on their compliance. Azure Automation can target virtual or physical machines, in the cloud or on-premises.

Prefer watching to reading? Have a look at the following video from May 2015, when Azure Automation DSC was first announced. **Note:** While the concepts and lifecycle discussed in this video are correct, Azure Automation DSC has progressed a lot since this video was recorded. It is now generally available, has a much more extensive UI in the Azure portal, and supports many additional capabilities.

Azure Automation DSC Terms

Configuration

PowerShell DSC introduced a new concept called configurations. Configurations allow you to define, via PowerShell syntax, the desired state of your environment. To use DSC to configure your environment, first define a Windows PowerShell script block using the configuration keyword, followed by an identifier, then braces ({}) to delimit the block.

```
1 Configuration MyConfiguration {  
2     ...  
3 }
```

Inside the configuration block, you can define node configurations that specify the desired configuration for a set of nodes (computers) in your environment that should be configured the same. In this way, a node configuration represents a "role" for one or more nodes to assume. A node configuration block starts with the node keyword. Follow this keyword with the name of the role, which can be a variable or expression. After the role name, use braces {} to delimit the node configuration block.

```
1 Configuration MyConfiguration {
2     Node "webserver" {
3         ...
4     }
5 }
6 }
7 }
```

Inside the node configuration block, you can define resource blocks to configure specific DSC resources. A resource block starts with the name of the resource, followed by the identifier you want to specify for that block, then braces {} to delimit the block.

```
1 Configuration MyConfiguration {
2     Node "webserver" {
3         WindowsFeature IIS {
4             Ensure="Present"
5             Name= "Web-Server"
6         }
7     }
8 }
9 }
10 }
11 }
```

For more detailed information about the configuration keyword, see: [DSC Configurations](#)

Running (compiling) a DSC configuration produces one or more DSC node configurations (MOF documents), which are what DSC nodes apply to comply with desired state.

Azure Automation DSC allows you to import, author, and compile DSC configurations in Azure Automation, similar to how runbooks can be imported, authored, and started in Azure Automation.

IMPORTANT

A configuration should contain only one configuration block, with the same name as the configuration, in Azure Automation DSC.

Node Configuration

When a DSC Configuration is compiled, one or more node configurations are produced depending on the Node blocks in the configuration. A node configuration is the same as a "MOF," or "configuration document". Node configurations represent a "role," such as webserver or worker, which desired state one or more nodes should assume or check for compliance against. Names of node configurations in Azure Automation DSC take the form of "Configuration Name.NodeConfigurationBlockName".

PS DSC nodes become aware of node configurations they should enact via either DSC push, or pull methods. Azure Automation DSC relies on the DSC pull method, where nodes request node configurations they should apply from the Azure Automation DSC pull server. Because the nodes make the request to Azure Automation DSC, nodes can be behind firewalls, have all inbound ports closed, etc. They only need outbound access to the Internet (either directly or via a proxy).

Node

A DSC node is any machine that has its configuration managed by DSC. This machine could be a Windows or Linux Azure VM, on-premises VM / physical host, or machine in another public cloud. Nodes enact node configurations to become and maintain compliance with the desired state they define. Nodes also report back to a reporting server on their configuration and compliance status.

Azure Automation DSC makes onboarding of nodes easy and flexible. Azure Automation DSC allows changing the node configuration assigned to each node. When a node configuration change happens, the node automatically assumes a different role by changing its configuration and begins reporting against its new role.

Resource

DSC resources are building blocks that you can use to define a Windows PowerShell Desired State Configuration (DSC) configuration. DSC comes with a set of built-in resources such as those for files and folders, server features and roles, registry settings, environment variables, and services and processes. To learn about the full list of built-in DSC resources and how to use them, see [Built-In Windows PowerShell Desired State Configuration Resources](#).

DSC resources can also be imported as part of PowerShell Modules to extend the set of built-in DSC resources. If a node configuration the node is meant to enact contains references to non-default resources, those resources are pulled down by DSC nodes from the DSC pull server. To learn how to create custom resources, see [Build Custom Windows PowerShell Desired State Configuration Resources](#).

Azure Automation DSC ships with all the same built-in DSC resources as does PS DSC. Additional resources can be added to Azure Automation DSC by importing PowerShell modules containing the resources into Azure Automation.

Compilation Job

A compilation job in Azure Automation DSC is an instance of compilation of a configuration, to create one or more node configurations. They are similar to Azure Automation runbook jobs, except that they do not actually perform any task except to create node configurations. Any node configurations created by a compilation job are automatically placed on the Azure Automation DSC pull server, and overwrite previous versions of node configurations, if they existed for this configuration. The name of a node configuration produced by a compilation job takes the form of "ConfigurationName.NodeConfigurationBlockName". For example, compiling the following configuration would produce a single node configuration called "MyConfiguration.webserver"

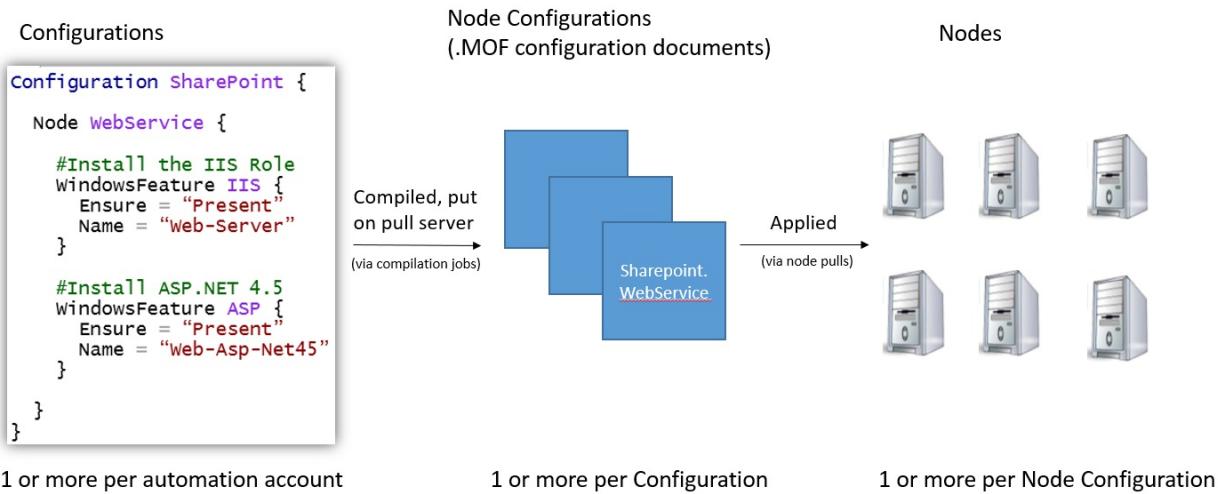
```
1 Configuration MyConfiguration {
2     Node "webserver" {
3         WindowsFeature IIS {
4             Ensure="Present"
5             Name= "Web-Server"
6         }
7     }
8 }
```

NOTE

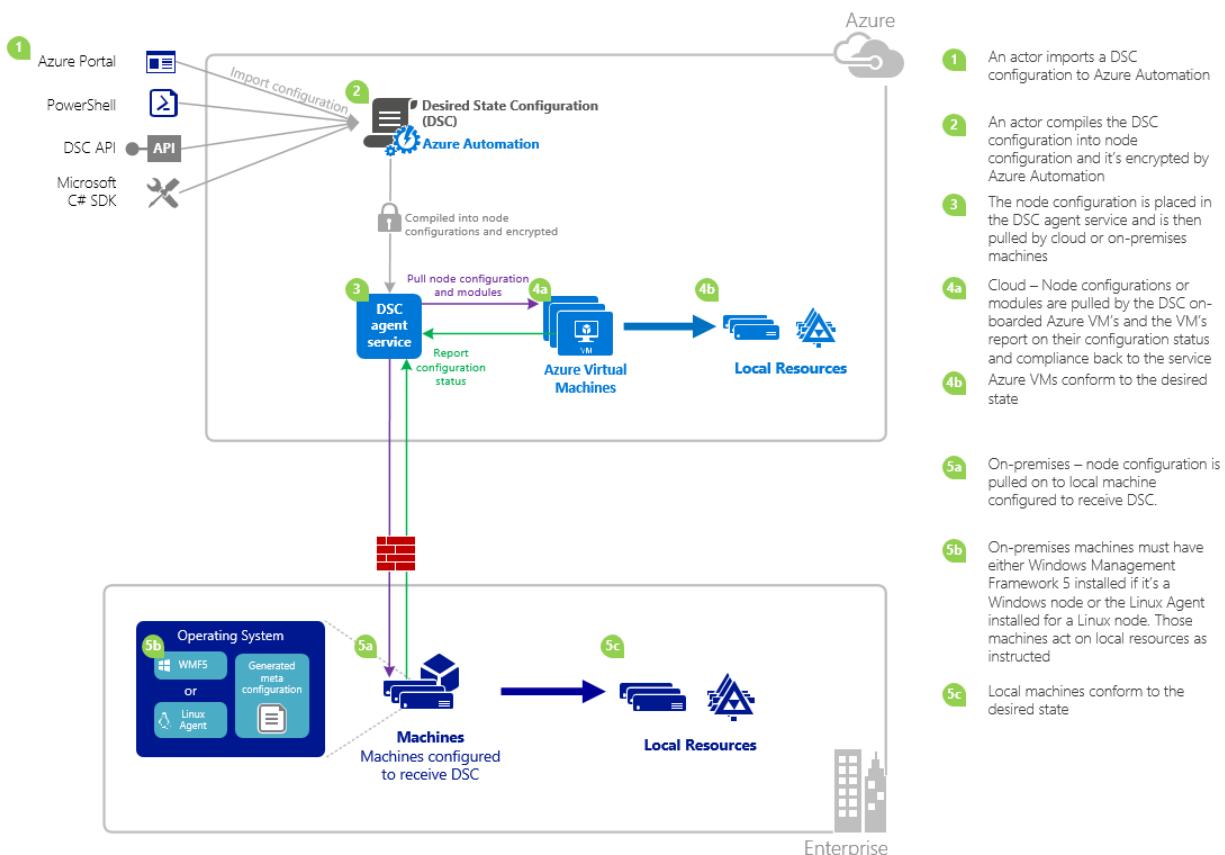
Just like runbooks, configurations can be published. This is not related to putting DSC items onto the Azure Automation DSC pull server. Compilation jobs cause DSC items to be placed on the Azure Automation DSC pull server. For more information on "publishing" in Azure Automation, see [Publishing a Runbook](#).

Azure Automation DSC LifeCycle

Going from an empty automation account to a managed set of correctly configured nodes involves a set of processes for defining configurations, turning those configurations into node configurations, and onboarding nodes to Azure Automation DSC and to those node configurations. The following diagram illustrates the Azure Automation DSC lifecycle:



The following image illustrates detailed step-by-step process in the life cycle of DSC. It includes different ways a configuration is imported and applied to nodes in Azure Automation, components required for an on-premises machine to support DSC and interactions between different components.



Gotchas / Known Issues:

- The latest version of WMF 5 must be installed for the PowerShell DSC agent for Windows to be able to communicate with Azure Automation. The latest version of the PowerShell DSC agent for Linux must be installed for Linux to be able to communicate with Azure Automation.
- If a machine is already registered as a node in Azure Automation DSC before upgrading to WMF5, unregister it from Azure Automation DSC and reregister it after completing the WMF 5 install. Delete the \$env:windir\system32\configuration\DSCEngineCache.mof file before re-registering
- If WMF 5 RTM is installed on top of WMF 5 Production Preview, DSC may not work correctly. To fix this issue, run the following command in an elevated PowerShell session (run as administrator):


```
mofcomp $env:windir\system32\wbem\DsccoreConfProv.mof
```

- Azure Automation DSC does not currently support partial or composite DSC configurations. However, DSC composite resources can be imported and used in Azure Automation DSC Configurations like in local PowerShell, enabling configuration reuse.
- The traditional PowerShell DSC pull server expects module zips to be placed on the pull server in the format **ModuleName_Version.zip**. Azure Automation expects PowerShell modules to be imported with names in the form of **ModuleName.zip**. See [this blog post](#) for more info on the Integration Module format needed to import the module into Azure Automation.
- PowerShell modules imported into Azure Automation cannot contain .doc or .docx files. Some PowerShell modules containing DSC resources contain these files, for help purposes. These files should be removed from modules before importing into Azure Automation.
- When a node is first registered with an Azure Automation account, or the node is changed to be mapped to a different node configuration server-side, its status is 'Compliant'. This occurs even if the node's status is not compliant with the node configuration it is now mapped to. After the node sends its first report after registration or a node configuration mapping change, the node status can be trusted.
- When onboarding an Azure Windows VM for management by Azure Automation DSC using any of our direct onboarding methods, it could take up to an hour for the VM to show up as a DSC node in Azure Automation. This is due to the installation of Windows Management Framework 5.0 on the VM by the Azure VM DSC extension, which is required to onboard the VM to Azure Automation DSC.
- After registering, each node automatically negotiates a unique certificate for authentication that expires after one year. Currently, the PowerShell DSC registration protocol cannot automatically renew certificates when they are nearing expiration, so you need to reregister the nodes after a year's time. Before reregistering, ensure that each node is running Windows Management Framework 5.0 RTM. If a node's authentication certificate expires, and the node is not reregistered, the node will be unable to communicate with Azure Automation and is marked 'Unresponsive.' Reregistration is performed in the same way you registered the node initially. Reregistration performed 90 days or less from the certificate expiration time, or at any point after the certificate expiration time, will result in a new certificate being generated and used.

Related Articles

- [Onboarding machines for management by Azure Automation DSC](#)
- [Compiling configurations in Azure Automation DSC](#)
- [Azure Automation DSC cmdlets](#)
- [Azure Automation DSC pricing](#)
- [Continuous Deployment to IaaS VMs Using Azure Automation DSC and Chocolatey](#)

Getting started with Azure Automation DSC

1/23/2017 • 8 min to read • [Edit on GitHub](#)

This topic explains how to do the most common tasks with Azure Automation Desired State Configuration (DSC), such as creating, importing, and compiling configurations, onboarding machines to manage, and viewing reports. For an overview of what Azure Automation DSC is, see [Azure Automation DSC Overview](#). For DSC documentation, see [Windows PowerShell Desired State Configuration Overview](#).

This topic provides a step-by-step guide to using Azure Automation DSC. If you want a sample environment that is already set up without following the steps described in this topic, you can use [the following ARM template](#). This template sets up a completed Azure Automation DSC environment, including an Azure VM that is managed by Azure Automation DSC.

Prerequisites

To complete the examples in this topic, the following are required:

- An Azure Automation account. For instructions on creating an Azure Automation Run As account, see [Azure Run As Account](#).
- An Azure Resource Manager VM (not Classic) running Windows Server 2008 R2 or later. For instructions on creating a VM, see [Create your first Windows virtual machine in the Azure portal](#)

Creating a DSC configuration

We will create a simple [DSC configuration](#) that ensures either the presence or absence of the **Web-Server** Windows Feature (IIS), depending on how you assign nodes.

1. Start the Windows PowerShell ISE (or any text editor).
2. Type the following text:

```
configuration TestConfig
{
    Node WebServer
    {
        WindowsFeature IIS
        {
            Ensure          = 'Present'
            Name           = 'Web-Server'
            IncludeAllSubFeature = $true
        }
    }

    Node NotWebServer
    {
        WindowsFeature IIS
        {
            Ensure          = 'Absent'
            Name           = 'Web-Server'
        }
    }
}
```

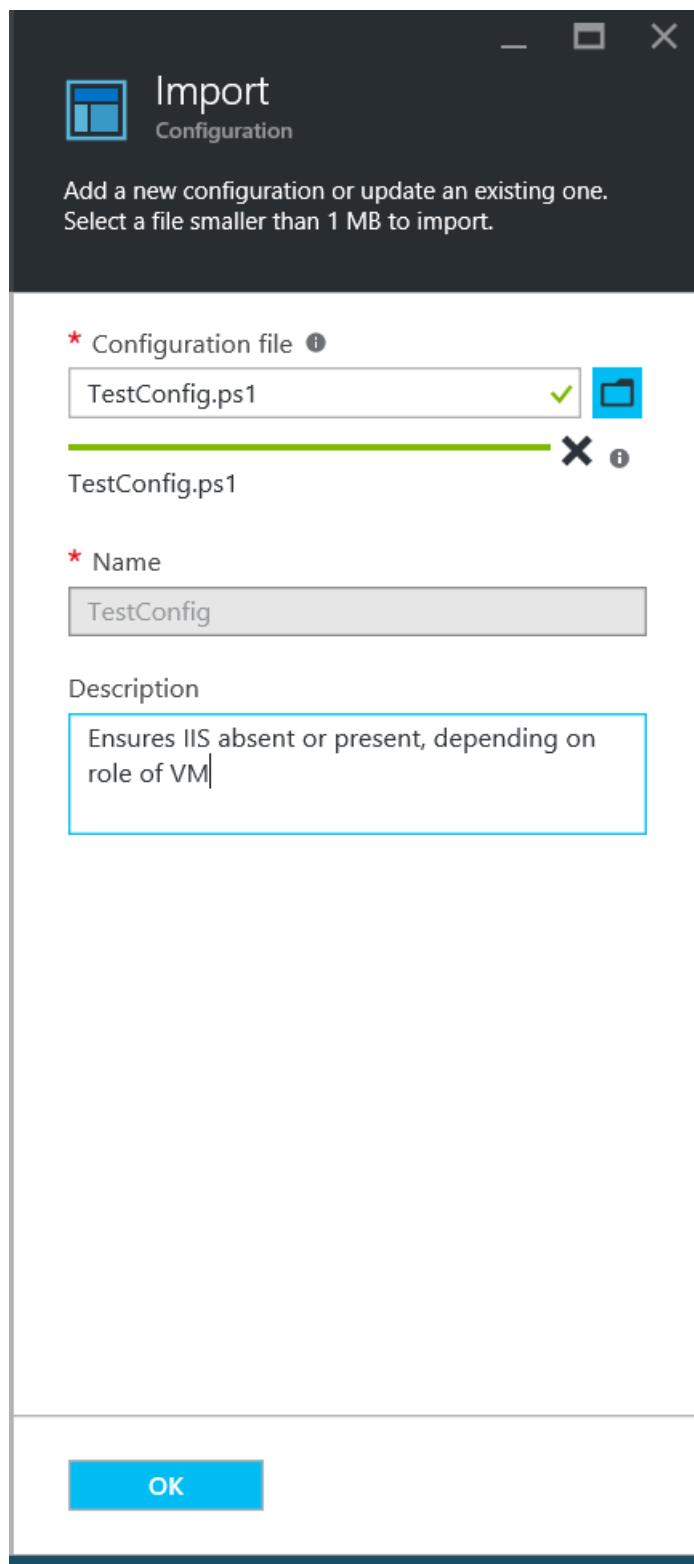
3. Save the file as `TestConfig.ps1`.

This configuration calls one resource in each node block, the [WindowsFeature resource](#), that ensures either the presence or absence of the **Web-Server** feature.

Importing a configuration into Azure Automation

Next, we'll import the configuration into the Automation account.

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Configurations**.
4. On the **DSC Configurations** blade, click **Add a configuration**.
5. On the **Import Configuration** blade, browse to the `TestConfig.ps1` file on your computer.



6. Click **OK**.

Viewing a configuration in Azure Automation

After you have imported a configuration, you can view it in the Azure portal.

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Configurations**
4. On the **DSC Configurations** blade, click **TestConfig** (this is the name of the configuration you imported in the previous procedure).
5. On the **TestConfig Configuration** blade, click **View configuration source**.

The screenshot shows the 'TestConfig Configuration' blade in the Azure portal. At the top, there are three buttons: 'Compile', 'Export', and 'Delete'. Below that is a section titled 'Essentials' with the following details:

Resource group AADsc1	Account AADsc1
Location eastus2	Subscription name Windows Azure MSDN - Visual Studio Ulti...
Subscription ID <Subscription ID>	Status Published
Last published 5/10/2016 2:45 PM	Configuration source View configuration source

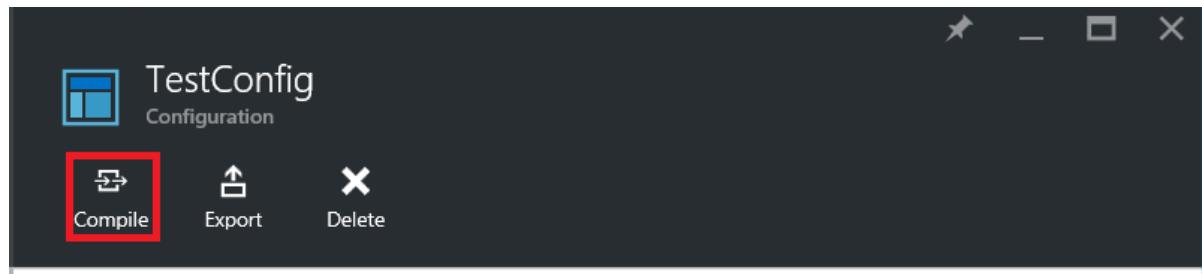
The 'View configuration source' link is highlighted with a red box. Below this is a 'Compilation jobs' section with columns: STATUS, CREATED, and LAST UPDATED. A message states 'No compilation jobs found.' At the bottom, status indicators say 'Available on Pull Server' and 'Add tiles' with a plus sign.

A **TestConfig Configuration source** blade opens, displaying the PowerShell code for the configuration.

Compiling a configuration in Azure Automation

Before you can apply a desired state to a node, a DSC configuration defining that state must be compiled into one or more node configurations (MOF document), and placed on the Automation DSC Pull Server. For a more detailed description of compiling configurations in Azure Automation DSC, see [Compiling configurations in Azure Automation DSC](#). For more information about compiling configurations, see [DSC Configurations](#).

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Configurations**
4. On the **DSC Configurations** blade, click **TestConfig** (the name of the previously imported configuration).
5. On the **TestConfig Configuration** blade, click **Compile**, and then click **Yes**. This starts a compilation job.



The screenshot shows the 'TestConfig' Configuration blade in the Azure portal. At the top, there are three buttons: 'Compile' (highlighted with a red box), 'Export', and 'Delete'. Below this is the 'Essentials' section, which displays various configuration details:

Resource group	AADsc1	Account	AADsc1
Location	eastus2	Subscription name	Windows Azure MSDN - Visual Studio Ulti...
Subscription ID	<Subscription ID>	Status	Published
Last published	5/10/2016 2:45 PM	Configuration source	View configuration source

Below the essentials section is a 'Compilation jobs' table:

STATUS	CREATED	LAST UPDATED
No compilation jobs found.		

At the bottom of the blade, there are two status indicators: 'Available on Pull Server' and 'Add tiles' with a plus sign icon.

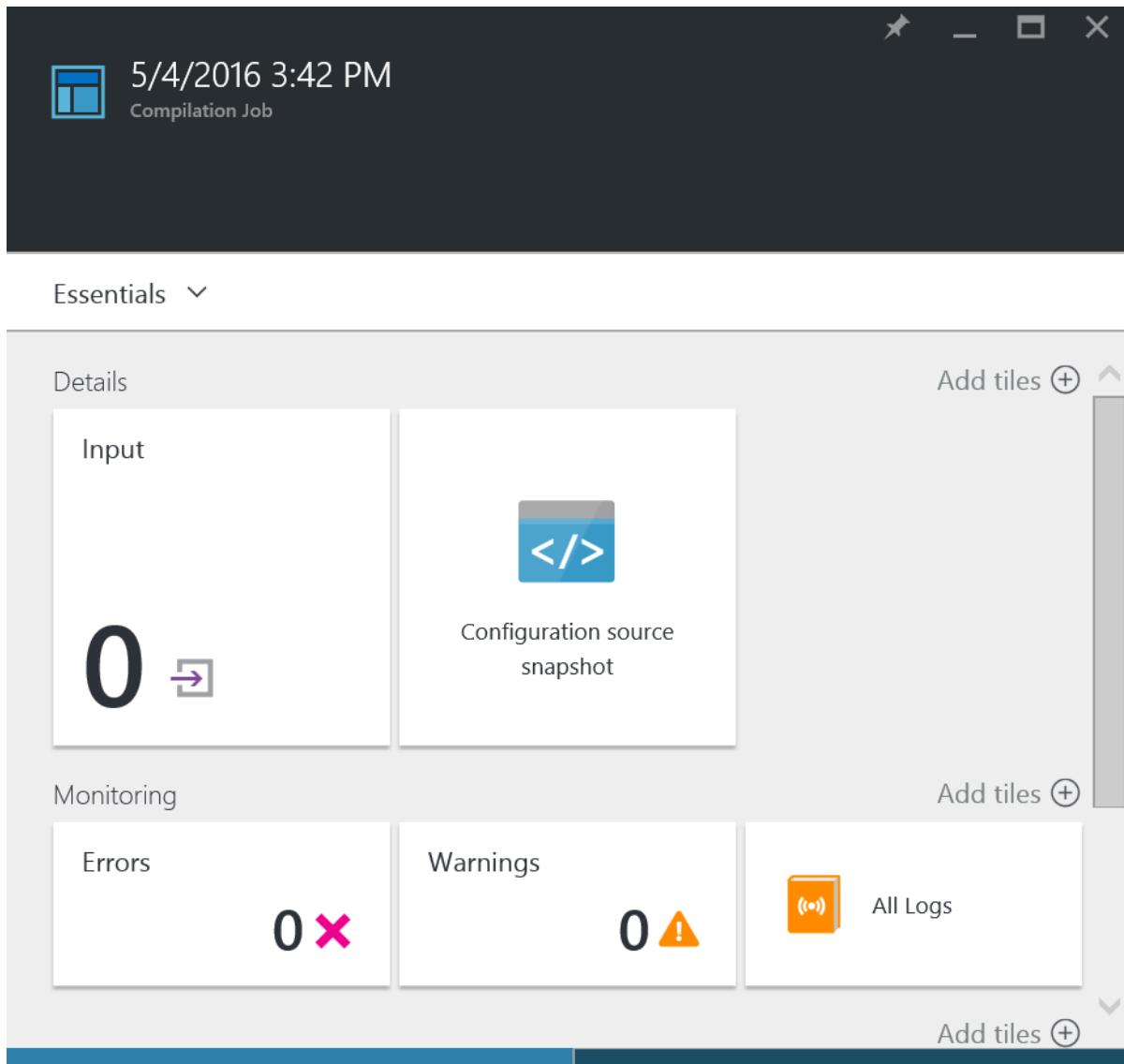
NOTE

When you compile a configuration in Azure Automation, it automatically deploys any created node configuration MOFs to the pull server.

Viewing a compilation job

After you start a compilation, you can view it in the **Compilation jobs** tile in the **Configuration** blade. The **Compilation jobs** tile shows currently running, completed, and failed jobs. When you open a compilation job blade, it shows information about that job including any errors or warnings encountered, input parameters used in the configuration, and compilation logs.

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Configurations**.
4. On the **DSC Configurations** blade, click **TestConfig** (the name of the previously imported configuration).
5. On the **Compilation jobs** tile of the **TestConfig Configuration** blade, click on any of the jobs listed. A **Compilation Job** blade opens, labeled with the date that the compilation job was started.



6. Click on any tile in the **Compilation Job** blade to see further details about the job.

Viewing node configurations

Successful completion of a compilation job creates one or more new node configurations. A node configuration is a MOF document that is deployed to the pull server and ready to be pulled and applied by one or more nodes. You can view the node configurations in your Automation account in the **DSC Node Configurations** blade. A node configuration has a name with the form *ConfigurationName.NodeName*.

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Node Configurations**.

NAME	CREATED	LAST MODIFIED
TestConfig.NotWebServer	5/10/2016 2:57 PM	5/10/2016 2:57 PM
TestConfig.WebServer	5/10/2016 2:57 PM	5/10/2016 2:57 PM

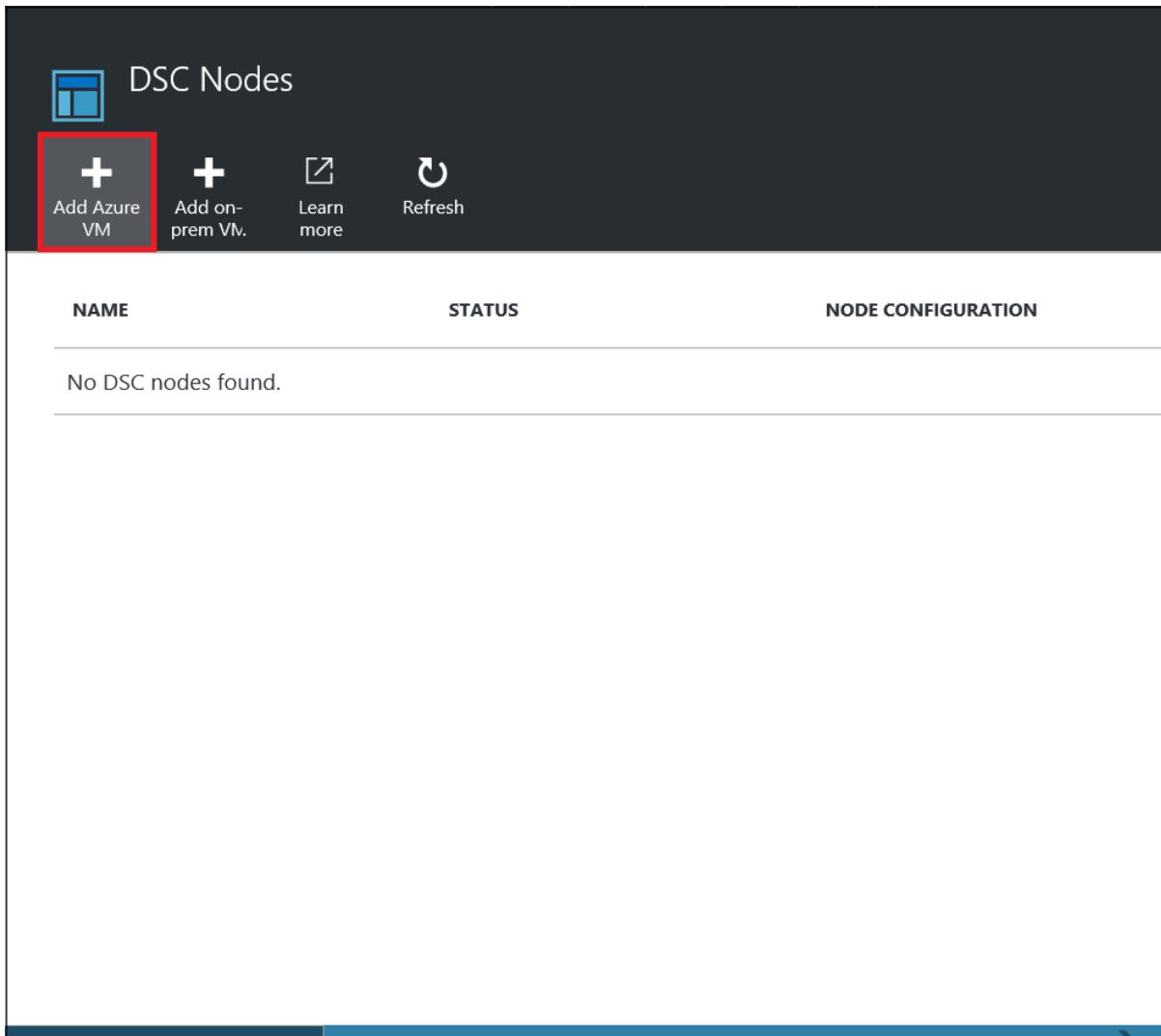
Onboarding an Azure VM for management with Azure Automation DSC

You can use Azure Automation DSC to manage Azure VMs (both Classic and Resource Manager), on-premises VMs, Linux machines, AWS VMs, and on-premises physical machines. In this topic, we cover how to onboard only Azure Resource Manager VMs. For information about onboarding other types of machines, see [Onboarding machines for Azure Automation](#)

management by Azure Automation DSC.

To onboard an Azure Resource Manager VM for management by Azure Automation DSC

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Nodes**.
4. In the **DSC Nodes** blade, click **Add Azure VM**.



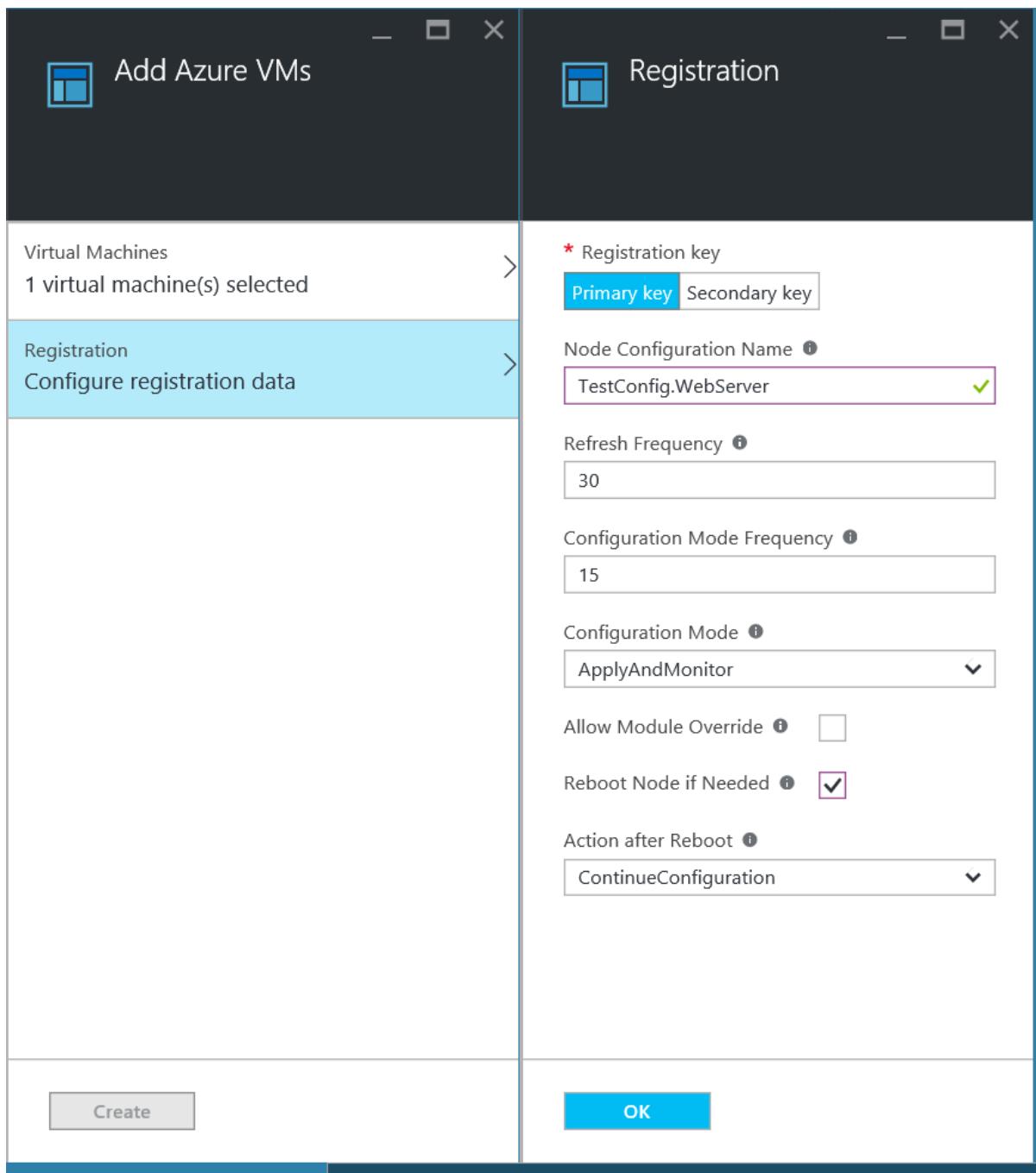
The screenshot shows the 'DSC Nodes' blade in the Azure portal. At the top, there are four buttons: 'Add Azure VM' (highlighted with a red box), 'Add on-prem VN.', 'Learn more', and 'Refresh'. Below the buttons is a table with three columns: 'NAME', 'STATUS', and 'NODE CONFIGURATION'. A message 'No DSC nodes found.' is displayed in the table body. The bottom of the blade has a dark blue footer bar.

5. In the **Add Azure VMs** blade, click **Select virtual machines to onboard**.
6. In the **Select VMs** blade, select the VM you want to onboard, and click **OK**.

IMPORTANT

This must be an Azure Resource Manager VM running Windows Server 2008 R2 or later.

7. In the **Add Azure VMs** blade, click **Configure registration data**.
8. In the **Registration** blade, enter the name of the node configuration you want to apply to the VM in the **Node Configuration Name** box. This must exactly match the name of a node configuration in the Automation account. Providing a name at this point is optional. You can change the assigned node configuration after onboarding the node. Check **Reboot Node if Needed**, and then click **OK**.



The node configuration you specified will be applied to the VM at intervals specified by the **Configuration Mode Frequency**, and the VM will check for updates to the node configuration at intervals specified by the **Refresh Frequency**. For more information about how these values are used, see [Configuring the Local Configuration Manager](#).

9. In the **Add Azure VMs** blade, click **Create**.

Azure will start the process of onboarding the VM. When it is complete, the VM will show up in the **DSC Nodes** blade in the Automation account.

Viewing the list of DSC nodes

You can view the list of all machines that have been onboarded for management in your Automation account in the **DSC Nodes** blade.

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Nodes**.

Viewing reports for DSC nodes

Each time Azure Automation DSC performs a consistency check on a managed node, the node sends a status report back to the pull server. You can view these reports on the blade for that node.

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Nodes**.
4. On the **Reports** tile, click on any of the reports in the list.

The screenshot shows the 'Report details' section of the Azure Automation DSC Node Report blade. It includes fields for Report ID (dd92354b-131b-11e6-80c2-000d3a00e59), Report status (Compliant), Report time (5/5/2016 4:48 PM), Start time (5/5/2016 4:48 PM), Total runtime (1 seconds), Type (Consistency), and Resources (WindowsFeature, Compliant). Below this, the 'Node state at report time' section displays the Node name (DSCNODE1), IP address (10.0.0.6), Configuration mode (Apply and monitor), and a status message indicating 'All resources are healthy'.

Resource Type	Status
WindowsFeature	Compliant

Node state at report time

Node State	Value
Node name	DSCNODE1
IP address	10.0.0.6
Configuration mode	Apply and monitor

On the blade for an individual report, you can see the following status information for the corresponding

consistency check:

- The report status — whether the node is "Compliant", the configuration "Failed", or the node is "Not Compliant" (when the node is in **applyandmonitor** mode and the machine is not in the desired state).
- The start time for the consistency check.
- The total runtime for the consistency check.
- The type of consistency check.
- Any errors, including the error code and error message.
- Any DSC resources used in the configuration, and the state of each resource (whether the node is in the desired state for that resource) — you can click on each resource to get more detailed information for that resource.
- The name, IP address, and configuration mode of the node.

You can also click **View raw report** to see the actual data that the node sends to the server. For more information about using that data, see [Using a DSC report server](#).

It can take some time after a node is onboarded before the first report is available. You might need to wait up to 30 minutes for the first report after you onboard a node.

Reassigning a node to a different node configuration

You can assign a node to use a different node configuration than the one you initially assigned.

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Nodes**.
4. On the **DSC Nodes** blade, click on the name of the node you want to reassign.
5. On the blade for that node, click **Assign node**.

DSCNODE1

Assign node ... Unregister

Essentials ^

Resource group AADsc1	IP address 10.0.0.6
Id <Node Id>	Account AADsc1
Last seen time 5/10/2016 5:02 PM	Virtual machine DSCNODE1
Configuration TestConfig	Node configuration TestConfig.WebServer
Registration time 5/10/2016 5:02 PM	Status Compliant

Reports

TYPE	STATUS	REPORT TIME
No reports found for this dsc node		

6. On the **Assign Node Configuration** blade, select the node configuration to which you want to assign the node, and then click **OK**.

The screenshot shows a Windows application window titled "Assign Node Configuration" for "DSCNODE1". The window contains a message box with an information icon and text about changing node configuration. Below is a table listing configurations with columns "NAME" and "LAST MODIFIED". At the bottom is an "OK" button.

NAME	LAST MODIFIED
TestConfig.NotWebServer	5/10/2016 2:57 PM
TestConfig.WebServer	5/10/2016 2:57 PM

OK

Unregistering a node

If you no longer want a node to be managed by Azure Automation DSC, you can unregister it.

1. Sign in to the [Azure portal](#).
2. On the Hub menu, click **All resources** and then the name of your Automation account.
3. On the **Automation account** blade, click **DSC Nodes**.
4. On the **DSC Nodes** blade, click on the name of the node you want to unregister.
5. On the blade for that node, click **Unregister**.

DSCNODE1

Assign node ... **Unregister**

Essentials ^

Resource group	IP address
AADsc1	10.0.0.6
Id	Account
<Node Id>	AADsc1
Last seen time	Virtual machine
5/10/2016 5:08 PM	DSCNODE1
Configuration	Node configuration
TestConfig	TestConfig.NotWebServer
Registration time	Status
5/10/2016 5:02 PM	Pending

Reports

TYPE	STATUS	REPORT TIME
Consistency	✓ Compliant	5/10/2016 5:08 PM
Consistency	✓ Compliant	5/10/2016 5:08 PM
Consistency	✓ Compliant	5/10/2016 4:52 PM
Consistency	✓ Compliant	5/10/2016 4:37 PM
Consistency	✓ Compliant	5/10/2016 4:37 PM

Related Articles

- [Azure Automation DSC overview](#)
- [Onboarding machines for management by Azure Automation DSC](#)
- [Windows PowerShell Desired State Configuration Overview](#)
- [Azure Automation DSC cmdlets](#)
- [Azure Automation DSC pricing](#)

Onboarding machines for management by Azure Automation DSC

2/15/2017 • 13 min to read • [Edit on GitHub](#)

Why manage machines with Azure Automation DSC?

Like [PowerShell Desired State Configuration](#), Azure Automation Desired State Configuration is a simple, yet powerful, configuration management service for DSC nodes (physical and virtual machines) in any cloud or on-premises datacenter. It enables scalability across thousands of machines quickly and easily from a central, secure location. You can easily onboard machines, assign them declarative configurations, and view reports showing each machine's compliance to the desired state you specified. The Azure Automation DSC management layer is to DSC what the Azure Automation management layer is to PowerShell scripting. In other words, in the same way that Azure Automation helps you manage PowerShell scripts, it also helps you manage DSC configurations. To learn more about the benefits of using Azure Automation DSC, see [Azure Automation DSC overview](#).

Azure Automation DSC can be used to manage a variety of machines:

- Azure virtual machines (classic)
- Azure virtual machines
- Amazon Web Services (AWS) virtual machines
- Physical/virtual Windows machines on-premises, or in a cloud other than Azure/AWS
- Physical/virtual Linux machines on-premises, in Azure, or in a cloud other than Azure

In addition, if you are not ready to manage machine configuration from the cloud, Azure Automation DSC can also be used as a report-only endpoint. This allows you to set (push) desired configuration through DSC on-premises and view rich reporting details on node compliance with the desired state in Azure Automation.

The following sections outline how you can onboard each type of machine to Azure Automation DSC.

Azure virtual machines (classic)

With Azure Automation DSC, you can easily onboard Azure virtual machines (classic) for configuration management using either the Azure portal, or PowerShell. Under the hood, and without an administrator having to remote into the VM, the Azure VM Desired State Configuration extension registers the VM with Azure Automation DSC. Since the Azure VM Desired State Configuration extension runs asynchronously, steps to track its progress or troubleshoot it are provided in the [Troubleshooting Azure virtual machine onboarding](#) section below.

Azure portal

In the [Azure portal](#), click **Browse** -> **Virtual machines (classic)**. Select the Windows VM you want to onboard. On the virtual machine's dashboard blade, click **All settings** -> **Extensions** -> **Add** -> **Azure Automation DSC** -> **Create**. Enter the [PowerShell DSC Local Configuration Manager values](#) required for your use case, your Automation account's registration key and registration URL, and optionally a node configuration to assign to the VM.

The screenshot shows two windows side-by-side. On the left is the 'Azure Automation DSC' extension page from the Microsoft Store, featuring a gear icon with a lightning bolt, the title 'Azure Automation DSC', the publisher 'Microsoft Corp.', a brief description, social sharing links, and a 'Create' button. On the right is the 'Add Extension' configuration dialog, which includes fields for 'Registration URL' and 'Registration Key', and dropdowns for 'Node Configuration Name', 'Refresh Frequency', 'Configuration Mode Frequency', 'Configuration Mode', and 'Action after Reboot'. An 'OK' button is at the bottom right of the dialog.

Azure Automation DSC brings the same management layer to PowerShell Desired State Configuration as Azure Automation offers for PowerShell Workflows today.

Azure Automation DSC allows you to author and manage PowerShell Desired State Configurations, import DSC Resources, and generate DSC Node Configurations (MOF documents), all in the cloud. These DSC items will be placed on the Azure Automation DSC pull server so that target nodes in the cloud or on-premises can pick them up, automatically conform to the desired state they specify, and report back on their compliance with the desired state to Azure Automation.

Adding the Azure Automation DSC extension to your Azure VM will automatically onboard the VM for configuration management using Azure Automation DSC.

PUBLISHER Microsoft Corp.

USEFUL LINKS [Learn more about Azure Automation DSC](#) [Learn more about PowerShell DSC](#)

Create

* Registration URL

* Registration Key

Node Configuration Name

Refresh Frequency 30

Configuration Mode Frequency 15

Configuration Mode ApplyAndMonitor

Action after Reboot ContinueConfiguration

OK

To find the registration URL and key for the Automation account to onboard the machine to, see the [Secure registration](#) section below.

PowerShell

```

# log in to both Azure Service Management and Azure Resource Manager
Add-AzureAccount
Add-AzureRmAccount

# fill in correct values for your VM/Automation account here
$VMName = ""
$ServiceName = ""
$AutomationAccountName = ""
$AutomationAccountResourceGroup = ""

# fill in the name of a Node Configuration in Azure Automation DSC, for this VM to conform to
$NodeConfigName = ""

# get Azure Automation DSC registration info
$Account = Get-AzureRmAutomationAccount -ResourceGroupName $AutomationAccountResourceGroup -Name
$AutomationAccountName
$RegistrationInfo = $Account | Get-AzureRmAutomationRegistrationInfo

# use the DSC extension to onboard the VM for management with Azure Automation DSC
$VM = Get-AzureVM -Name $VMName -ServiceName $ServiceName

$PublicConfiguration = ConvertTo-Json -Depth 8 @{
    SasToken = ""
    ModulesUrl =
    "https://eus2oaasibizamarketprod1.blob.core.windows.net/automationdscpreview/RegistrationMetaConfigV2.zip"
    ConfigurationFunction = "RegistrationMetaConfigV2.ps1\RegistrationMetaConfigV2"
}

# update these PowerShell DSC Local Configuration Manager defaults if they do not match your use case.
# See https://technet.microsoft.com/library/dn249922.aspx?f=255&MSPPError=-2147217396 for more details
Properties = @{
    RegistrationKey = @{
        UserName = 'notused'
        Password = 'PrivateSettingsRef:RegistrationKey'
    }
    RegistrationUrl = $RegistrationInfo.Endpoint
    NodeConfigurationName = $NodeConfigName
    ConfigurationMode = "ApplyAndMonitor"
    ConfigurationModeFrequencyMins = 15
    RefreshFrequencyMins = 30
    RebootNodeIfNeeded = $False
    ActionAfterReboot = "ContinueConfiguration"
    AllowModuleOverwrite = $False
}
}

$PrivateConfiguration = ConvertTo-Json -Depth 8 @{
    Items = @{
        RegistrationKey = $RegistrationInfo.PrimaryKey
    }
}

$VM = Set-AzureVMExtension `

-VM $vm `

-Publisher Microsoft.Powershell `

-ExtensionName DSC `

-Version 2.19 `

-PublicConfiguration $PublicConfiguration `

-PrivateConfiguration $PrivateConfiguration `

-ForceUpdate

$VM | Update-AzureVM

```

Azure virtual machines

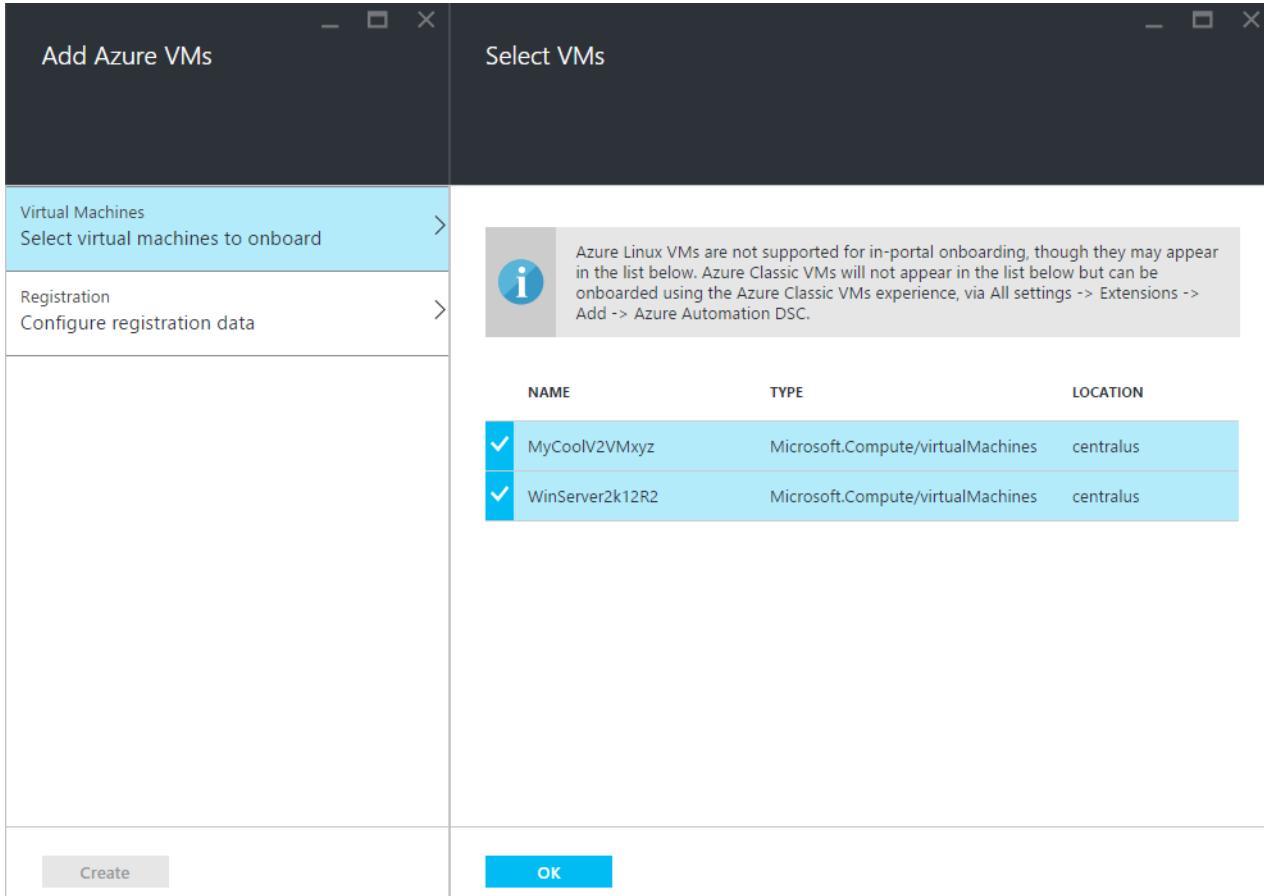
Azure Automation DSC lets you easily onboard Azure virtual machines for configuration management, using

either the Azure portal, Azure Resource Manager templates, or PowerShell. Under the hood, and without an administrator having to remote into the VM, the Azure VM Desired State Configuration extension registers the VM with Azure Automation DSC. Since the Azure VM Desired State Configuration extension runs asynchronously, steps to track its progress or troubleshoot it are provided in the [Troubleshooting Azure virtual machine onboarding](#) section below.

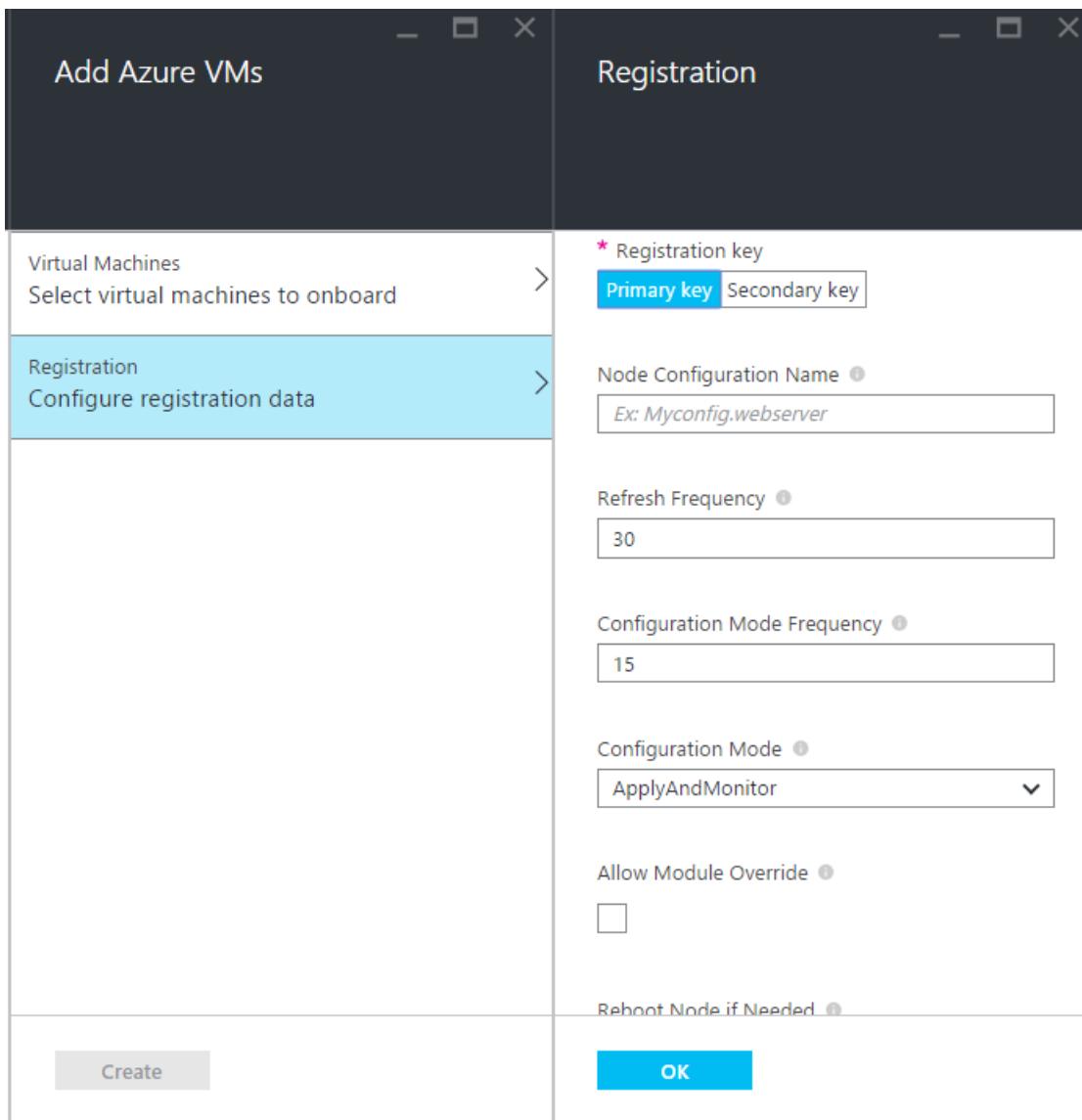
Azure portal

In the [Azure portal](#), navigate to the Azure Automation account where you want to onboard virtual machines. On the Automation account dashboard, click **DSC Nodes** -> **Add Azure VM**.

Under **Select virtual machines to onboard**, select one or more Azure virtual machines to onboard.



Under **Configure registration data**, enter the [PowerShell DSC Local Configuration Manager values](#) required for your use case, and optionally a node configuration to assign to the VM.



Azure Resource Manager templates

Azure virtual machines can be deployed and onboarded to Azure Automation DSC via Azure Resource Manager templates. See [Configure a VM via DSC extension and Azure Automation DSC](#) for an example template that onboards an existing VM to Azure Automation DSC. To find the registration key and registration URL taken as input in this template, see the [Secure registration](#) section below.

PowerShell

The `Register-AzureRmAutomationDscNode` cmdlet can be used to onboard virtual machines in the Azure portal via PowerShell.

Amazon Web Services (AWS) virtual machines

You can easily onboard Amazon Web Services virtual machines for configuration management by Azure Automation DSC using the AWS DSC Toolkit. You can learn more about the toolkit [here](#).

Physical/virtual Windows machines on-premises, or in a cloud other than Azure/AWS

On-premises Windows machines and Windows machines in non-Azure clouds (such as Amazon Web Services) can also be onboarded to Azure Automation DSC, as long as they have outbound access to the internet, via a few simple steps:

1. Make sure the latest version of [WMF 5](#) is installed on the machines you want to onboard to Azure Automation

DSC.

2. Follow the directions in section [Generating DSC metaconfigurations](#) below to generate a folder containing the needed DSC metaconfigurations.
3. Remotely apply the PowerShell DSC metaconfiguration to the machines you want to onboard. **The machine this command is run from must have the latest version of WMF 5 installed:**

```
Set-DscLocalConfigurationManager -Path C:\Users\joe\Desktop\DscMetaConfigs -ComputerName MyServer1, MyServer2
```

4. If you cannot apply the PowerShell DSC metaconfigurations remotely, copy the metaconfigurations folder from step 2 onto each machine to onboard. Then call **Set-DscLocalConfigurationManager** locally on each machine to onboard.
5. Using the Azure portal or cmdlets, check that the machines to onboard now show up as DSC nodes registered in your Azure Automation account.

Physical/virtual Linux machines on-premises, in Azure, or in a cloud other than Azure

On-premises Linux machines, Linux machines in Azure, and Linux machines in non-Azure clouds can also be onboarded to Azure Automation DSC, as long as they have outbound access to the internet, via a few simple steps:

1. Make sure the latest version of the [DSC Linux agent](#) is installed on the machines you want to onboard to Azure Automation DSC.
2. If the [PowerShell DSC Local Configuration Manager defaults](#) match your use case, and you want to onboard machines such that they **both** pull from and report to Azure Automation DSC:
 - On each Linux machine to onboard to Azure Automation DSC, use Register.py to onboard using the PowerShell DSC Local Configuration Manager defaults:

```
/opt/microsoft/dsc/Scripts/Register.py <Automation account registration key> <Automation account registration URL>
```

- To find the registration key and registration URL for your Automation account, see the [Secure registration](#) section below.

If the PowerShell DSC Local Configuration Manager defaults **do not** match your use case, or you want to onboard machines such that they only report to Azure Automation DSC, but do not pull configuration or PowerShell modules from it, follow steps 3 - 6. Otherwise, proceed directly to step 6.

3. Follow the directions in the [Generating DSC metaconfigurations](#) section below to generate a folder containing the needed DSC metaconfigurations.
4. Remotely apply the PowerShell DSC metaconfiguration to the machines you want to onboard:

```
$SecurePass = ConvertTo-SecureString -String "<root password>" -AsPlainText -Force
$Cred = New-Object System.Management.Automation.PSCredential "root", $SecurePass
$Opt = New-CimSessionOption -UseSsl -SkipCACheck -SkipCNCheck -SkipRevocationCheck

# need a CimSession for each Linux machine to onboard

$Session = New-CimSession -Credential $Cred -ComputerName <your Linux machine> -Port 5986 -
Authentication basic -SessionOption $Opt

Set-DscLocalConfigurationManager -CimSession $Session -Path C:\Users\joe\Desktop\DscMetaConfigs
```

The machine this command is run from must have the latest version of [WMF 5](#) installed.

1. If you cannot apply the PowerShell DSC metaconfigurations remotely, for each Linux machine to onboard, copy the metaconfiguration corresponding to that machine from the folder in step 5 onto the Linux machine. Then call `SetDscLocalConfigurationManager.py` locally on each Linux machine you want to onboard to Azure Automation DSC:

```
/opt/microsoft/dsc/Scripts/SetDscLocalConfigurationManager.py -configurationmof <path to metaconfiguration file>
```

2. Using the Azure portal or cmdlets, check that the machines to onboard now show up as DSC nodes registered in your Azure Automation account.

Generating DSC metaconfigurations

To generically onboard any machine to Azure Automation DSC, a [DSC metaconfiguration](#) can be generated that, when applied, tells the DSC agent on the machine to pull from and/or report to Azure Automation DSC. DSC metaconfigurations for Azure Automation DSC can be generated using either a PowerShell DSC configuration, or the Azure Automation PowerShell cmdlets.

NOTE

DSC metaconfigurations contain the secrets needed to onboard a machine to an Automation account for management. Make sure to properly protect any DSC metaconfigurations you create, or delete them after use.

Using a DSC Configuration

1. Open the PowerShell ISE as an administrator in a machine in your local environment. The machine must have the latest version of [WMF 5](#) installed.
2. Copy the following script locally. This script contains a PowerShell DSC configuration for creating metaconfigurations, and a command to kick off the metaconfiguration creation.

```
# The DSC configuration that will generate metaconfigurations
[DscLocalConfigurationManager()]
Configuration DscMetaConfigs
{
    param
    (
        [Parameter(Mandatory=$True)]
        [String]$RegistrationUrl,
        [Parameter(Mandatory=$True)]
        [String]$RegistrationKey,
        [Parameter(Mandatory=$True)]
        [String[]]$ComputerName,
        [Int]$RefreshFrequencyMins = 30,
        [Int]$ConfigurationModeFrequencyMins = 15,
        [String]$ConfigurationMode = "ApplyAndMonitor",
        [String]$NodeConfigurationName,
        [Boolean]$RebootNodeIfNeeded= $False,
        [String]$ActionAfterReboot = "ContinueConfiguration",
        [Boolean]$AllowModuleOverwrite = $False,
    )
}
```

```

    [Boolean]$ReportOnly
)

if(!$NodeConfigurationName -or $NodeConfigurationName -eq "")
{
    $ConfigurationNames = $null
}
else
{
    $ConfigurationNames = @($NodeConfigurationName)
}

if($ReportOnly)
{
$RefreshMode = "PUSH"
}
else
{
$RefreshMode = "PULL"
}

Node $ComputerName
{

Settings
{
    RefreshFrequencyMins = $RefreshFrequencyMins
    RefreshMode = $RefreshMode
    ConfigurationMode = $ConfigurationMode
    AllowModuleOverwrite = $AllowModuleOverwrite
    RebootNodeIfNeeded = $RebootNodeIfNeeded
    ActionAfterReboot = $ActionAfterReboot
    ConfigurationModeFrequencyMins = $ConfigurationModeFrequencyMins
}

if(!$ReportOnly)
{
ConfigurationRepositoryWeb AzureAutomationDSC
{
    ServerUrl = $RegistrationUrl
    RegistrationKey = $RegistrationKey
    ConfigurationNames = $ConfigurationNames
}

ResourceRepositoryWeb AzureAutomationDSC
{
    ServerUrl = $RegistrationUrl
    RegistrationKey = $RegistrationKey
}
}

ReportServerWeb AzureAutomationDSC
{
    ServerUrl = $RegistrationUrl
    RegistrationKey = $RegistrationKey
}
}

# Create the metaconfigurations
# TODO: edit the below as needed for your use case
$Params = @{
    RegistrationUrl = '<fill me in>';
    RegistrationKey = '<fill me in>';
    ComputerName = @('<some VM to onboard>', '<some other VM to onboard>');
    NodeConfigurationName = 'SimpleConfig.webserver';
    RefreshFrequencyMins = 30;
    ConfigurationModeFrequencyMins = 15;
    RebootNodeIfNeeded = $false;
}

```

```

    RebootOnNodeITNeedsMe = $praise;
    AllowModuleOverwrite = $False;
    ConfigurationMode = 'ApplyAndMonitor';
    ActionAfterReboot = 'ContinueConfiguration';
    ReportOnly = $False; # Set to $True to have machines only report to AA DSC but not pull from it
}

# Use PowerShell splatting to pass parameters to the DSC configuration being invoked
# For more info about splatting, run: Get-Help -Name about_Splatting
DscMetaConfigs @Params

```

3. Fill in the registration key and URL for your Automation account, as well as the names of the machines to onboard. All other parameters are optional. To find the registration key and registration URL for your Automation account, see the [Secure registration](#) section below.
4. If you want the machines to report DSC status information to Azure Automation DSC, but not pull configuration or PowerShell modules, set the **ReportOnly** parameter to true.
5. Run the script. You should now have a folder called **DscMetaConfigs** in your working directory, containing the PowerShell DSC metaconfigurations for the machines to onboard (as an administrator):

```
Set-DscLocalConfigurationManager -Path ./DscMetaConfigs
```

Using the Azure Automation cmdlets

If the PowerShell DSC Local Configuration Manager defaults match your use case, and you want to onboard machines such that they both pull from and report to Azure Automation DSC, the Azure Automation cmdlets provide a simplified method of generating the DSC metaconfigurations needed:

1. Open the PowerShell console or PowerShell ISE as an administrator in a machine in your local environment.
2. Connect to Azure Resource Manager using [Add-AzureRmAccount](#)
3. Download the PowerShell DSC metaconfigurations for the machines you want to onboard from the Automation account to which you want to onboard nodes:

```

# Define the parameters for Get-AzureRmAutomationDscOnboardingMetaconfig using PowerShell Splatting
$Params = @{

    ResourceGroupName = 'ContosoResources'; # The name of the ARM Resource Group that contains your
    AutomationAccountName = 'ContosoAutomation'; # The name of the Azure Automation Account where you
    want a node on-boarded to
    ComputerName = @('web01', 'web02', 'sql01'); # The names of the computers that the meta
    configuration will be generated for
    OutputFolder = "$env:UserProfile\Desktop\";

}
# Use PowerShell splatting to pass parameters to the Azure Automation cmdlet being invoked
# For more info about splatting, run: Get-Help -Name about_Splatting
Get-AzureRmAutomationDscOnboardingMetaconfig @Params

```

4. You should now have a folder called **DscMetaConfigs**, containing the PowerShell DSC metaconfigurations for the machines to onboard (as an administrator):

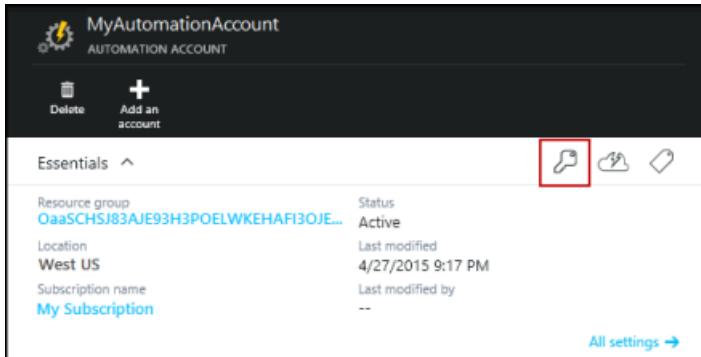
```
Set-DscLocalConfigurationManager -Path $env:UserProfile\Desktop\DscMetaConfigs
```

Secure registration

Machines can securely onboard to an Azure Automation account through the WMF 5 DSC registration protocol, which allows a DSC node to authenticate to a PowerShell DSC V2 Pull or Reporting server (including Azure

Automation DSC). The node registers to the server at a **Registration URL**, authenticating using a **Registration key**. During registration, the DSC node and DSC Pull/Reporting server negotiate a unique certificate for this node to use for authentication to the server post-registration. This process prevents onboarded nodes from impersonating one another, such as if a node is compromised and behaving maliciously. After registration, the Registration key is not used for authentication again, and is deleted from the node.

You can get the information required for the DSC registration protocol from the **Manage Keys** blade in the Azure preview portal. Open this blade by clicking the key icon on the **Essentials** panel for the Automation account.



- Registration URL is the URL field in the Manage Keys blade.
- Registration key is the Primary Access Key or Secondary Access Key in the Manage Keys blade. Either key can be used.

For added security, the primary and secondary access keys of an Automation account can be regenerated at any time (on the **Manage Keys** blade) to prevent future node registrations using previous keys.

Troubleshooting Azure virtual machine onboarding

Azure Automation DSC lets you easily onboard Azure Windows VMs for configuration management. Under the hood, the Azure VM Desired State Configuration extension is used to register the VM with Azure Automation DSC. Since the Azure VM Desired State Configuration extension runs asynchronously, tracking its progress and troubleshooting its execution may be important.

NOTE

Any method of onboarding an Azure Windows VM to Azure Automation DSC that uses the Azure VM Desired State Configuration extension could take up to an hour for the node to show up as registered in Azure Automation. This is due to the installation of Windows Management Framework 5.0 on the VM by the Azure VM DSC extension, which is required to onboard the VM to Azure Automation DSC.

To troubleshoot or view the status of the Azure VM Desired State Configuration extension, in the Azure portal navigate to the VM being onboarded, then click -> **All settings** -> **Extensions** -> **DSC**. For more details, you can click **View detailed status**.

PUBLISHER	NAME	VERSION	STATUS
Microsoft.Compute	BGInfo	1.2.2	Ready
Microsoft.PowerShell	DSC	1.10.1.0	Success

PUBLISHER
Microsoft.PowerShell
VERSION
1.10.1.0
STATUS
Success
MESSAGE
DSC configuration was applied successfully.
[View detailed status](#)

Certificate expiration and reregistration

After registering a machine as a DSC node in Azure Automation DSC, there are a number of reasons why you may need to reregister that node in the future:

- After registering, each node automatically negotiates a unique certificate for authentication that expires after one year. Currently, the PowerShell DSC registration protocol cannot automatically renew certificates when they are nearing expiration, so you need to reregister the nodes after a year's time. Before reregistering, ensure that each node is running Windows Management Framework 5.0 RTM. If a node's authentication certificate expires, and the node is not reregistered, the node will be unable to communicate with Azure Automation and will be marked 'Unresponsive.' Reregistration performed 90 days or less from the certificate expiration time, or at any point after the certificate expiration time, will result in a new certificate being generated and used.
- To change any [PowerShell DSC Local Configuration Manager values](#) that were set during initial registration of the node, such as ConfigurationMode. Currently, these DSC agent values can only be changed through reregistration. The one exception is the Node Configuration assigned to the node -- this can be changed in Azure Automation DSC directly.

Reregistration can be performed in the same way you registered the node initially, using any of the onboarding methods described in this document. You do not need to unregister a node from Azure Automation DSC before reregistering it.

Related Articles

- [Azure Automation DSC overview](#)
- [Azure Automation DSC cmdlets](#)
- [Azure Automation DSC pricing](#)

Compiling configurations in Azure Automation DSC

2/21/2017 • 6 min to read • [Edit on GitHub](#)

You can compile Desired State Configuration (DSC) configurations in two ways with Azure Automation: in the Azure portal, and with Windows PowerShell. The following table will help you determine when to use which method based on the characteristics of each:

Azure portal

- Simplest method with interactive user interface
- Form to provide simple parameter values
- Easily track job state
- Access authenticated with Azure logon

Windows PowerShell

- Call from command line with Windows PowerShell cmdlets
- Can be included in automated solution with multiple steps
- Provide simple and complex parameter values
- Track job state
- Client required to support PowerShell cmdlets
- Pass ConfigurationData
- Compile configurations that use credentials

Once you have decided on a compilation method, you can follow the respective procedures below to start compiling.

Compiling a DSC Configuration with the Azure portal

1. From your Automation account, click **DSC Configurations**.
2. Click a configuration to open its blade.
3. Click **Compile**.
4. If the configuration has no parameters, you will be prompted to confirm whether you want to compile it. If the configuration has parameters, the **Compile Configuration** blade will open so you can provide parameter values. See the **Basic Parameters** section below for further details on parameters.
5. The **Compilation Job** blade is opened so that you can track the compilation job's status, and the node configurations (MOF configuration documents) it caused to be placed on the Azure Automation DSC Pull Server.

Compiling a DSC Configuration with Windows PowerShell

You can use `Start-AzureRmAutomationDscCompilationJob` to start compiling with Windows PowerShell. The following sample code starts compilation of a DSC configuration called **SampleConfig**.

```
Start-AzureRmAutomationDscCompilationJob -ResourceGroupName "MyResourceGroup" -AutomationAccountName "MyAutomationAccount" -ConfigurationName "SampleConfig"
```

`Start-AzureRmAutomationDscCompilationJob` returns a compilation job object that you can use to track its status. You can then use this compilation job object with `Get-AzureRmAutomationDscCompilationJob` to determine the status of the compilation job, and `Get-AzureRmAutomationDscCompilationJobOutput` to view its streams (output). The following

sample code starts compilation of the **SampleConfig** configuration, waits until it has completed, and then displays its streams.

```
$CompilationJob = Start-AzureRmAutomationDscCompilationJob -ResourceGroupName "MyResourceGroup" -  
AutomationAccountName "MyAutomationAccount" -ConfigurationName "SampleConfig"  
  
while($CompilationJob.EndTime -eq $null -and $CompilationJob.Exception -eq $null)  
{  
    $CompilationJob = $CompilationJob | Get-AzureRmAutomationDscCompilationJob  
    Start-Sleep -Seconds 3  
}  
  
$CompilationJob | Get-AzureRmAutomationDscCompilationJobOutput -Stream Any
```

Basic Parameters

Parameter declaration in DSC configurations, including parameter types and properties, works the same as in Azure Automation runbooks. See [Starting a runbook in Azure Automation](#) to learn more about runbook parameters.

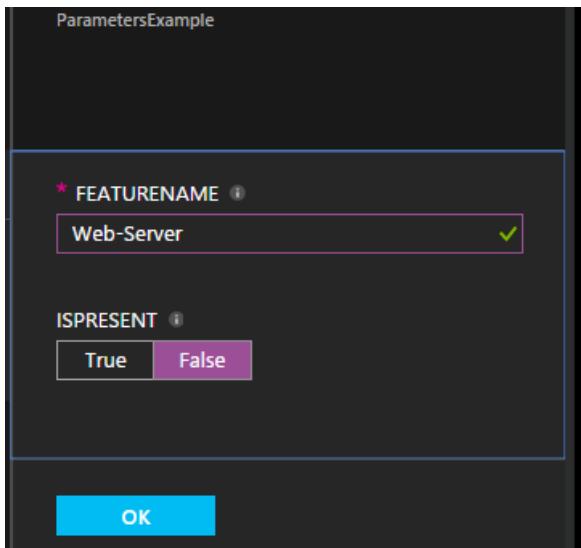
The following example uses two parameters called **FeatureName** and **IsPresent**, to determine the values of properties in the **ParametersExample.sample** node configuration, generated during compilation.

```
Configuration ParametersExample  
{  
    param(  
        [Parameter(Mandatory=$true)]  
  
        [string] $FeatureName,  
  
        [Parameter(Mandatory=$true)]  
        [boolean] $IsPresent  
    )  
  
    $EnsureString = "Present"  
    if($IsPresent -eq $false)  
    {  
        $EnsureString = "Absent"  
    }  
  
    Node "sample"  
    {  
        WindowsFeature ($FeatureName + "Feature")  
        {  
            Ensure = $EnsureString  
            Name = $FeatureName  
        }  
    }  
}
```

You can compile DSC Configurations that use basic parameters in the Azure Automation DSC portal, or with Azure PowerShell:

Portal

In the portal, you can enter parameter values after clicking **Compile**.



PowerShell

PowerShell requires parameters in a [hashtable](#) where the key matches the parameter name, and the value equals the parameter value.

```
$Parameters = @{
    "FeatureName" = "Web-Server"
    "IsPresent" = $False
}

Start-AzureRmAutomationDscCompilationJob -ResourceGroupName "MyResourceGroup" -AutomationAccountName
"MyAutomationAccount" -ConfigurationName "ParametersExample" -Parameters $Parameters
```

For information about passing PSCredentials as parameters, see [Credential Assets](#) below.

ConfigurationData

ConfigurationData allows you to separate structural configuration from any environment specific configuration while using PowerShell DSC. See [Separating "What" from "Where" in PowerShell DSC](#) to learn more about **ConfigurationData**.

NOTE

You can use **ConfigurationData** when compiling in Azure Automation DSC using Azure PowerShell, but not in the Azure portal.

The following example DSC configuration uses **ConfigurationData** via the **\$ConfigurationData** and **\$AllNodes** keywords. You'll also need the [xWebAdministration](#) module for this example:

```

Configuration ConfigurationDataSample
{
    Import-DscResource -ModuleName xWebAdministration -Name MSFT_xWebsite

    Write-Verbose $ConfigurationData.NonNodeData.SomeMessage

    Node $AllNodes.Where{$_.Role -eq "WebServer"}.NodeName
    {
        xWebsite Site
        {
            Name = $Node.SiteName
            PhysicalPath = $Node.SiteContents
            Ensure = "Present"
        }
    }
}

```

You can compile the DSC configuration above with PowerShell. The below PowerShell adds two node configurations to the Azure Automation DSC Pull Server: **ConfigurationDataSample.MyVM1** and **ConfigurationDataSample.MyVM3**:

```

$ConfigData = @{
    AllNodes = @(
        @{
            NodeName = "MyVM1"
            Role = "WebServer"
        },
        @{
            NodeName = "MyVM2"
            Role = "SQLServer"
        },
        @{
            NodeName = "MyVM3"
            Role = "WebServer"
        }
    )

    NonNodeData = @{
        SomeMessage = "I love Azure Automation DSC!"
    }
}

Start-AzureRmAutomationDscCompilationJob -ResourceGroupName "MyResourceGroup" -AutomationAccountName "MyAutomationAccount" -ConfigurationName "ConfigurationDataSample" -ConfigurationData $ConfigData

```

Assets

Asset references are the same in Azure Automation DSC configurations and runbooks. See the following for more information:

- [Certificates](#)
- [Connections](#)
- [Credentials](#)
- [Variables](#)

Credential Assets

While DSC configurations in Azure Automation can reference credential assets using **Get-AzureRmAutomationCredential**, credential assets can also be passed in via parameters, if desired. If a configuration takes a parameter of **PSCredential** type, then you need to pass the string name of an Azure Automation credential asset as that parameter's value, rather than a PSCredential object. Behind the scenes, the

Azure Automation credential asset with that name will be retrieved and passed to the configuration.

Keeping credentials secure in node configurations (MOF configuration documents) requires encrypting the credentials in the node configuration MOF file. Azure Automation takes this one step further and encrypts the entire MOF file. However, currently you must tell PowerShell DSC it is okay for credentials to be outputted in plain text during node configuration MOF generation, because PowerShell DSC doesn't know that Azure Automation will be encrypting the entire MOF file after its generation via a compilation job.

You can tell PowerShell DSC that it is okay for credentials to be outputted in plain text in the generated node configuration MOFs using **ConfigurationData**. You should pass `PSDscAllowPlainTextPassword = $true` via **ConfigurationData** for each node block's name that appears in the DSC configuration and uses credentials.

The following example shows a DSC configuration that uses an Automation credential asset.

```
Configuration CredentialSample
{
    $Cred = Get-AzureRmAutomationCredential -ResourceGroupName "ResourceGroup01" -AutomationAccountName "AutomationAcct" -Name "SomeCredentialAsset"

    Node $AllNodes.NodeName
    {
        File ExampleFile
        {
            SourcePath = "\\\$Server\share\path\file.ext"
            DestinationPath = "C:\destinationPath"
            Credential = $Cred
        }
    }
}
```

You can compile the DSC configuration above with PowerShell. The below PowerShell adds two node configurations to the Azure Automation DSC Pull Server: **CredentialSample.MyVM1** and **CredentialSample.MyVM2**.

```
$ConfigData = @{
    AllNodes = @(
        @{
            NodeName = "*"
            PSDscAllowPlainTextPassword = $True
        },
        @{
            NodeName = "MyVM1"
        },
        @{
            NodeName = "MyVM2"
        }
    )
}

Start-AzureRmAutomationDscCompilationJob -ResourceGroupName "MyResourceGroup" -AutomationAccountName "MyAutomationAccount" -ConfigurationName "CredentialSample" -ConfigurationData $ConfigData
```

Importing node configurations

You can also import node configurations (MOFs) that you have compiled outside of Azure. One advantage of this is that node configurations can be signed. A signed node configuration is verified locally on a managed node by the DSC agent, ensuring that the configuration being applied to the node comes from an authorized source.

NOTE

You can use import signed configurations into your Azure Automation account, but Azure Automation does not currently support compiling signed configurations.

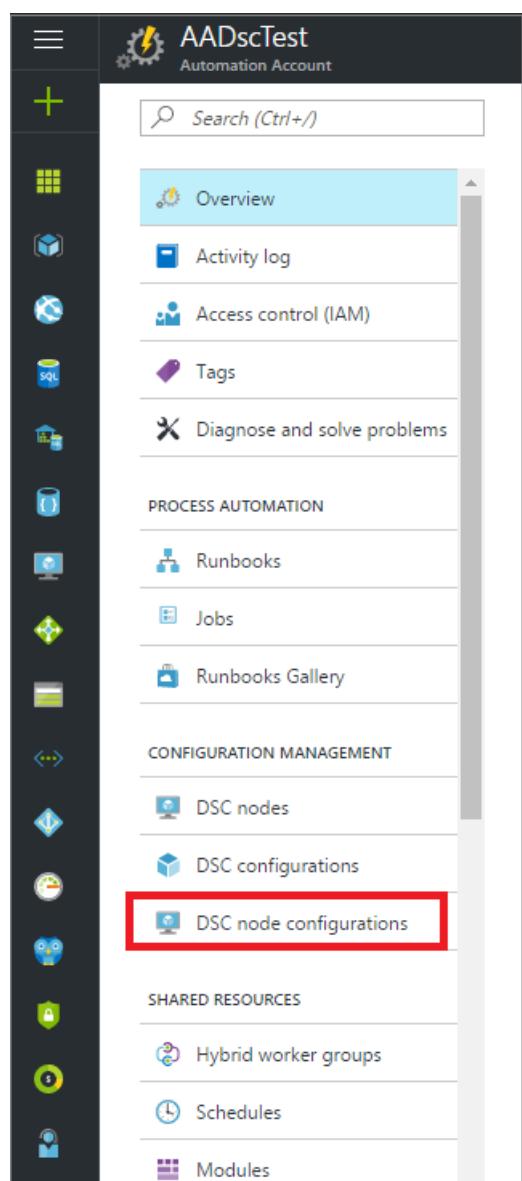
NOTE

A node configuration file must be no larger than 1 MB to allow it to be imported into Azure Automation.

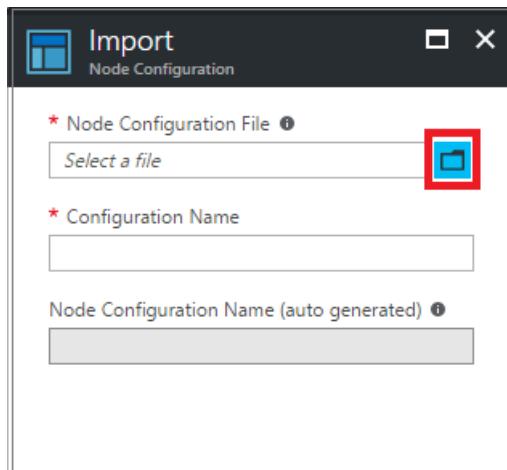
You can learn how to sign node configurations at <https://msdn.microsoft.com/en-us/powershell/wmf/5.1/dsc-improvements#how-to-sign-configuration-and-module>.

Importing a node configuration in the Azure portal

1. From your Automation account, click **DSC node configurations**.



2. In the **DSC node configurations** blade, click **Add a NodeConfiguration**.
3. In the **Import** blade, click the folder icon next to the **Node Configuration File** textbox to browse for a node configuration file (MOF) on your local computer.



4. Enter a name in the **Configuration Name** textbox. This name must match the name of the configuration from which the node configuration was compiled.
5. Click **OK**.

Importing a node configuration with PowerShell

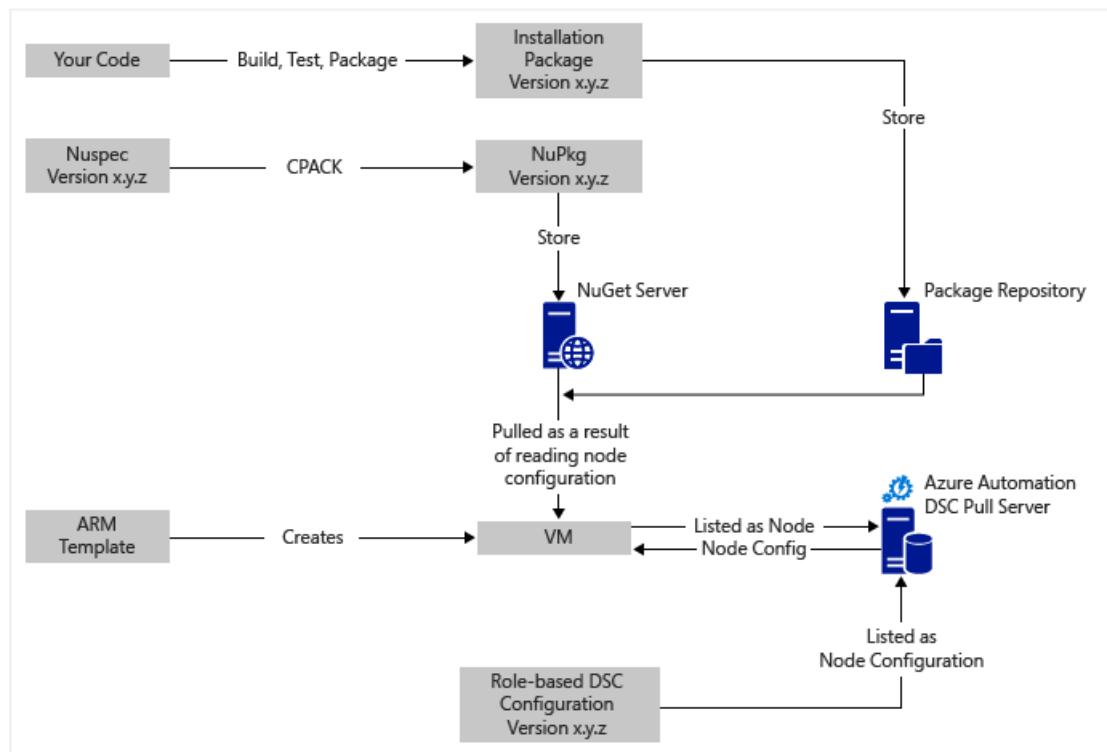
You can use the [Import-AzureRmAutomationDscNodeConfiguration](#) cmdlet to import a node configuration into your automation account.

```
Import-AzureRmAutomationDscNodeConfiguration -AutomationAccountName "MyAutomationAccount" -ResourceGroupName "MyResourceGroup" -ConfigurationName "MyNodeConfiguration" -Path "C:\MyConfigurations\TestVM1.mof"
```

Usage Example: Continuous deployment to Virtual Machines using Automation DSC and Chocolatey

1/23/2017 • 11 min to read • [Edit on GitHub](#)

In a DevOps world there are many tools to assist with various points in the Continuous Integration pipeline. Azure Automation Desired State Configuration (DSC) is a welcome new addition to the options that DevOps teams can employ. This article demonstrates setting up Continuous Deployment (CD) for a Windows computer. You can easily extend the technique to include as many Windows computers as necessary in the role (a web site, for example), and from there to additional roles as well.



At a high level

There is quite a bit going on here, but fortunately it can be broken into two main processes:

- Writing code and testing it, then creating and publishing installation packages for major and minor versions of the system.
- Creating and managing VMs that will install and execute the code in the packages.

Once both of these core processes are in place, it's a short step to automatically update the package running on any particular VM as new versions are created and deployed.

Component overview

Package managers such as [apt-get](#) are pretty well known in the Linux world, but not so much in the Windows world. [Chocolatey](#) is such a thing, and Scott Hanselman's [blog](#) on the topic is a great intro. In a nutshell, Chocolatey allows you to install packages from a central repository of packages into a Windows system using the command line. You can create and manage your own repository, and Chocolatey can install packages from any number of repositories that you designate.

Desired State Configuration (DSC) ([overview](#)) is a PowerShell tool that allows you to declare the configuration that you want for a machine. For example, you can say, "I want Chocolatey installed, I want IIS installed, I want port 80 opened, I want version 1.0.0 of my website installed." The DSC Local Configuration Manager (LCM) implements that configuration. A DSC Pull Server holds a repository of configurations for your machines. The LCM on each machine checks in periodically to see if its configuration matches the stored configuration. It can either report status or attempt to bring the machine back into alignment with the stored configuration. You can edit the stored configuration on the pull server to cause a machine or set of machines to come into alignment with the changed configuration.

Azure Automation is a managed service in Microsoft Azure that allows you to automate various tasks using runbooks, nodes, credentials, resources and assets such as schedules and global variables. Azure Automation DSC extends this automation capability to include PowerShell DSC tools. Here's a great [overview](#).

A DSC Resource is a module of code that has specific capabilities, such as managing networking, Active Directory, or SQL Server. The Chocolatey DSC Resource knows how to access a NuGet Server (among others), download packages, install packages, and so on. There are many other DSC Resources in the [PowerShell Gallery](#). These modules are installed into your Azure Automation DSC Pull Server (by you) so they can be used by your configurations.

Resource Manager templates provide a declarative way of generating your infrastructure - things like networks, subnets, network security and routing, load balancers, NICs, VMs, and so on. Here's an [article](#) that compares the Resource Manager deployment model (declarative) with the Azure Service Management (ASM, or classic) deployment model (imperative), and discusses the core resource providers, compute, storage and network.

One key feature of an Resource Manager template is its ability to install a VM extension into the VM as it's provisioned. A VM extension has specific capabilities such as running a custom script, installing anti-virus software, or running a DSC configuration script. There are many other types of VM extensions.

Quick trip around the diagram

Starting at the top, you write your code, build and test, then create an installation package. Chocolatey can handle various types of installation packages, such as MSI, MSU, ZIP. And you have the full power of PowerShell to do the actual installation if Chocolatey's native capabilities aren't quite up to it. Put the package into someplace reachable – a package repository. This usage example uses a public folder in an Azure blob storage account, but it can be anywhere. Chocolatey works natively with NuGet servers and a few others for management of package metadata. [This article](#) describes the options. This usage example uses NuGet. A Nuspec is metadata about your packages. The Nuspec's are "compiled" into NuPkg's and stored in a NuGet server. When your configuration requests a package by name, and references a NuGet server, the Chocolatey DSC Resource (now on the VM) grabs the package and installs it for you. You can also request a specific version of a package.

In the bottom left portion of the picture, there is an Azure Resource Manager (ARM) template. In this usage example, the VM extension registers the VM with the Azure Automation DSC Pull Server (that is, a pull server) as a Node. The configuration is stored in the pull server. Actually, it's stored twice: once as plain text and once compiled as an MOF file (for those that know about such things.) In the portal, the MOF is a "node configuration" (as opposed to simply "configuration"). It's the artifact that's associated with a Node so the node will know its configuration. Details below show how to assign the node configuration to the node.

Presumably you're already doing the bit at the top, or most of it. Creating the nuspec, compiling and storing it in a NuGet server is a small thing. And you're already managing VMs. Taking the next step to continuous deployment requires setting up the pull server (once), registering your nodes with it (once), and creating and storing the configuration there (initially). Then as packages are upgraded and deployed to the repository, refresh the Configuration and Node Configuration in the pull server (repeat as needed).

If you're not starting with an ARM template, that's also OK. There are PowerShell cmdlets designed to help you register your VMs with the pull server and all of the rest. For more details, see this article: [Onboarding machines for](#)

Step 1: Setting up the pull server and automation account

At an authenticated (Add-AzureRmAccount) PowerShell command line: (can take a few minutes while the pull server is set up)

```
New-AzureRmResourceGroup -Name MY-AUTOMATION-RG -Location MY-RG-LOCATION-IN-QUOTES  
New-AzureRmAutomationAccount -ResourceGroupName MY-AUTOMATION-RG -Location MY-RG-LOCATION-IN-QUOTES -Name MY-AUTOMATION-ACCOUNT
```

You can put your automation account into any of the following regions (aka location): East US 2, South Central US, US Gov Virginia, West Europe, Southeast Asia, Japan East, Central India and Australia Southeast, Canada Central, North Europe.

Step 2: VM extension tweaks to the ARM template

Details for VM registration (using the PowerShell DSC VM extension) provided in this [Azure Quickstart Template](#). This step registers your new VM with the pull server in the list of DSC Nodes. Part of this registration is specifying the node configuration to be applied to the node. This node configuration doesn't have to exist yet in the pull server, so it's OK that Step 4 is where this is done for the first time. But here in Step 2 you do need to have decided the name of the node and the name of the configuration. In this usage example, the node is 'isvbox' and the configuration is 'ISVBoxConfig'. So the node configuration name (to be specified in DeploymentTemplate.json) is 'ISVBoxConfig.isvbox'.

Step 3: Adding required DSC resources to the pull server

The PowerShell Gallery is instrumented to install DSC resources into your Azure Automation account. Navigate to the resource you want and click the "Deploy to Azure Automation" button.

The screenshot shows the PowerShell Gallery interface. At the top, there's a navigation bar with back, forward, refresh, and search icons, followed by the URL 'powershellgallery.com/packages/xNetworking'. To the right are a bookmark icon and a star icon. Below the URL, a yellow banner states 'This gallery is under limited preview.' The main header features a blue PowerShell icon and the text 'PowerShell Gallery'. A dark navigation bar below has links for 'Home', 'Get Started', 'Modules' (which is underlined), 'Publish Module', and 'Statistics'. To the right of the navigation bar is a search bar with the placeholder 'Search'. On the left side, there's a large blue PowerShell icon. In the center, the 'xNetworking 2.3.0' module is displayed. It includes a brief description: 'Module with DSC Resources for Networking area'. Below this are sections for 'Inspect' (with a command: PS> Save-Module -Name xNetworking -Path <path>), 'Install' (with a command: PS> Install-Module -Name xNetworking), and 'Deploy' (with a prominent blue button labeled 'Deploy to Azure Automation'). On the far left, there are download statistics: '2,175 Downloads' and '1,097 Downloads of v 2.3.0'. At the bottom left, it says '2015-09-11 Last published'.

Another technique recently added to the Azure Portal allows you to pull in new modules or update existing modules. Click through the Automation Account resource, the Assets tile, and finally the Modules tile. The Browse Gallery icon allows you to see the list of modules in the gallery, drill down into details and ultimately import into your Automation Account. This is a great way to keep your modules up to date from time to time. And, the import feature checks dependencies with other modules to ensure nothing gets out of sync.

Or, there's the manual approach. The folder structure of a PowerShell Integration Module for a Windows computer is a little different from the folder structure expected by the Azure Automation. This requires a little tweaking on your part. But it's not hard, and it's done only once per resource (unless you want to upgrade it in future.) For more information on authoring PowerShell Integration Modules, see this article: [Authoring Integration Modules for Azure Automation](#)

- Install the module that you need on your workstation, as follows:
 - Install [Windows Management Framework, v5](#) (not needed for Windows 10)
 - `Install-Module -Name MODULE-NAME` <—grabs the module from the PowerShell Gallery
- Copy the module folder from `c:\Program Files\WindowsPowerShell\Modules\MODULE-NAME` to a temp folder
- Delete samples and documentation from the main folder
- Zip the main folder, naming the ZIP file exactly the same as the folder
- Put the ZIP file into a reachable HTTP location, such as blob storage in an Azure Storage Account.
- Run this PowerShell:

```
New-AzureRmAutomationModule ` 
-ResourceGroupName MY-AUTOMATION-RG -AutomationAccountName MY-AUTOMATION-ACCOUNT ` 
-Name MODULE-NAME -ContentLink "https://STORAGE-URI/CONTAINERNAME/MODULE-NAME.zip"
```

The included example performs these steps for cChoco and xNetworking. See the [Notes](#) for special handling for cChoco.

Step 4: Adding the node configuration to the pull server

There's nothing special about the first time you import your configuration into the pull server and compile. All subsequent import/compiles of the same configuration look exactly the same. Each time you update your package and need to push it out to production you do this step after ensuring the configuration file is correct – including the new version of your package. Here's the configuration file and PowerShell:

ISVBoxConfig.ps1:

```

Configuration ISVBoxConfig
{
    Import-DscResource -ModuleName cChoco
    Import-DscResource -ModuleName xNetworking

    Node "isvbox" {

        cChocoInstaller installChoco
        {
            InstallDir = "C:\choco"
        }

        WindowsFeature installIIS
        {
            Ensure="Present"
            Name="Web-Server"
        }

        xFirewall WebFirewallRule
        {
            Direction = "Inbound"
            Name = "Web-Server-TCP-In"
            DisplayName = "Web Server (TCP-In)"
            Description = "IIS allow incoming web site traffic."
            DisplayGroup = "IIS Incoming Traffic"
            State = "Enabled"
            Access = "Allow"
            Protocol = "TCP"
            LocalPort = "80"
            Ensure = "Present"
        }

        cChocoPackageInstaller trivialWeb
        {
            Name = "trivialweb"
            Version = "1.0.0"
            Source = "MY-NUGET-V2-SERVER-ADDRESS"
            DependsOn = "[cChocoInstaller]installChoco",
                        "[WindowsFeature]installIIS"
        }
    }
}

```

New-ConfigurationScript.ps1:

```

Import-AzureRmAutomationDscConfiguration ` 
    -ResourceGroupName MY-AUTOMATION-RG -AutomationAccountName MY-AUTOMATION-ACCOUNT ` 
    -SourcePath C:\temp\AzureAutomationDsc\ISVBoxConfig.ps1 ` 
    -Published -Force

$jobData = Start-AzureRmAutomationDscCompilationJob ` 
    -ResourceGroupName MY-AUTOMATION-RG -AutomationAccountName MY-AUTOMATION-ACCOUNT ` 
    -ConfigurationName ISVBoxConfig

$compilationJobId = $jobData.Id

Get-AzureRmAutomationDscCompilationJob ` 
    -ResourceGroupName MY-AUTOMATION-RG -AutomationAccountName MY-AUTOMATION-ACCOUNT ` 
    -Id $compilationJobId

```

These steps result in a new node configuration named "ISVBoxConfig.isvbox" being placed on the pull server. The node configuration name is built as "configurationName.nodeName".

Step 5: Creating and maintaining package metadata

For each package that you put into the package repository, you need a nuspec that describes it. That nuspec must be compiled and stored in your NuGet server. This process is described [here](#). You can use MyGet.org as a NuGet server. They sell this service, but have a starter SKU that's free. At NuGet.org you'll find instructions on installing your own NuGet server for your private packages.

Step 6: Tying it all together

Each time a version passes QA and is approved for deployment, the package is created, nuspec and nupkg updated and deployed to the NuGet server. In addition, the configuration (Step 4 above) must be updated to agree with the new version number. It must be sent to the pull server and compiled. From that point on, it's up to the VMs that depend on that configuration to pull the update and install it. Each of these updates are simple - just a line or two of PowerShell. In the case of Visual Studio Team Services, some of them are encapsulated in build tasks that can be chained together in a build. This [article](#) provides more details. This [GitHub repo](#) details the various available build tasks.

Notes

This usage example starts with a VM from a generic Windows Server 2012 R2 image from the Azure gallery. You can start from any stored image and then tweak from there with the DSC configuration. However, changing configuration that is baked into an image is much harder than dynamically updating the configuration using DSC.

You don't have to use an ARM template and the VM extension to use this technique with your VMs. And your VMs don't have to be on Azure to be under CD management. All that's necessary is that Chocolatey be installed and the LCM configured on the VM so it knows where the pull server is.

Of course, when you update a package on a VM that's in production, you need to take that VM out of rotation while the update is installed. How you do this varies widely. For example, with a VM behind an Azure Load Balancer, you can add a Custom Probe. While updating the VM, have the probe endpoint return a 400. The tweak necessary to cause this change can be inside your configuration, as can the tweak to switch it back to returning a 200 once the update is complete.

Full source for this usage example is in [this Visual Studio project](#) on GitHub.

Related Articles

- [Azure Automation DSC Overview](#)
- [Azure Automation DSC cmdlets](#)
- [Onboarding machines for management by Azure Automation DSC](#)

Authenticate Runbooks with Azure Service Management and Resource Manager

1/17/2017 • 4 min to read • [Edit on GitHub](#)

This article describes the steps you must perform to configure an Azure AD User account for Azure Automation runbooks running against Azure Service Management or Azure Resource Manager resources. While this continues to be a supported authentication identity for your Azure Resource Manager based runbooks, the recommended method is using the new Azure Run As account.

Create a new Azure Active Directory user

1. Log in to the Azure Classic Portal as a service administrator for the Azure subscription you want to manage.
2. Select **Active Directory**, and then select the name of your organization directory.
3. Select the **Users** tab, and then, in the command area, select **Add User**.
4. On the **Tell us about this user** page, under **Type of user**, select **New user in your organization**.
5. Enter a user name.
6. Select the directory name that is associated with your Azure subscription on the Active Directory page.
7. On the **user profile** page, provide a first and last name, a user-friendly name, and User from the **Roles** list. Do not **Enable Multi-Factor Authentication**.
8. Note the user's full name and temporary password.
9. Select **Settings > Administrators > Add**.
10. Type the full user name of the user that you created.
11. Select the subscription that you want the user to manage.
12. Log out of Azure and then log back in with the account you just created. You will be prompted to change the user's password.

Create an Automation account in Azure Classic Portal

In this section, you will perform the following steps to create a new Azure Automation account in the Azure Portal that will be used with your runbooks managing resources in Azure Service Manager and Azure Resource Manager mode.

NOTE

Automation accounts created with the Azure Classic Portal can be managed by both the Azure Classic and Azure Portal and either set of cmdlets. Once the account is created, it makes no difference how you create and manage resources within the account. If you are planning to continue to use the Azure Classic Portal, then you should use it instead of the Azure Portal to create any Automation accounts.

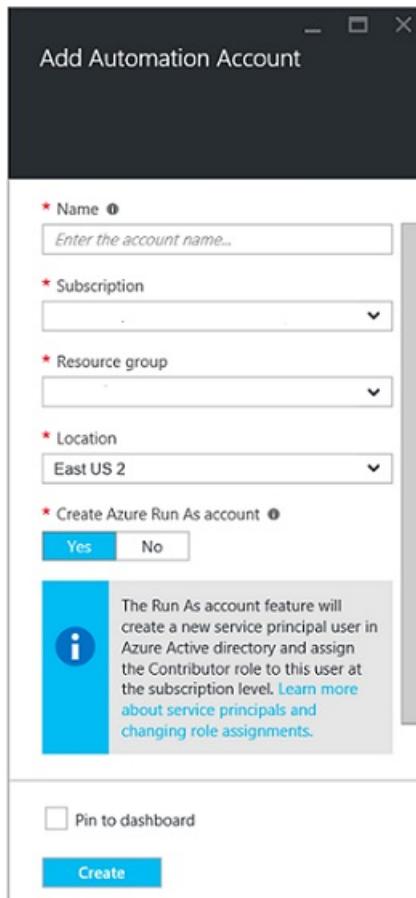
1. Log in to the Azure Classic Portal as a service administrator for the Azure subscription you want to manage.
2. Select **Automation**.
3. On the **Automation** page, select **Create an Automation Account**.
4. In the **Create an Automation Account** box, type in a name for your new Automation account and select a **Region** from the drop-down list.
5. Click **OK** to accept your settings and create the account.
6. After it is created it will be listed on the **Automation** page.

7. Click on the account and it will bring you to the Dashboard page.
8. On the Automation Dashboard page, select **Assets**.
9. On the **Assets** page, select **Add Settings** located at the bottom of the page.
10. On the **Add Settings** page, select **Add Credential**.
11. On the **Define Credential** page, select **Windows PowerShell Credential** from the **Credential Type** drop-down list and provide a name for the credential.
12. On the following **Define Credential** page type in the username of the AD user account created earlier in the **User Name** field and the password in the **Password** and **Confirm Password** fields. Click **OK** to save your changes.

Create an Automation account in the Azure Portal

In this section, you will perform the following steps to create a new Azure Automation account in the Azure Portal that will be used with your runbooks managing resources in Azure Resource Manager mode.

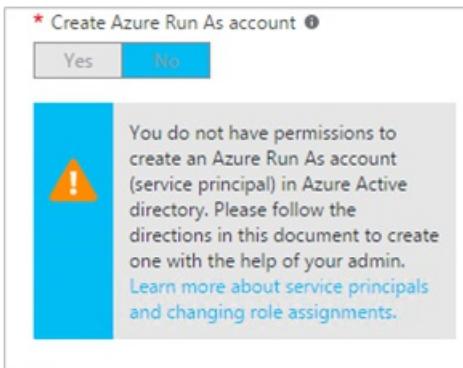
1. Log in to the Azure portal as a service administrator for the Azure subscription you want to manage.
2. Select **Automation Accounts**.
3. In the Automation Accounts blade, click **Add**.



4. In the **Add Automation Account** blade, in the **Name** box type in a name for your new Automation account.
5. If you have more than one subscription, specify the one for the new account, as well as a new or existing **Resource group** and an Azure datacenter **Location**.
6. Select the value **No** for the **Create Azure Run As account** option, and click the **Create** button.

NOTE

If you choose to not create the Run As account by selecting the option **No**, you will be presented with a warning message in the **Add Automation Account** blade. While the account is created and assigned to the **Contributor** role in the subscription, it will not have a corresponding authentication identity within your subscriptions directory service and therefore, no access resources in your subscription. This will prevent any runbooks referencing this account from being able to authenticate and perform tasks against Azure Resource Manager resources.



7. While Azure creates the Automation account, you can track the progress under **Notifications** from the menu.

When the creation of the credential is completed, you will then need to create a Credential Asset to associate the Automation Account with the AD User account created earlier. Remember, we only created the Automation account and it is not associated with an authentication identity. Perform the steps outlined in the [Credential assets in Azure Automation article](#) and enter the value for **username** in the format **domain\user**.

Use the credential in a runbook

You can retrieve the credential in a runbook using the [Get-AutomationPSCredential](#) activity and then use it with [Add-AzureAccount](#) to connect to your Azure subscription. If the credential is an administrator of multiple Azure subscriptions, then you should also use [Select-AzureSubscription](#) to specify the correct one. This is shown in the sample Windows PowerShell below that will typically appear at the top of most Azure Automation runbooks.

```
$cred = Get-AutomationPSCredential -Name "myuseraccount.onmicrosoft.com"
Add-AzureAccount -Credential $cred
Select-AzureSubscription -SubscriptionName "My Subscription"
```

You should repeat these lines after any [checkpoints](#) in your runbook. If the runbook is suspended and then resumes on another worker, then it will need to perform the authentication again.

Next Steps

- Review the different runbook types and steps for creating your own runbooks from the following article [Azure Automation runbook types](#)

Authenticate Runbooks with Amazon Web Services

1/17/2017 • 1 min to read • [Edit on GitHub](#)

Automating common tasks with resources in Amazon Web Services (AWS) can be accomplished with Automation runbooks in Azure. You can automate many tasks in AWS using Automation runbooks just like you can with resources in Azure. All that is required are two things:

- An AWS subscription and a set of credentials. Specifically your AWS Access Key and Secret Key. For more information, please review the article [Using AWS Credentials](#).
- An Azure subscription and Automation account. For more information on setting up an Azure Automation account, please review the article [Configure Azure Run As Account](#).

To authenticate with AWS, you must specify a set of AWS credentials to authenticate your runbooks running from Azure Automation. If you already have an Automation account created and you want to use that to authenticate with AWS, you can follow the steps in the following section. If you want to dedicated an account for runbooks targetting AWS resources, you should first create a new [Automation Run As account](#) (skip the option to create a service principal) and then follow the steps below.

Configure Automation account

For Azure Automation to communicate with AWS, you will first need to retrieve your AWS credentials and store them as assets in Azure Automation. Perform the following steps documented in the AWS document [Managing Access Keys for your AWS Account](#) to create an Access Key and copy the **Access Key ID** and **Secret Access Key** (optionally download your key file to store it somewhere safe).

After you have created and copied your AWS security keys, you will need to create a Credential asset with an Azure Automation account to securely store them and reference them with your runbooks. Follow the steps in the section **Creating a new credential asset** in the [Credential assets in Azure Automation](#) article and enter the following information:

1. In the **Name** box, enter **AWScred** or an appropriate value following your naming standards.
2. In the **User name** box type your **Access ID** and your **Secret Access Key** in the **Password** and **Confirm password** box.

Next steps

- Review the solution article [Automating deployment of a VM in Amazon Web Services](#) to learn how to create runbooks to automate tasks in AWS.

Authenticate Runbooks with Azure Run As account

2/24/2017 • 19 min to read • [Edit on GitHub](#)

This topic will show you how to configure an Automation account from the Azure portal using the Run As account feature to authenticate runbooks managing resources in either Azure Resource Manager or Azure Service Management.

When you create a new Automation account in the Azure portal, it automatically creates:

- Run As account which creates a new service principal in Azure Active Directory, a certificate, and assigns the Contributor role-based access control (RBAC), which will be used to manage Resource Manager resources using runbooks.
- Classic Run As account by uploading a management certificate, which will be used to manage Azure Service Management or classic resources using runbooks.

This simplifies the process for you and helps you quickly start building and deploying runbooks to support your automation needs.

Using a Run As and Classic Run As account, you can:

- Provide a standardized way to authenticate with Azure when managing Azure Resource Manager or Azure Service Management resources from runbooks in the Azure portal.
- Automate the use of global runbooks configured in Azure Alerts.

NOTE

The Azure [Alert integration feature](#) with Automation Global Runbooks requires an Automation account that is configured with a Run As and Classic Run As account. You can either select an Automation account that already has a Run As and Classic Run As account defined or choose to create a new one.

We will show you how to create the Automation account from the Azure portal, update an Automation account using PowerShell, manage the account configuration, and demonstrate how to authenticate in your runbooks.

Before we do that, there are a few things that you should understand and consider before proceeding.

1. This does not impact existing Automation accounts already created in either the classic or Resource Manager deployment model.
2. This will only work for Automation accounts created through the Azure portal. Attempting to create an account from the classic portal will not replicate the Run As account configuration.
3. If you currently have runbooks and assets (i.e. schedules, variables, etc.) previously created to manage classic resources, and you want those runbooks to authenticate with the new Classic Run As account, you will need to create a Classic Run As Account using Managing an Run As Account or update your existing account using the PowerShell script below.
4. To authenticate using the new Run As account and Classic Run As Automation account, you will need to modify your existing runbooks with the example code below. **Please note** that the Run As account is for authentication against Resource Manager resources using the certificate-based service principal, and the Classic Run As account is for authenticating against Service Management resources with the management certificate.

Create a new Automation Account from the Azure portal

In this section, you will perform the following steps to create a new Azure Automation account from the Azure portal. This creates both the Run As and classic Run As account.

NOTE

The user performing these steps must be a member of the Service Admins role or co-administrator of the subscription which is granting access to the subscription for the user. The user must also be added as a User to that subscriptions default Active Directory; the account does not need to be assigned to a privileged role. Users who are not a member of the Subscription's Active Directory prior to being added to the Co-Admin role of the subscription will be added to Active Directory as a Guest and will see the "You do not have permissions to create..." warning in the **Add Automation Account** blade. Users who were added to the co-admin role first can be removed from the subscriptions Active Directory and re-added to make them a full User in Active Directory. This situation can be verified from the **Azure Active Directory** pane in the Azure portal by selecting **Users and groups**, select **All users** and after selecting the specific user select **Profile**. The value of the **User type** attribute under the users profile should not equal **Guest**.

1. Sign in to the Azure portal with an account that is a member of the Subscription Admins role and co-administrator of the subscription.
2. Select **Automation Accounts**.
3. In the Automation Accounts blade, click **Add**.

Add Automation...



* Name ⓘ

Enter the account name...

* Subscription



* Resource group ⓘ

Create new Use existing



* Location



* Create Azure Run As account ⓘ



The Run As account feature will
create a Run As account and a
Classic Run As account.[Click here to
learn more about Run As accounts.](#)



Pin to dashboard

[Automation options](#)

NOTE

If you see the following warning in the **Add Automation Account** blade, this is because your account is not a member of the Subscription Admins role and co-admin of the subscription.



You do not have permissions to create a Run As account in Azure Active directory. Please follow the directions in the documentation to learn how to create a Run As account.[Click here to learn more about Run As accounts.](#)

4. In the **Add Automation Account** blade, in the **Name** box type in a name for your new Automation account.
5. If you have more than one subscription, specify one for the new account, as well as a new or existing **Resource group** and an Azure datacenter **Location**.
6. Verify the value **Yes** is selected for the **Create Azure Run As account** option, and click the **Create** button.

NOTE

If you choose to not create the Run As account by selecting the option **No**, you will be presented with a warning message in the **Add Automation Account** blade. While the account is created in the Azure portal, it will not have a corresponding authentication identity within your classic or Resource Manager subscription directory service and therefore, no access to resources in your subscription. This will prevent any runbooks referencing this account from being able to authenticate and perform tasks against resources in those deployment models.



You have chosen not to create a Run As Account. Doing so might block the execution of some runbooks due to lack of access to required resources.[Click here to learn more about Run As accounts.](#)

When the service principal is not created the Contributor role will not be assigned.

7. While Azure creates the Automation account, you can track the progress under **Notifications** from the menu.

Resources included

When the Automation account is successfully created, several resources are automatically created for you. The following table summarizes resources for the Run As account.

RESOURCE	DESCRIPTION
AzureAutomationTutorial Runbook	An example Graphical runbook that demonstrates how to authenticate using the Run As account and gets all the Resource Manager resources.
AzureAutomationTutorialScript Runbook	An example PowerShell runbook that demonstrates how to authenticate using the Run As account and gets all the Resource Manager resources.

RESOURCE	DESCRIPTION
AzureRunAsCertificate	Certificate asset automatically created during Automation account creation or using the PowerShell script below for an existing account. It allows you to authenticate with Azure so that you can manage Azure Resource Manager resources from runbooks. This certificate has a one-year lifespan.
AzureRunAsConnection	Connection asset automatically created during Automation account creation or using the PowerShell script below for an existing account.

The following table summarizes resources for the Classic Run As account.

RESOURCE	DESCRIPTION
AzureClassicAutomationTutorial Runbook	An example Graphical runbook which gets all the Classic VMs in a subscription using the Classic Run As Account (certificate) and then outputs the VM name and status.
AzureClassicAutomationTutorial Script Runbook	An example PowerShell runbook which gets all the Classic VMs in a subscription using the Classic Run As Account (certificate) and then outputs the VM name and status.
AzureClassicRunAsCertificate	Certificate asset automatically created that is used to authenticate with Azure so that you can manage Azure classic resources from runbooks. This certificate has a one-year lifespan.
AzureClassicRunAsConnection	Connection asset automatically created that is used to authenticate with Azure so that you can manage Azure classic resources from runbooks.

Verify Run As authentication

Next we will perform a small test to confirm you are able to successfully authenticate using the new Run As account.

1. In the Azure portal, open the Automation account created earlier.
2. Click on the **Runbooks** tile to open the list of runbooks.
3. Select the **AzureAutomationTutorialScript** runbook and then click **Start** to start the runbook. You will receive a prompt verifying you wish to start the runbook.
4. A **runbook job** is created, the Job blade is displayed, and the job status displayed in the **Job Summary** tile.
5. The job status will start as *Queued* indicating that it is waiting for a runbook worker in the cloud to become available. It will then move to *Starting* when a worker claims the job, and then *Running* when the runbook actually starts running.
6. When the runbook job completes, we should see a status of **Completed**.

The screenshot shows the Azure Automation Job blade for a runbook named "AzureAutomationTutorialScript". The top navigation bar includes options like "Resume", "Stop", "Suspend", and "View source". Below the navigation is an "Overview" section with a "Job Summary" tile. The summary details the job ID (e94e68f6-dc0c-4a40-8fb9-f6dd7f6a553d), creation date (7/20/2016 2:45 PM), last update date (7/20/2016 2:47 PM), and the fact it ran on Azure. It also indicates the job is "Completed" with a green checkmark icon. To the right of the summary are tiles for "Runbook" (blue folder icon), "INPUT" (0 items, represented by a document icon with a pink X), and "Output" (document icon with a blue arrow). Below the overview is a "Status" section with tiles for "Errors" (0 items, pink X icon) and "Warnings" (0 items, orange exclamation mark icon). A "All Logs" tile (orange folder icon) is also present. There are "Add tiles" (+) buttons in both the "Overview" and "Status" sections.

7. To see the detailed results of the runbook, click on the **Output** tile.
8. In the **Output** blade, you should see it has successfully authenticated and returned a list of all resources available in the resource group.
9. Close the **Output** blade to return to the **Job Summary** blade.
10. Close the **Job Summary** and the corresponding **AzureAutomationTutorialScript** runbook blade.

Verify Classic Run As authentication

Next we will perform a small test to confirm you are able to successfully authenticate using the new Classic Run As account.

1. In the Azure portal, open the Automation account created earlier.
2. Click on the **Runbooks** tile to open the list of runbooks.
3. Select the **AzureClassicAutomationTutorialScript** runbook and then click **Start** to start the runbook. You will receive a prompt verifying you wish to start the runbook.
4. A **runbook job** is created, the Job blade is displayed, and the job status displayed in the **Job Summary** tile.
5. The job status will start as *Queued* indicating that it is waiting for a runbook worker in the cloud to become available. It will then move to *Starting* when a worker claims the job, and then *Running* when the runbook actually starts running.
6. When the runbook job completes, we should see a status of **Completed**.

The screenshot shows the Azure portal interface for a specific runbook. At the top, there's a navigation bar with icons for Home, Automation, Functions, Logic Apps, and App Services. Below that is a search bar and a 'New' button. The main content area has a title 'AzureClassicAutomationTutorialScript 7/20/201...' and a 'Job' icon. Below the title are buttons for Resume, Stop, Suspend, and View source. The main area is divided into sections: 'Overview' (Job Summary, Runbook, Input, Output), 'Status' (Errors, Warnings, All Logs), and 'Logs' (All logs). Each section has an 'Add tiles' button.

7. To see the detailed results of the runbook, click on the **Output** tile.
8. In the **Output** blade, you should see it has successfully authenticated and returned a list of all classic VM's in the subscription.
9. Close the **Output** blade to return to the **Job Summary** blade.
10. Close the **Job Summary** and the corresponding **AzureClassicAutomationTutorialScript** runbook blade.

Managing Azure Run As account

During the lifetime of your Automation account, you will need to renew the certificate before it expires or if you believe the account has been compromised, you can delete the Run As account and re-create it. This section will provide steps on how to perform these operations.

Certificate renewal

The certificate created for the Azure Run As account can be renewed at anytime, up until it expires, which is one year from date of creation. When you renew it, the old valid certificate will be retained in order to ensure that any runbooks queued up or actively running, which authenticate with the Run As account, will not be impacted. The certificate will continue to exist until expiration.

1. In the Azure portal, open the Automation account.
2. On the Automation account blade, in the account properties pane, select **Run As Accounts** under the section **Account Settings**.

 TestAutomationAcct
Automation Account

Search (Ctrl+/)

Connections

Certificates

Variables

RELATED RESOURCES

Solutions

Workspace

Unlink Workspace

ACCOUNT SETTINGS

Properties

Keys

Pricing tier and usage

Source Control

Run As Accounts

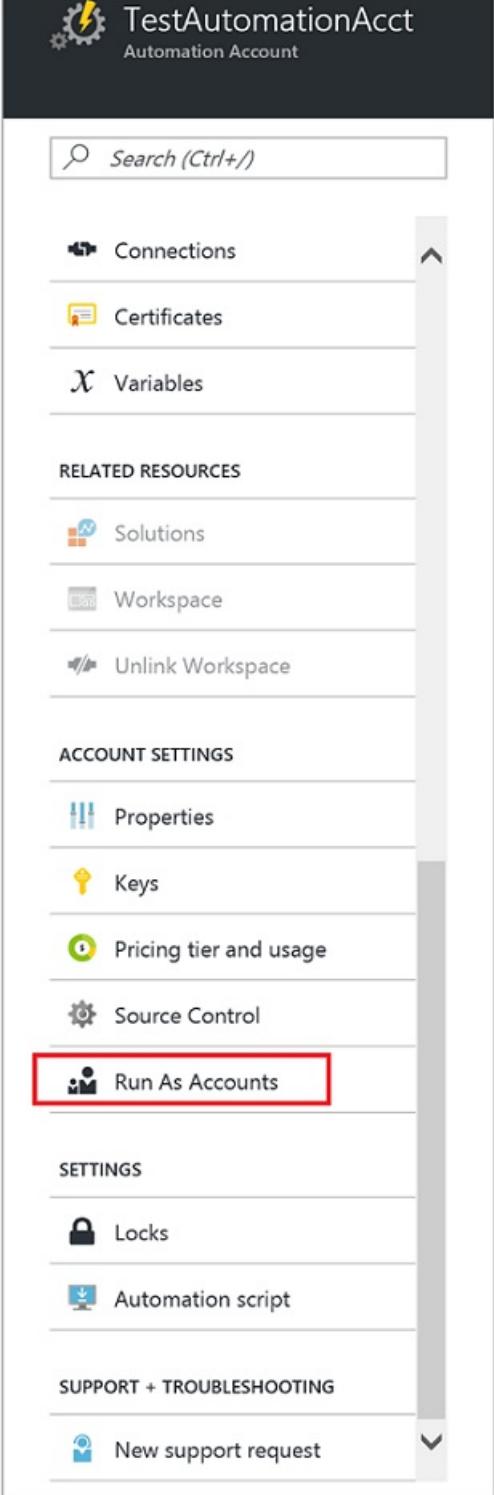
SETTINGS

Locks

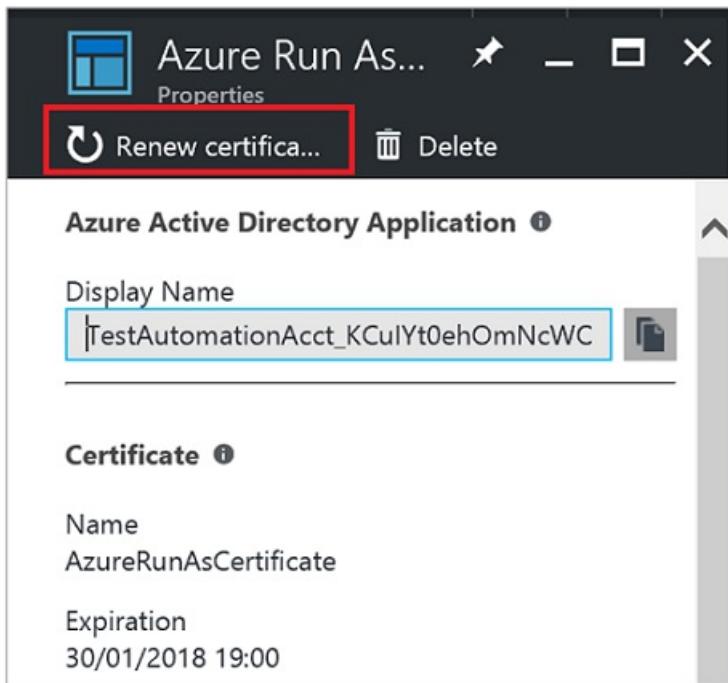
Automation script

SUPPORT + TROUBLESHOOTING

New support request



3. On the **Run As Accounts** properties blade, select either the Run As Account or Classic Run As account that you wish to renew the certificate for, and on the properties blade for the selected account, click **Renew certificate**.



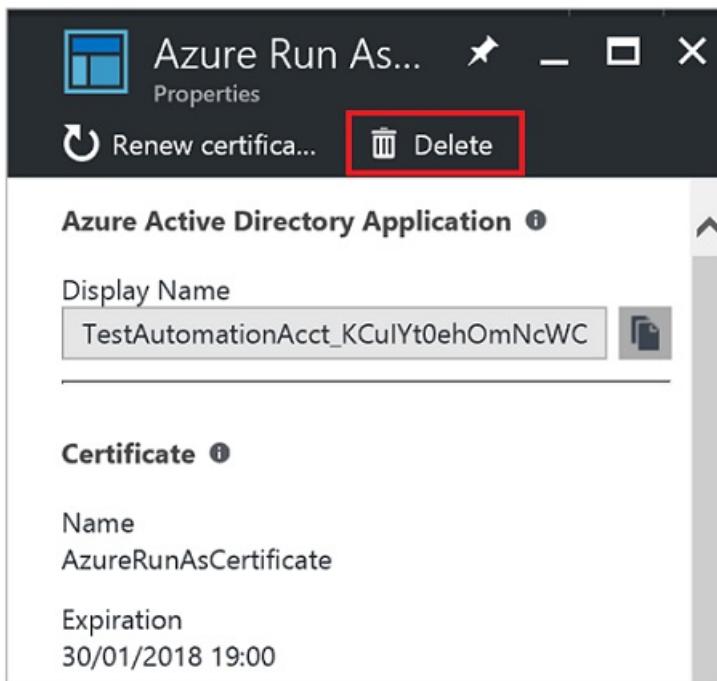
You will receive a prompt verifying you wish to proceed.

4. While the certificate is being renewed, you can track the progress under **Notifications** from the menu.

Delete Run As account

The following steps describe how to delete and re-create your Azure Run As or Classic Run As account. When you perform this action the Automation account is retained. After deleting the Run As or Classic Run As account, you can recreate it in the portal.

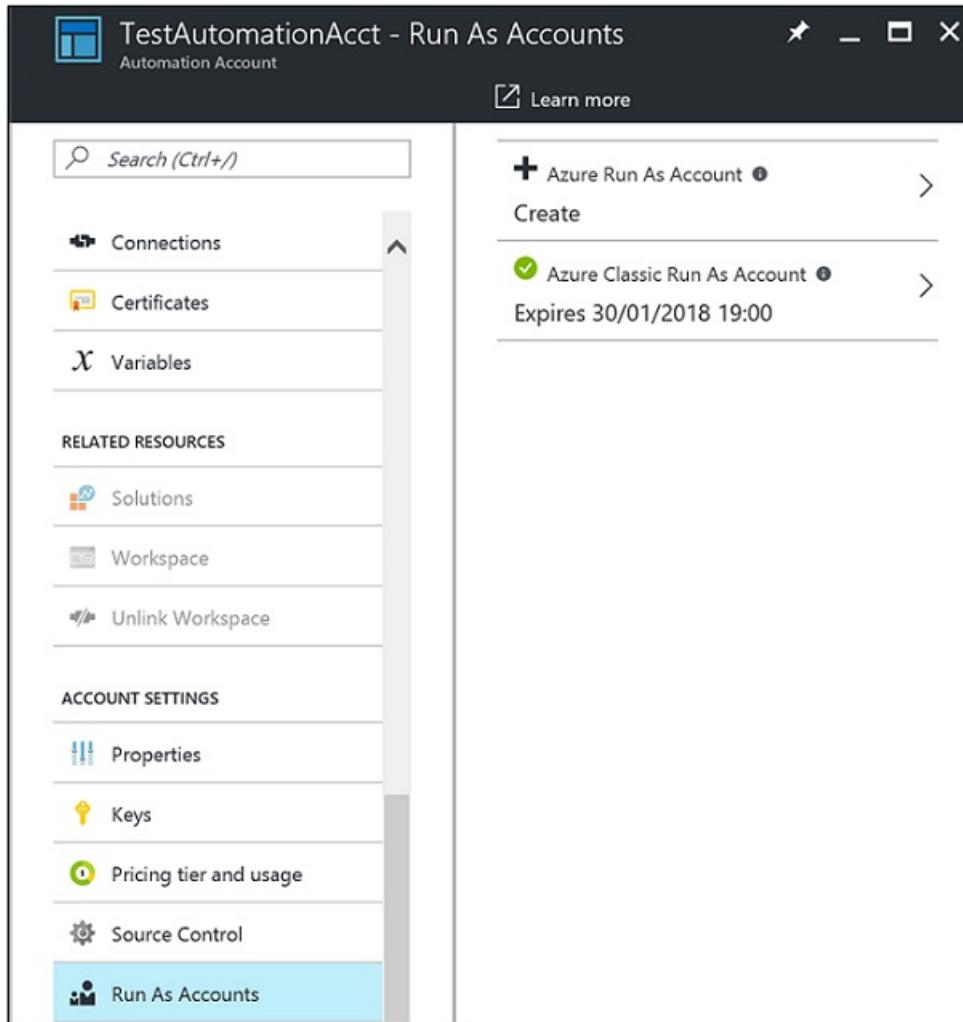
1. In the Azure portal, open the Automation account.
2. On the Automation account blade, in the account properties pane, select **Run As Accounts** under the section **Account Settings**.
3. On the **Run As Accounts** properties blade, select either the Run As Account or Classic Run As account that you wish to delete, and on the properties blade for the selected account, click **Delete**.



You will receive a prompt verifying you wish to proceed.

4. While the account is being deleted, you can track the progress under **Notifications** from the menu. Once the

deletion is complete, you can re-create it from the On the **Run As Accounts** properties blade and selecting the create option **Azure Run As Account**.



Misconfiguration

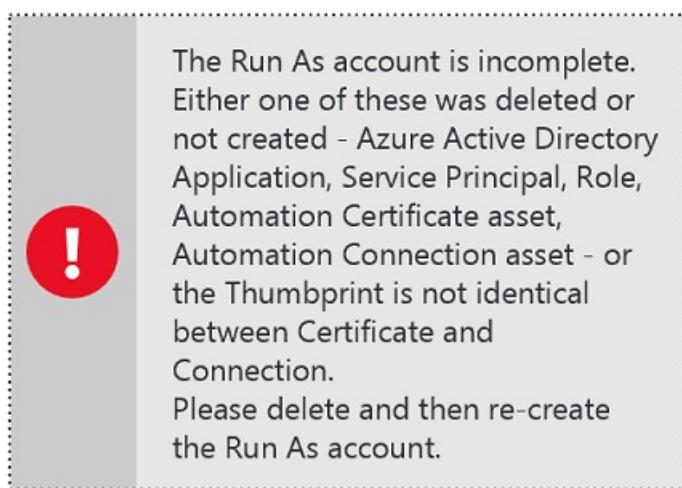
If any of the configuration items necessary for the Run As or Clasic Run As account to function properly are deleted or were not created properly during initial setup, such as:

- Certificate asset
- Connection asset
- Run As account has been removed from the contributor role
- Service principal or application in Azure AD

Automation will detect these changes and notify you with a status of **Incomplete** in the **Run As Accounts** properties blade for the account.

The screenshot shows the Azure portal interface for an Automation Account named 'TestAutomationAcct'. The left sidebar contains navigation links for 'Connections', 'Certificates', 'Variables', 'Solutions', 'Workspace', 'Unlink Workspace', 'Properties', 'Keys', 'Pricing tier and usage', 'Source Control', and 'Run As Accounts'. The 'Run As Accounts' link is highlighted with a blue background. The main content area displays two entries: 'Azure Run As Account' (Incomplete) and 'Azure Classic Run As Account' (Expires 31/01/2018 19:00).

When you select the Run As account, the following warning will be presented in the properties pane of the account:



If your Run As account is misconfigured, you can quickly resolve this by deleting and re-creating the Run As account.

Update an Automation Account using PowerShell

Here we provide you with the option to use PowerShell to update your existing Automation account if:

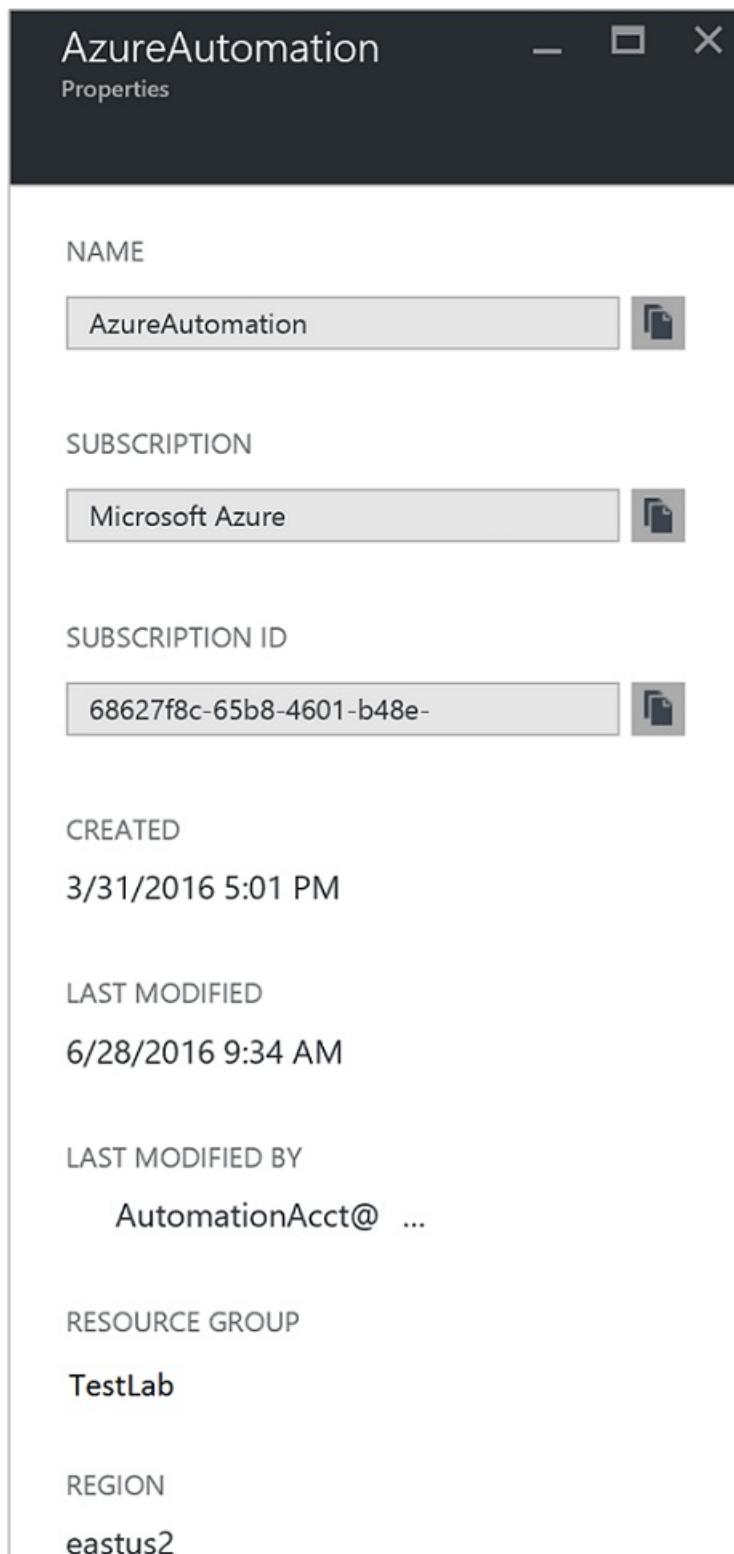
1. You created an Automation account, but declined to create the Run As account
2. You need to create an Automation account in Azure Government cloud

3. You already have an Automation account to manage Resource Manager resources and you want to update it to include the Run As account for runbook authentication
4. You already have an Automation account to manage classic resources and you want to update it to use the Classic Run As instead of creating a new account and migrating your runbooks and assets to it

Before proceeding, please verify the following:

1. This script supports running only on Windows 10 and Windows Server 2016 with Azure Resource Manager modules 2.01 and higher installed. It is not supported on earlier versions of Windows.
2. Azure PowerShell 1.0 and higher. For information about this release and how to install it, see [How to install and configure Azure PowerShell](#).
3. You have created an automation account. This account will be referenced as the value for parameters – `AutomationAccountName` and `-ApplicationDisplayName` in both scripts below.

To get the values for `SubscriptionID`, `ResourceGroup`, and `AutomationAccountName`, which are required parameters for the scripts, in the Azure portal select your Automation account from the **Automation account** blade and select **All settings**. From the **All settings** blade, under **Account Settings** select **Properties**. In the **Properties** blade, you can note these values.



Create Run As Account PowerShell script

The PowerShell script below will configure the following:

- An Azure AD application that will be authenticated with the self-signed cert, create a service principal account for this application in Azure AD, and assigned the Contributor role (you could change this to Owner or any other role) for this account in your current subscription. For further information, please review the [Role-based access control in Azure Automation](#) article.
- An Automation certificate asset in the specified automation account named **AzureRunAsCertificate**, which holds the certificate used by the service principal.
- An Automation connection asset in the specified automation account named **AzureRunAsConnection**, which holds the applicationId, tenantId, subscriptionId, and certificate thumbprint.

The steps below will walk you through the process of executing the script.

- Save the following script on your computer. In this example, save it with the filename **New-AzureServicePrincipal.ps1**.

```
#Requires -RunAsAdministrator
Param (
[Parameter(Mandatory=$true)]
[String] $ResourceGroup,
[Parameter(Mandatory=$true)]
[String] $AutomationAccountName,
[Parameter(Mandatory=$true)]
[String] $ApplicationDisplayName,
[Parameter(Mandatory=$true)]
[String] $SubscriptionId,
[Parameter(Mandatory=$true)]
[String] $CertPlainPassword,
[Parameter(Mandatory=$false)]
[int] $NoOfMonthsUntilExpired = 12

[Parameter(Mandatory=$True)]
[ValidateSet("AzureCloud", "AzureUSGovernment")]
[string]$Environment="AzureCloud"
)

#Check to see which cloud environment to sign into.
Switch ($Environment)
{
    "AzureCloud" {Login-AzureRmAccount}
    "AzureUSGovernment" {Login-AzureRmAccount -EnvironmentName AzureUSGovernment}
}
Import-Module AzureRM.Resources
Select-AzureRmSubscription -SubscriptionId $SubscriptionId

$CurrentDate = Get-Date
$EndDate = $CurrentDate.AddMonths($NoOfMonthsUntilExpired)
$keyId = (New-Guid).Guid
$certPath = Join-Path $env:TEMP ($ApplicationDisplayName + ".pfx")

$cert = New-SelfSignedCertificate -DnsName $ApplicationDisplayName -CertStoreLocation cert:\LocalMachine\My -KeyExportPolicy Exportable -Provider "Microsoft Enhanced RSA and AES Cryptographic Provider"

$certPassword = ConvertTo-SecureString $CertPlainPassword -AsPlainText -Force
Export-PfxCertificate -Cert ("Cert:\localmachine\my\" + $cert.Thumbprint) -FilePath $certPath -Password $certPassword -Force | Write-Verbose

$pfxCert = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Certificate -ArgumentList @($certPath, $certPlainPassword)
$keyValue = [System.Convert]::ToBase64String($pfxCert.GetRawCertData())

$keyCredential = New-Object Microsoft.Azure.Commands.Resources.Models.ActiveDirectory.PSADKeyCredential
$keyCredential.StartDate = $CurrentDate
$keyCredential.EndDate= $EndDate
$keyCredential.KeyId = $keyId
##$keyCredential.Type = "AsymmetricX509Cert"
##$keyCredential.Usage = "Verify"
$keyCredential.CertValue = $keyValue

# Use Key credentials
$application = New-AzureRmADApplication -DisplayName $ApplicationDisplayName -HomePage ("http://" + $ApplicationDisplayName) -IdentifierUris ("http://" + $keyId) -KeyCredentials $keyCredential
```

```

New-AzureRMADServicePrincipal -ApplicationId $Application.ApplicationId | Write-Verbose
Get-AzureRmADServicePrincipal | Where {$_.ApplicationId -eq $Application.ApplicationId} | Write-Verbose

$NewRole = $null
$Retries = 0;
While ($NewRole -eq $null -and $Retries -le 6)
{
    # Sleep here for a few seconds to allow the service principal application to become active
    (should only take a couple of seconds normally)
    Sleep 5
    New-AzureRMRoleAssignment -RoleDefinitionName Contributor -ServicePrincipalName
    $Application.ApplicationId | Write-Verbose -ErrorAction SilentlyContinue
    Sleep 10
    $NewRole = Get-AzureRMRoleAssignment -ServicePrincipalName $Application.ApplicationId -
    ErrorAction SilentlyContinue
    $Retries++;
}

# Get the tenant id for this subscription
$SubscriptionInfo = Get-AzureRmSubscription -SubscriptionId $SubscriptionId
$TenantID = $SubscriptionInfo | Select TenantId -First 1

# Create the automation resources
New-AzureRmAutomationCertificate -ResourceGroupName $ResourceGroup -AutomationAccountName
$AutomationAccountName -Path $CertPath -Name AzureRunAsCertificate -Password $CertPassword -
Exportable | write-verbose

# Create a Automation connection asset named AzureRunAsConnection in the Automation account. This
connection uses the service principal.
$ConnectionAssetName = "AzureRunAsConnection"
Remove-AzureRmAutomationConnection -ResourceGroupName $ResourceGroup -AutomationAccountName
$AutomationAccountName -Name $ConnectionAssetName -Force -ErrorAction SilentlyContinue
$ConnectionFieldValues = @{"ApplicationId" = $Application.ApplicationId; "TenantId" =
$TenantID.TenantId; "CertificateThumbprint" = $Cert.Thumbprint; "SubscriptionId" = $SubscriptionId}
New-AzureRmAutomationConnection -ResourceGroupName $ResourceGroup -AutomationAccountName
$AutomationAccountName -Name $ConnectionAssetName -ConnectionTypeName AzureServicePrincipal -
ConnectionFieldValues $ConnectionFieldValues

```

2. On your computer, start **Windows PowerShell** from the **Start** screen with elevated user rights.
3. From the elevated PowerShell command-line shell, navigate to the folder which contains the script created in Step 1 and execute the script changing the values for parameters *-ResourceGroup*, *-AutomationAccountName*, *-ApplicationDisplayName*, *-SubscriptionId*, *-CertPlainPassword*, and *-Environment*.

NOTE

You will be prompted to authenticate with Azure after you execute the script. You must log in with an account that is a member of the Subscription Admins role and co-admin of the subscription.

```

.\New-AzureServicePrincipal.ps1 -ResourceGroup <ResourceGroupName>
-AutomationAccountName <NameofAutomationAccount> ` 
-ApplicationDisplayName <DisplayNameofAutomationAccount> ` 
-SubscriptionId <SubscriptionId> ` 
-CertPlainPassword "<StrongPassword>" -Environment <valid values are AzureCloud or
AzureUSGovernment>

```

After the script completes successfully, refer to the [sample code](#) below to authenticate with Resource Manager resources and validate credential configuration.

Create Classic Run As account PowerShell script

The PowerShell script below will configure the following:

- An Automation certificate asset in the specified automation account named **AzureClassicRunAsCertificate**, which holds the certificate used to authenticate your runbooks.
- An Automation connection asset in the specified automation account named **AzureClassicRunAsConnection**, which holds the subscription name, subscriptionId and certificate asset name.

The script will create a self-signed management certificate and save it to the temporary files folder on your computer under the user profile used to execute the PowerShell session - `%USERPROFILE%\AppData\Local\Temp`. After script execution, you will need to upload the Azure management certificate into the management store for the subscription the Automation account was created in. The steps below will walk you through the process of executing the script and uploading the certificate.

1. Save the following script on your computer. In this example, save it with the filename **New-AzureClassicRunAsAccount.ps1**.

```

#Requires -RunAsAdministrator
Param (
[Parameter(Mandatory=$true)]
[String] $ResourceGroup,
[Parameter(Mandatory=$true)]
[String] $AutomationAccountName,
[Parameter(Mandatory=$true)]
[String] $ApplicationDisplayName,
[Parameter(Mandatory=$true)]
[String] $SubscriptionId,
[Parameter(Mandatory=$true)]
[String] $CertPlainPassword,
[Parameter(Mandatory=$false)]
[int] $NoOfMonthsUntilExpired = 12
)

Login-AzureRmAccount
Import-Module AzureRM.Resources
$Subscription = Select-AzureRmSubscription -SubscriptionId $SubscriptionId
$SubscriptionName = $subscription.Subscription.SubscriptionName

$CurrentDate = Get-Date
$EndDate = $CurrentDate.AddMonths($NoOfMonthsUntilExpired)
$keyId = (New-Guid).Guid
$CertPath = Join-Path $env:TEMP ($ApplicationDisplayName + ".pfx")
$CertPathCer = Join-Path $env:TEMP ($ApplicationDisplayName + ".cer")

$cert = New-SelfSignedCertificate -DnsName $ApplicationDisplayName -CertStoreLocation
cert:\LocalMachine\My -KeyExportPolicy Exportable -Provider "Microsoft Enhanced RSA and AES
Cryptographic Provider"

$CertPassword = ConvertTo-SecureString $CertPlainPassword -AsPlainText -Force
Export-PfxCertificate -Cert ("Cert:\localmachine\my\" + $cert.Thumbprint) -FilePath $CertPath -
Password $CertPassword -Force | Write-Verbose
Export-Certificate -Cert ("Cert:\localmachine\my\" + $cert.Thumbprint) -FilePath $CertPathCer -Type
CERT | Write-Verbose

# Create the automation resources
$ClassicCertificateAssetName = "AzureClassicRunAsCertificate"
New-AzureRmAutomationCertificate -ResourceGroupName $ResourceGroup -AutomationAccountName
$AutomationAccountName -Path $CertPath -Name $ClassicCertificateAssetName -Password $CertPassword -
Exportable | write-verbose

# Create a Automation connection asset named AzureClassicRunAsConnection in the Automation account.
# This connection uses the ClassicCertificateAssetName.
$ConnectionAssetName = "AzureClassicRunAsConnection"
Remove-AzureRmAutomationConnection -ResourceGroupName $ResourceGroup -AutomationAccountName
$AutomationAccountName -Name $ConnectionAssetName -Force -ErrorAction SilentlyContinue
$ConnectionFieldValues = @{"SubscriptionName" = $SubscriptionName; "SubscriptionId" =
$SubscriptionId; "CertificateAssetName" = $ClassicCertificateAssetName}
New-AzureRmAutomationConnection -ResourceGroupName $ResourceGroup -AutomationAccountName
$AutomationAccountName -Name $ConnectionAssetName -ConnectionTypeName AzureClassicCertificate -
ConnectionFieldValues $ConnectionFieldValues

Write-Host -ForegroundColor red "Please upload the cert $CertPathCer to the Management store by
following the steps below."
Write-Host -ForegroundColor red "Log in to the Microsoft Azure Management portal
(https://manage.windowsazure.com) and select Settings -> Management Certificates."
Write-Host -ForegroundColor red "Then click Upload and upload the certificate $CertPathCer"

```

2. On your computer, start **Windows PowerShell** from the **Start** screen with elevated user rights.

3. From the elevated PowerShell command-line shell, navigate to the folder which contains the script created in Step 1 and execute the script changing the values for parameters *-ResourceGroup*, *-AutomationAccountName*, *-ApplicationDisplayName*, *-SubscriptionId*, and *-CertPlainPassword*.

NOTE

You will be prompted to authenticate with Azure after you execute the script. You must log in with an account that is a member of the Subscription Admins role and co-admin of the subscription.

```
.\New-AzureClassicRunAsAccount.ps1 -ResourceGroup <ResourceGroupName>
-AutomationAccountName <NameofAutomationAccount> ` 
-ApplicationDisplayName <DisplayNameofAutomationAccount> ` 
-SubscriptionId <SubscriptionId> ` 
-CertPlainPassword "<StrongPassword>"
```

After the script completes successfully, you will need to copy the certificate created in your user profile **Temp** folder. Follow the steps for [uploading a management API certificate](#) to the Azure classic portal and then refer to the [sample code](#) to validate credential configuration with Service Management resources.

Sample code to authenticate with Resource Manager resources

You can use the updated sample code below, taken from the **AzureAutomationTutorialScript** example runbook, to authenticate using the Run As account to manage Resource Manager resources with your runbooks.

```
$connectionName = "AzureRunAsConnection"
$subId = Get-AutomationVariable -Name 'SubscriptionId'
try
{
    # Get the connection "AzureRunAsConnection "
    $servicePrincipalConnection=Get-AutomationConnection -Name $connectionName

    "Signing in to Azure..."
    Add-AzureRmAccount ` 
        -ServicePrincipal ` 
        -TenantId $servicePrincipalConnection.TenantId ` 
        -ApplicationId $servicePrincipalConnection.ApplicationId ` 
        -CertificateThumbprint $servicePrincipalConnection.CertificateThumbprint
    "Setting context to a specific subscription"
    Set-AzureRmContext -SubscriptionId $subId
}
catch {
    if (!$servicePrincipalConnection)
    {
        $errorMessage = "Connection $connectionName not found."
        throw $errorMessage
    } else{
        Write-Error -Message $_.Exception
        throw $_.Exception
    }
}
```

The script includes two additional lines of code to support referencing a subscription context so you can easily work between multiple subscriptions. A variable asset named *SubscriptionId* contains the ID of the subscription, and after the *Add-AzureRmAccount* cmdlet statement, the [Set-AzureRmContext cmdlet](#) is stated with the parameter set *-SubscriptionId*. If the variable name is too generic, you can revise the name of the variable to include a prefix or other naming convention to make it easier to identify for your purposes. Alternatively, you can use the parameter set *-SubscriptionName* instead of *-SubscriptionId* with a corresponding variable asset.

Notice the cmdlet used for authenticating in the runbook - **Add-AzureRmAccount**, uses the *ServicePrincipalCertificate* parameter set. It authenticates by using service principal certificate, not credentials.

Sample code to authenticate with Service Management resources

You can use the updated sample code below, taken from the **AzureClassicAutomationTutorialScript** example runbook, to authenticate using the Classic Run As account to manage classic resources with your runbooks.

```
$ConnectionAssetName = "AzureClassicRunAsConnection"
# Get the connection
$connection = Get-AutomationConnection -Name $connectionAssetName

# Authenticate to Azure with certificate
Write-Verbose "Get connection asset: $ConnectionAssetName" -Verbose
$Conn = Get-AutomationConnection -Name $ConnectionAssetName
if ($Conn -eq $null)
{
    throw "Could not retrieve connection asset: $ConnectionAssetName. Assure that this asset exists in the Automation account."
}

$CertificateAssetName = $Conn.CertificateAssetName
Write-Verbose "Getting the certificate: $CertificateAssetName" -Verbose
$AzureCert = Get-AutomationCertificate -Name $CertificateAssetName
if ($AzureCert -eq $null)
{
    throw "Could not retrieve certificate asset: $CertificateAssetName. Assure that this asset exists in the Automation account."
}

Write-Verbose "Authenticating to Azure with certificate." -Verbose
Set-AzureSubscription -SubscriptionName $Conn.SubscriptionName -SubscriptionId $Conn.SubscriptionID -
    Certificate $AzureCert
Select-AzureSubscription -SubscriptionId $Conn.SubscriptionID
```

Next steps

- For more information about Service Principals, refer to [Application Objects and Service Principal Objects](#).
- For more information about Role-based Access Control in Azure Automation, refer to [Role-based access control in Azure Automation](#).
- For more information about certificates and Azure services, refer to [Certificates overview for Azure Cloud Services](#)

Certificate assets in Azure Automation

2/10/2017 • 3 min to read • [Edit on GitHub](#)

Certificates can be stored securely in Azure Automation so they can be accessed by runbooks or DSC configurations using the **Get-AzureRmAutomationRmCertificate** activity for Azure Resource Manager resources. This allows you to create runbooks and DSC configurations that use certificates for authentication or adds them to Azure or third party resources.

NOTE

Secure assets in Azure Automation include credentials, certificates, connections, and encrypted variables. These assets are encrypted and stored in the Azure Automation using a unique key that is generated for each automation account. This key is encrypted by a master certificate and stored in Azure Automation. Before storing a secure asset, the key for the automation account is decrypted using the master certificate and then used to encrypt the asset.

Windows PowerShell Cmdlets

The cmdlets in the following table are used to create and manage automation certificate assets with Windows PowerShell. They ship as part of the [Azure PowerShell module](#) which is available for use in Automation runbooks and DSC configurations.

CMDLETS	DESCRIPTION
Get-AzureRmAutomationCertificate	Retrieves information about a certificate to use in a runbook or DSC configuration. You can only retrieve the certificate itself from Get-AutomationCertificate activity.
New-AzureRmAutomationCertificate	Creates a new certificate into Azure Automation.
Remove-AzureRmAutomationCertificate	Removes a certificate from Azure Automation.
Set-AzureRmAutomationCertificate	Sets the properties for an existing certificate including uploading the certificate file and setting the password for a .pfx.
Add-AzureCertificate	Uploads a service certificate for the specified cloud service.

Creating a new certificate

When you create a new certificate, you upload a .cer or .pfx file to Azure Automation. If you mark the certificate as exportable, then you can transfer it out of the Azure Automation certificate store. If it is not exportable, then it can only be used for signing within the runbook or DSC configuration.

To create a new certificate with the Azure portal

1. From your Automation account, click the **Assets** tile to open the **Assets** blade.
2. Click the **Certificates** tile to open the **Certificates** blade.
3. Click **Add a certificate** at the top of the blade.
4. Type a name for the certificate in the **Name** box.
5. Click **Select a file** under **Upload a certificate file** to browse for a .cer or .pfx file. If you select a .pfx file, specify

a password and whether it should be allowed to be exported.

6. Click **Create** to save the new certificate asset.

To create a new certificate with Windows PowerShell

The following example demonstrates how to create a new Automation certificate and mark it exportable. This imports an existing .pfx file.

```
$certName = 'MyCertificate'  
$certPath = '.\MyCert.pfx'  
$certPwd = ConvertTo-SecureString -String 'P@$$w0rd' -AsPlainText -Force  
$ResourceGroup = "ResourceGroup01"  
  
New-AzureRmAutomationCertificate -AutomationAccountName "MyAutomationAccount" -Name $certName -Path $certPath  
-Password $certPwd -Exportable -ResourceGroupName $ResourceGroup
```

Using a certificate

You must use the **Get-AutomationCertificate** activity to use a certificate. You cannot use the [Get-AzureRmAutomationCertificate](#) cmdlet since it returns information about the certificate asset but not the certificate itself.

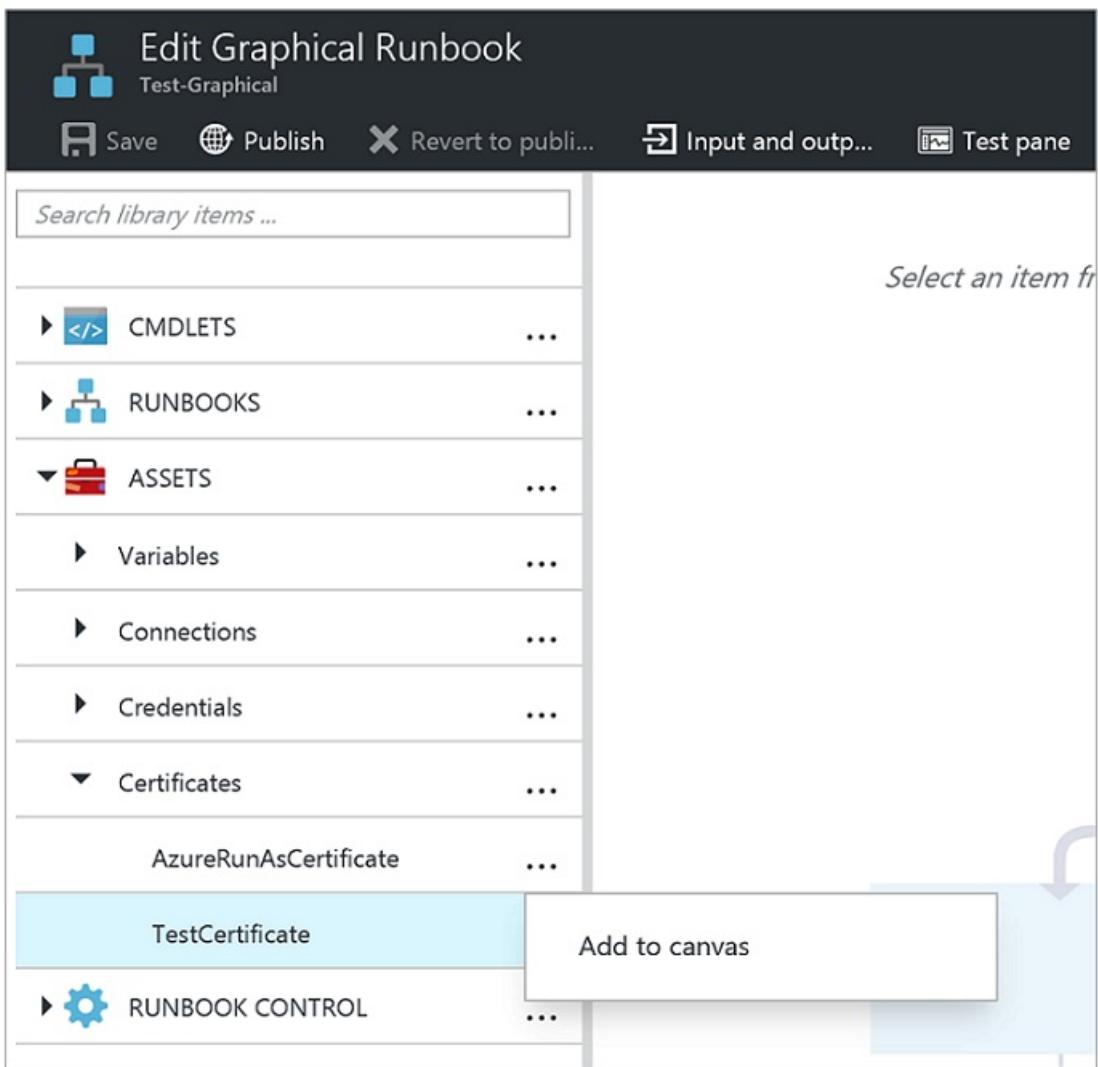
Textual runbook sample

The following sample code shows how to add a certificate to a cloud service in a runbook. In this sample, the password is retrieved from an encrypted automation variable.

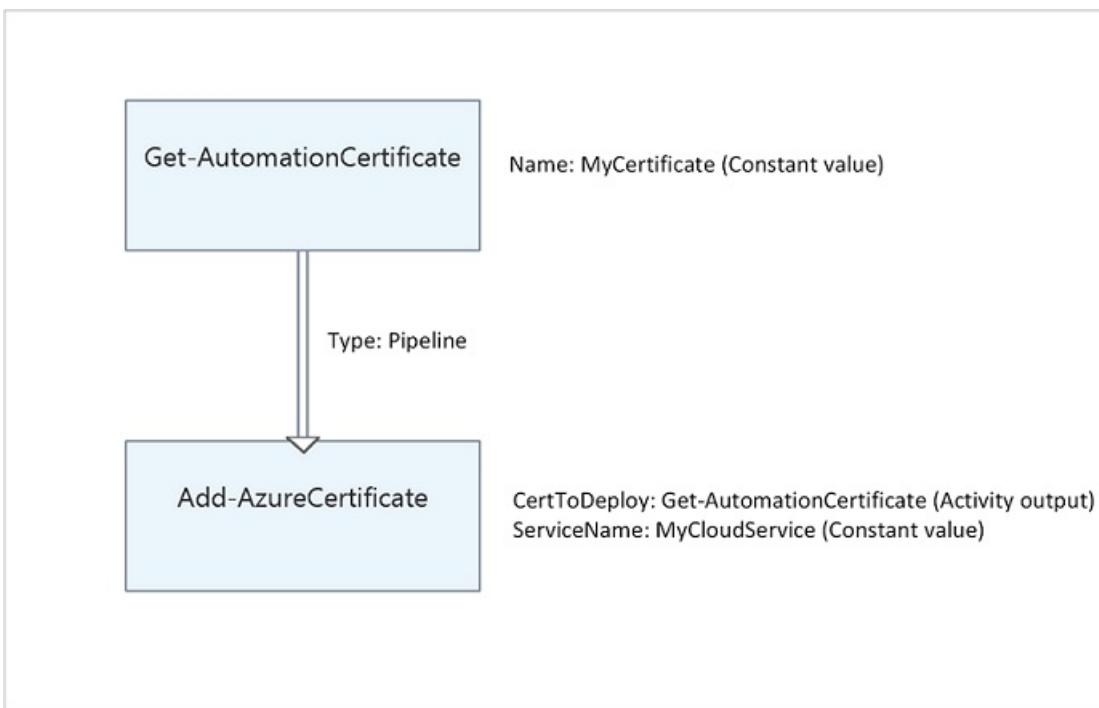
```
$serviceName = 'MyCloudService'  
$cert = Get-AutomationCertificate -Name 'MyCertificate'  
$certPwd = Get-AzureRmAutomationVariable -ResourceGroupName "ResourceGroup01" `  
-AutomationAccountName "MyAutomationAccount" -Name 'MyCertPassword'  
Add-AzureCertificate -ServiceName $serviceName -CertToDeploy $cert
```

Graphical runbook sample

You add a **Get-AutomationCertificate** to a graphical runbook by right-clicking on the certificate in the Library pane of the graphical editor and selecting **Add to canvas**.



The following image shows an example of using a certificate in a graphical runbook. This is the same example shown above for adding a certificate to a cloud service from a textual runbook.



Next steps

- To learn more about working with links to control the logical flow of activities your runbook is designed to

perform, see [Links in graphical authoring](#).

Connection assets in Azure Automation

1/17/2017 • 5 min to read • [Edit on GitHub](#)

An Automation connection asset contains the information required to connect to an external service or application from a runbook or DSC configuration. This may include information required for authentication such as a username and password in addition to connection information such as a URL or a port. The value of a connection is keeping all of the properties for connecting to a particular application in one asset as opposed to creating multiple variables. The user can edit the values for a connection in one place, and you can pass the name of a connection to a runbook or DSC configuration in a single parameter. The properties for a connection can be accessed in the runbook or DSC configuration with the **Get-AutomationConnection** activity.

When you create a connection, you must specify a *connection type*. The connection type is a template that defines a set of properties. The connection defines values for each property defined in its connection type. Connection types are added to Azure Automation in integration modules or created with the [Azure Automation API](#) if the integration module includes a connection type and is imported into your Automation account. Otherwise, you will need to create a metadata file to specify an Automation connection type. For further information regarding this, see [Integration Modules](#).

NOTE

Secure assets in Azure Automation include credentials, certificates, connections, and encrypted variables. These assets are encrypted and stored in the Azure Automation using a unique key that is generated for each automation account. This key is encrypted by a master certificate and stored in Azure Automation. Before storing a secure asset, the key for the automation account is decrypted using the master certificate and then used to encrypt the asset.

Windows PowerShell Cmdlets

The cmdlets in the following table are used to create and manage Automation connections with Windows PowerShell. They ship as part of the [Azure PowerShell module](#) which is available for use in Automation runbooks and DSC configurations.

CMDLET	DESCRIPTION
Get-AzureRmAutomationConnection	Retrieves a connection. Includes a hash table with the values of the connection's fields.
New-AzureRmAutomationConnection	Creates a new connection.
Remove-AzureRmAutomationConnection	Remove an existing connection.
Set-AzureRmAutomationConnectionFieldValue	Sets the value of a particular field for an existing connection.

Activities

The activities in the following table are used to access connections in a runbook or DSC configuration.

ACTIVITIES	DESCRIPTION
------------	-------------

ACTIVITIES	DESCRIPTION
Get-AutomationConnection	Gets a connection to use. Returns a hash table with the properties of the connection.

NOTE

You should avoid using variables with the –Name parameter of **Get- AutomationConnection** since this can complicate discovering dependencies between runbooks or DSC configurations, and connection assets at design time.

Creating a New Connection

To create a new connection with the Azure portal

1. From your automation account, click the **Assets** part to open the **Assets** blade.
2. Click the **Connections** part to open the **Connections** blade.
3. Click **Add a connection** at the top of the blade.
4. In the **Type** dropdown, select the type of connection you want to create. The form will present the properties for that particular type.
5. Complete the form and click **Create** to save the new connection.

To create a new connection with the Azure classic portal

1. From your automation account, click **Assets** at the top of the window.
2. At the bottom of the window, click **Add Setting**.
3. Click **Add Connection**.
4. In the **Connection Type** dropdown, select the type of connection you want to create. The wizard will present the properties for that particular type.
5. Complete the wizard and click the checkbox to save the new connection.

To create a new connection with Windows PowerShell

Create a new connection with Windows PowerShell using the [New-AzureRmAutomationConnection](#) cmdlet. This cmdlet has a parameter named **ConnectionFieldValues** that expects a [hash table](#) defining values for each of the properties defined by the connection type.

If you are familiar with the Automation [Run As account](#) to authenticate runbooks using the service principal, the PowerShell script, provided as an alternative to creating the the Run As account from the portal, creates a new connection asset using the following sample commands.

```
$ConnectionAssetName = "AzureRunAsConnection"
$ConnectionFieldValues = @{"ApplicationId" = $Application.ApplicationId; "TenantId" = $TenantID.TenantId;
"CertificateThumbprint" = $Cert.Thumbprint; "SubscriptionId" = $SubscriptionId}
New-AzureRmAutomationConnection -ResourceGroupName $ResourceGroup -AutomationAccountName
$AutomationAccountName -Name $ConnectionAssetName -ConnectionTypeName AzureServicePrincipal -
ConnectionFieldValues $ConnectionFieldValues
```

You are able to use the script to create the connection asset because when you create your Automation account, it automatically includes several global modules by default along with the connection type **AzurServicePrincipal** to create the **AzureRunAsConnection** connection asset. This is important to keep in mind, because if you attempt to create a new connection asset to connect to a service or application with a different authentication method, it will fail because the connection type is not already defined in your Automation account. For further information on how to create your own connection type for your custom or module from the [PowerShell Gallery](#), see [Integration Modules](#)

Using a connection in a runbook or DSC configuration

You retrieve a connection in a runbook or DSC configuration with the **Get-AutomationConnection** cmdlet. You cannot use the [Get-AzureRmAutomationConnection](#) activity. This activity retrieves the values of the different fields in the connection and returns them as a [hash table](#) which can then be used with the appropriate commands in the runbook or DSC configuration.

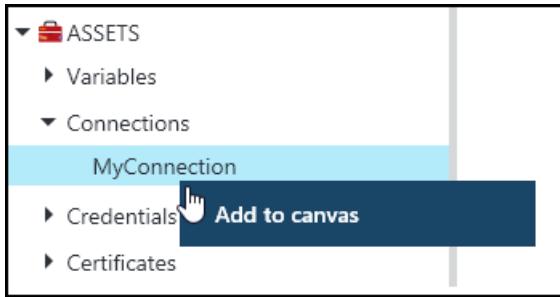
Textual runbook sample

The following sample commands show how to use the Run As account mentioned earlier, to authenticate with Azure Resource Manager resources in your runbook. It uses the connection asset representing the Run As account, which references the certificate-based service principal, not credentials.

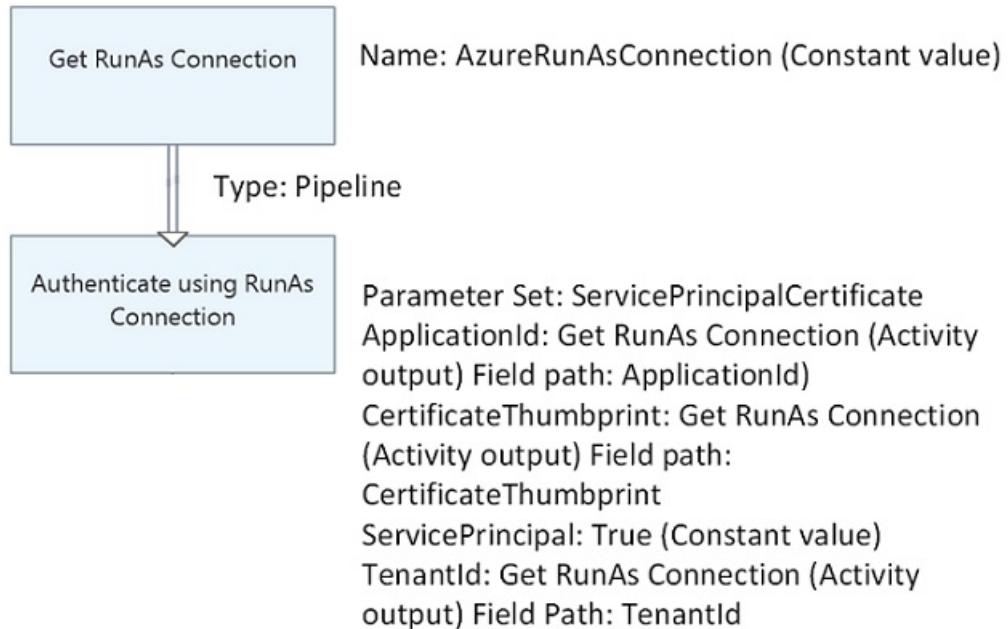
```
$Conn = Get-AutomationConnection -Name AzureRunAsConnection  
Add-AzureRMAccount -ServicePrincipal -Tenant $Conn.TenantID -ApplicationId $Conn.ApplicationID -  
CertificateThumbprint
```

Graphical runbook samples

You add a **Get-AutomationConnection** activity to a graphical runbook by right-clicking on the connection in the Library pane of the graphical editor and selecting **Add to canvas**.



The following image shows an example of using a connection in a graphical runbook. This is the same example shown above for authenticating using the Run As account with a textual runbook. This example uses the **Constant value** data set for the **Get RunAs Connection** activity that uses a connection object for authentication. A [pipeline link](#) is used here since the ServicePrincipalCertificate parameter set is expecting a single object.



Next steps

- Review [Links in graphical authoring](#) to understand how to direct and control the flow of logic in your runbooks.
- To learn more about Azure Automation's use of PowerShell modules and best practices for creating your own PowerShell modules to work as Integration Modules within Azure Automation, see [Integration Modules](#).

Credential assets in Azure Automation

1/17/2017 • 4 min to read • [Edit on GitHub](#)

An Automation credential asset holds a [PSCredential](#) object which contains security credentials such as a username and password. Runbooks and DSC configurations may use cmdlets that accept a PSCredential object for authentication, or they may extract the username and password of the PSCredential object to provide to some application or service requiring authentication. The properties for a credential are stored securely in Azure Automation and can be accessed in the runbook or DSC configuration with the [Get-AutomationPSCredential](#) activity.

NOTE

Secure assets in Azure Automation include credentials, certificates, connections, and encrypted variables. These assets are encrypted and stored in the Azure Automation using a unique key that is generated for each automation account. This key is encrypted by a master certificate and stored in Azure Automation. Before storing a secure asset, the key for the automation account is decrypted using the master certificate and then used to encrypt the asset.

Windows PowerShell cmdlets

The cmdlets in the following table are used to create and manage automation credential assets with Windows PowerShell. They ship as part of the [Azure PowerShell module](#) which is available for use in Automation runbooks and DSC configurations.

CMDLETS	DESCRIPTION
Get-AzureAutomationCredential	Retrieves information about a credential asset. You can only retrieve the credential itself from Get-AutomationPSCredential activity.
New-AzureAutomationCredential	Creates a new Automation credential.
Remove-AzureAutomationCredential	Removes an Automation credential.
Set-AzureAutomationCredential	Sets the properties for an existing Automation credential.

Runbook activities

The activities in the following table are used to access credentials in a runbook and DSC configurations.

ACTIVITIES	DESCRIPTION
Get-AutomationPSCredential	Gets a credential to use in a runbook or DSC configuration. Returns a System.Management.Automation.PSCredential object.

NOTE

You should avoid using variables in the –Name parameter of Get-AutomationPSCredential since this can complicate discovering dependencies between runbooks or DSC configurations, and credential assets at design time.

Creating a new credential asset

To create a new credential asset with the Azure portal

1. From your automation account, click the **Assets** part to open the **Assets** blade.
2. Click the **Credentials** part to open the **Credentials** blade.
3. Click **Add a credential** at the top of the blade.
4. Complete the form and click **Create** to save the new credential.

To create a new credential asset with Windows PowerShell

The following sample commands show how to create a new automation credential. A PSCredential object is first created with the name and password and then used to create the credential asset. Alternatively, you could use the **Get-Credential** cmdlet to be prompted to type in a name and password.

```
$user = "MyDomain\MyUser"  
$pw = ConvertTo-SecureString "Password!" -AsPlainText -Force  
$cred = New-Object -TypeName System.Management.Automation.PSCredential -ArgumentList $user, $pw  
New-AzureAutomationCredential -AutomationAccountName "MyAutomationAccount" -Name "MyCredential" -Value $cred
```

To create a new credential asset with the Azure classic portal

1. From your automation account, click **Assets** at the top of the window.
2. At the bottom of the window, click **Add Setting**.
3. Click **Add Credential**.
4. In the **Credential Type** dropdown, select **PowerShell Credential**.
5. Complete the wizard and click the checkbox to save the new credential.

Using a PowerShell credential

You retrieve a credential asset in a runbook or DSC configuration with the **Get-AutomationPSCredential** activity. This returns a [PSCredential object](#) that you can use with an activity or cmdlet that requires a PSCredential parameter. You can also retrieve the properties of the credential object to use individually. The object has a property for the username and the secure password, or you can use the **GetNetworkCredential** method to return a [NetworkCredential](#) object that will provide an unsecured version of the password.

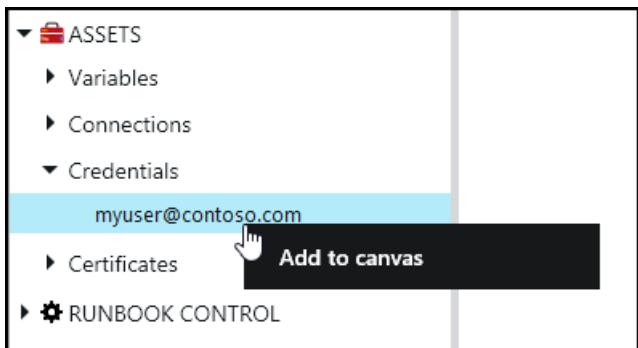
Textual runbook sample

The following sample commands show how to use a PowerShell credential in a runbook. In this example, the credential is retrieved and its username and password assigned to variables.

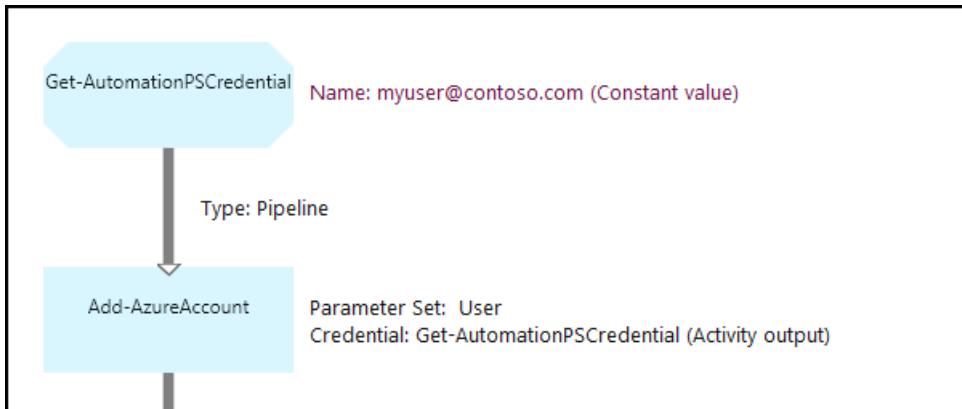
```
$myCredential = Get-AutomationPSCredential -Name 'MyCredential'  
$userName = $myCredential.UserName  
$securePassword = $myCredential.Password  
$password = $myCredential.GetNetworkCredential().Password
```

Graphical runbook sample

You add a **Get-AutomationPSCredential** activity to a graphical runbook by right-clicking on the credential in the Library pane of the graphical editor and selecting **Add to canvas**.



The following image shows an example of using a credential in a graphical runbook. In this case, it is being used to provide authentication for a runbook to Azure resources as described in [Authenticate Runbooks with Azure AD User account](#). The first activity retrieves the credential that has access to the Azure subscription. The **Add-AzureAccount** activity then uses this credential to provide authentication for any activities that come after it. A [pipeline link](#) is here since **Get-AutomationPSCredential** is expecting a single object.



Using a PowerShell credential in DSC

While DSC configurations in Azure Automation can reference credential assets using **Get-AutomationPSCredential**, credential assets can also be passed in via parameters, if desired. For more information, see [Compiling configurations in Azure Automation DSC](#).

Next Steps

- To learn more about links in graphical authoring, see [Links in graphical authoring](#)
- To understand the different authentication methods with Automation, see [Azure Automation Security](#)
- To get started with Graphical runbooks, see [My first graphical runbook](#)
- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)

Azure Automation Integration Modules

1/17/2017 • 10 min to read • [Edit on GitHub](#)

PowerShell is the fundamental technology behind Azure Automation. Since Azure Automation is built on PowerShell, PowerShell modules are key to the extensibility of Azure Automation. In this article, we will guide you through the specifics of Azure Automation's use of PowerShell modules, referred to as "Integration Modules", and best practices for creating your own PowerShell modules to make sure they work as Integration Modules within Azure Automation.

What is a PowerShell Module?

A PowerShell module is a group of PowerShell cmdlets like **Get-Date** or **Copy-Item**, that can be used from the PowerShell console, scripts, workflows, runbooks, and PowerShell DSC resources like WindowsFeature or File, that can be used from PowerShell DSC configurations. All of the functionality of PowerShell is exposed through cmdlets and DSC resources, and every cmdlet/DSC resource is backed by a PowerShell module, many of which ship with PowerShell itself. For example, the **Get-Date** cmdlet is part of the Microsoft.PowerShell.Utility PowerShell module, and **Copy-Item** cmdlet is part of the Microsoft.PowerShell.Management PowerShell module and the Package DSC resource is part of the PSDesiredStateConfiguration PowerShell module. Both of these modules ship with PowerShell. But many PowerShell modules do not ship as part of PowerShell, and are instead distributed with first or third-party products like System Center 2012 Configuration Manager or by the vast PowerShell community on places like PowerShell Gallery. The modules are useful because they make complex tasks simpler through encapsulated functionality. You can learn more about [PowerShell modules on MSDN](#).

What is an Azure Automation Integration Module?

An Integration Module isn't very different from a PowerShell module. Its simply a PowerShell module that optionally contains one additional file - a metadata file specifying an Azure Automation connection type to be used with the module's cmdlets in runbooks. Optional file or not, these PowerShell modules can be imported into Azure Automation to make their cmdlets available for use within runbooks and their DSC resources available for use within DSC configurations. Behind the scenes, Azure Automation stores these modules, and at runbook job and DSC compilation job execution time, loads them into the Azure Automation sandboxes where runbooks are executed and DSC configurations are compiled. Any DSC resources in modules are also automatically placed on the Automation DSC pull server, so that they can be pulled by machines attempting to apply DSC configurations.

We ship a number of Azure PowerShell modules out of the box in Azure Automation for you to use so you can get started automating Azure management right away, but you can import PowerShell modules for whatever system, service, or tool you want to integrate with.

NOTE

Certain modules are shipped as "global modules" in the Automation service. These global modules are available to you when you create an automation account, and we update them sometimes which automatically pushes them out to your automation account. If you don't want them to be auto-updated, you can always import the same module yourself, and that will take precedence over the global module version of that module that we ship in the service.

The format in which you import an Integration Module package is a compressed file with the same name as the module and a .zip extension. It contains the Windows PowerShell module and any supporting files, including a manifest file (.psd1) if the module has one.

If the module should contain an Azure Automation connection type, it must also contain a file with the name

<ModuleName>-Automation.json that specifies the connection type properties. This is a json file placed within the module folder of your compressed .zip file, and contains the fields of a "connection" that is required to connect to the system or service the module represents. This will end up creating a connection type in Azure Automation. Using this file you can set the field names, types, and whether the fields should be encrypted and / or optional, for the connection type of the module. The following is a template in the json file format:

```
{  
    "ConnectionFields": [  
        {  
            "IsEncrypted": false,  
            "IsOptional": false,  
            "Name": "ComputerName",  
            "TypeName": "System.String"  
        },  
        {  
            "IsEncrypted": false,  
            "IsOptional": true,  
            "Name": "Username",  
            "TypeName": "System.String"  
        },  
        {  
            "IsEncrypted": true,  
            "IsOptional": false,  
            "Name": "Password",  
            "TypeName": "System.String"  
        }],  
    "ConnectionTypeName": "DataProtectionManager",  
    "IntegrationModuleName": "DataProtectionManager"  
}
```

If you have deployed Service Management Automation and created Integration Modules packages for your automation runbooks, this should look very familiar to you.

Authoring Best Practices

Even though Integration Modules are essentially PowerShell modules, there's still a number of things we recommend you consider while authoring a PowerShell module, to make it most usable in Azure Automation. Some of these are Azure Automation specific, and some of them are useful just to make your modules work well in PowerShell Workflow, regardless of whether or not you're using Automation.

1. Include a synopsis, description, and help URI for every cmdlet in the module. In PowerShell, you can define certain help information for cmdlets to allow the user to receive help on using them with the **Get-Help** cmdlet. For example, here's how you can define a synopsis and help URI for a PowerShell module written in a .psm1 file.

```

<#
    .SYNOPSIS
        Gets all outgoing phone numbers for this Twilio account
#>
function Get-TwilioPhoneNumbers {
[CmdletBinding(DefaultParameterSetName='SpecifyConnectionFields', ` 
HelpUri='http://www.twilio.com/docs/api/rest/outgoing-caller-ids')]
param(
    [Parameter(ParameterSetName='SpecifyConnectionFields', Mandatory=$true)]
    [ValidateNotNullOrEmpty()]
    [string]
    $AccountSid,
    [Parameter(ParameterSetName='SpecifyConnectionFields', Mandatory=$true)]
    [ValidateNotNullOrEmpty()]
    [string]
    $AuthToken,
    [Parameter(ParameterSetName='UseConnectionObject', Mandatory=$true)]
    [ValidateNotNullOrEmpty()]
    [Hashtable]
    $Connection
)
$cred = CreateTwilioCredential -Connection $Connection -AccountSid $AccountSid -AuthToken $AuthToken

$uri = "$TWILIO_BASE_URL/Accounts/" + $cred.UserName + "/IncomingPhoneNumbers"

$response = Invoke-RestMethod -Method Get -Uri $uri -Credential $cred
$response.TwilioResponse.IncomingPhoneNumbers.IncomingPhoneNumber
}

```

Providing this info will not only show this help using the **Get-Help** cmdlet in the PowerShell console, it will also expose this help functionality within Azure Automation. For example, when inserting activities during runbook authoring. Clicking “View detailed help” will open the help URI in another tab of the web browser you’re using to access Azure Automation.

Activities	
13	
NAME	DESCRIPTION
Connect-WSMan	Connect-WSMan [[-ComputerName] <string>] [-Appli...]
Disable-WSManCredSSP	Disable-WSManCredSSP [-Role] <string> [<Common...]
Disconnect-WSMan	Disconnect-WSMan [[-ComputerName] <string>] [<C...]
Enable-WSManCredSSP	Enable-WSManCredSSP [-Role] <string> [[-DelegateC...
Get-WSManCredSSP	Get-WSManCredSSP [<CommonParameters>]
Get-WSManInstance	Get-WSManInstance [-ResourceURI] <uri> [-Aplicati...
Invoke-WSManAction	Invoke-WSManAction [-ResourceURI] <uri> [-Action]...

2. If the module runs against a remote system,

- It should contain an Integration Module metadata file that defines the information needed to connect to that remote system, meaning the connection type.
- Each cmdlet in the module should be able to take in a connection object (an instance of that connection type) as a parameter.

Cmdlets in the module become easier to use in Azure Automation if you allow passing an object with the

fields of the connection type as a parameter to the cmdlet. This way users don't have to map parameters of the connection asset to the cmdlet's corresponding parameters each time they call a cmdlet. Based on the runbook example above, it uses a Twilio connection asset called CorpTwilio to access Twilio and return all the phone numbers in the account. Notice how it is mapping the fields of the connection to the parameters of the cmdlet?

```
workflow Get-CorpTwilioPhones
{
    $CorpTwilio = Get-AutomationConnection -Name 'CorpTwilio'

    Get-TwilioPhoneNumbers
        -AccountSid $CorpTwilio.AccountSid
        -AuthToken $CorpTwilio.AuthToken
}
```

An easier and better way to approach this is directly passing the connection object to the cmdlet -

```
workflow Get-CorpTwilioPhones
{
    $CorpTwilio = Get-AutomationConnection -Name 'CorpTwilio'

    Get-TwilioPhoneNumbers -Connection $CorpTwilio
}
```

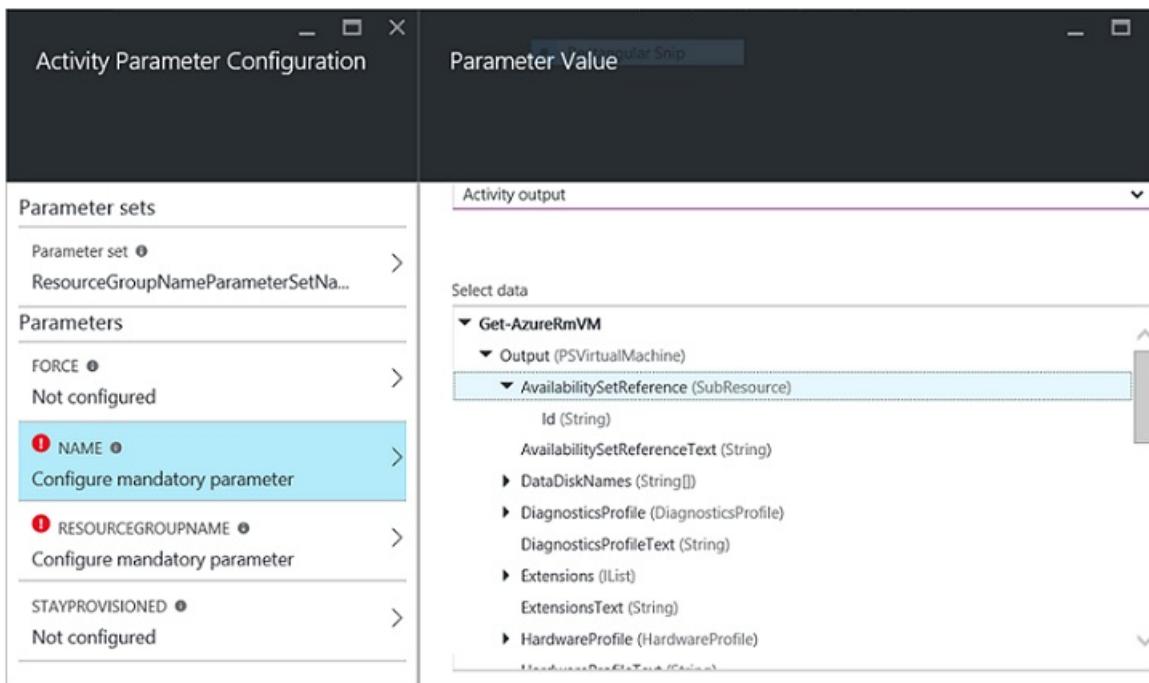
You can enable behavior like this for your cmdlets by allowing them to accept a connection object directly as a parameter, instead of just connection fields for parameters. Usually you'll want a parameter set for each, so that a user not using Azure Automation can call your cmdlets without constructing a hashtable to act as the connection object. Parameter set **SpecifyConnectionFields** below is used to pass the connection field properties one by one. **UseConnectionObject** lets you pass the connection straight through. As you can see, the [Send-TwilioSMS cmdlet](#) in the [Twilio PowerShell module](#) allows passing either way:

```
function Send-TwilioSMS {
    [CmdletBinding(DefaultParameterSetName='SpecifyConnectionFields',
    HelpUri='http://www.twilio.com/docs/api/rest/sending-sms')]
    param(
        [Parameter(ParameterSetName='SpecifyConnectionFields', Mandatory=$true)]
        [ValidateNotNullOrEmpty()]
        [string]
        $AccountSid,

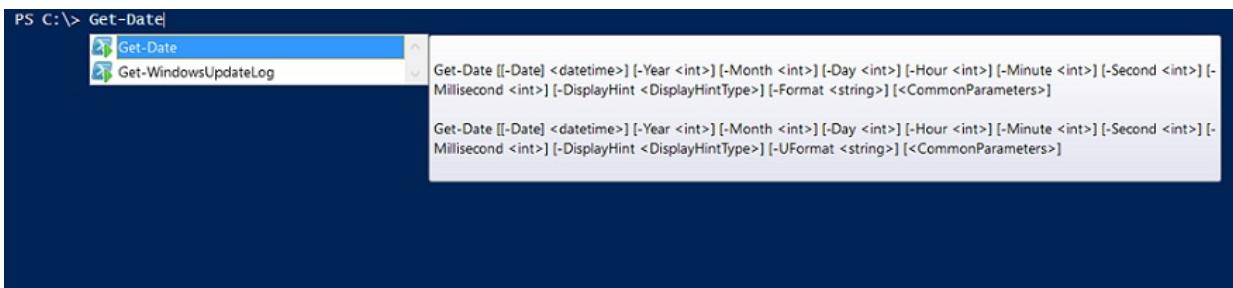
        [Parameter(ParameterSetName='SpecifyConnectionFields', Mandatory=$true)]
        [ValidateNotNullOrEmpty()]
        [string]
        $AuthToken,

        [Parameter(ParameterSetName='UseConnectionObject', Mandatory=$true)]
        [ValidateNotNullOrEmpty()]
        [Hashtable]
        $Connection
    )
}
```

3. Define output type for all cmdlets in the module. Defining an output type for a cmdlet allows design-time IntelliSense to help you determine the output properties of the cmdlet, for use during authoring. It is especially helpful during Automation runbook graphical authoring, where design time knowledge is key to an easy user experience with your module.



This is similar to the "type ahead" functionality of a cmdlet's output in PowerShell ISE without having to run it.



- Cmdlets in the module should not take complex object types for parameters. PowerShell Workflow is different from PowerShell in that it stores complex types in deserialized form. Primitive types will stay as primitives, but complex types are converted to their deserialized versions, which are essentially property bags. For example, if you used the **Get-Process** cmdlet in a runbook (or a PowerShell Workflow for that matter), it would return an object of type [Deserialized.System.Diagnostic.Process], not the expected [System.Diagnostic.Process] type. This type has all the same properties as the non-deserialized type, but none of the methods. And if you try to pass this value as a parameter to a cmdlet, where the cmdlet expects a [System.Diagnostic.Process] value for this parameter, you'll receive the following error: *Cannot process argument transformation on parameter 'process'. Error: "Cannot convert the "System.Diagnostics.Process (CcmExec)" value of type "Deserialized.System.Diagnostics.Process" to type "System.Diagnostics.Process".* This is because there is a type mismatch between the expected [System.Diagnostic.Process] type and the given [Deserialized.System.Diagnostic.Process] type. The way around this issue is to ensure the cmdlets of your module do not take complex types for parameters. Here is the wrong way to do it.

```
function Get-ProcessDescription {
    param (
        [System.Diagnostic.Process] $process
    )
    $process.Description
}
```

And here is the right way, taking in a primitive that can be used internally by the cmdlet to grab the complex object and use it. Since cmdlets execute in the context of PowerShell, not PowerShell Workflow, inside the cmdlet \$process becomes the correct [System.Diagnostic.Process] type.

```

function Get-ProcessDescription {
    param (
        [String] $processName
    )
    $process = Get-Process -Name $processName

    $process.Description
}

```

Connection assets in runbooks are hashtables, which are a complex type, and yet these hashtables seem to be able to be passed into cmdlets for their –Connection parameter perfectly, with no cast exception.

Technically, some PowerShell types are able to cast properly from their serialized form to their deserialized form, and hence can be passed into cmdlets for parameters accepting the non-deserialized type. Hashtable is one of these. It's possible for a module author's defined types to be implemented in a way that they can correctly deserialize as well, but there are some trade-offs to consider. The type needs to have a default constructor, have all of its properties public, and have a PTypeConverter. However, for already-defined types that the module author does not own, there is no way to "fix" them, hence the recommendation to avoid complex types for parameters all together. Runbook Authoring tip: If for some reason your cmdlets need to take a complex type parameter, or you are using someone else's module that requires a complex type parameter, the workaround in PowerShell Workflow runbooks and PowerShell Workflows in local PowerShell, is to wrap the cmdlet that generates the complex type and the cmdlet that consumes the complex type in the same InlineScript activity. Since InlineScript executes its contents as PowerShell rather than PowerShell Workflow, the cmdlet generating the complex type would produce that correct type, not the deserialized complex type.

5. Make all cmdlets in the module stateless. PowerShell Workflow runs every cmdlet called in the workflow in a different session. This means any cmdlets that depend on session state created / modified by other cmdlets in the same module will not work in PowerShell Workflow runbooks. Here is an example of what not to do.

```

$globalNum = 0
function Set-GlobalNum {
    param(
        [int] $num
    )

    $globalNum = $num
}
function Get-GlobalNumTimesTwo {
    $output = $globalNum * 2

    $output
}

```

6. The module should be fully contained in an Xcopy-able package. Because Azure Automation modules are distributed to the Automation sandboxes when runbooks need to execute, they need to work independently of the host they are running on. What this means is that you should be able to Zip up the module package, move it to any other host with the same or newer PowerShell version, and have it function as normal when imported into that host's PowerShell environment. In order for that to happen, the module should not depend on any files outside the module folder (the folder that gets zipped up when importing into Azure Automation), or on any unique registry settings on a host, such as those set by the install of a product. If this best practice is not followed, the module will not be usable in Azure Automation.

Next steps

- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)
- To learn more about creating PowerShell Modules, see [Writing a Windows PowerShell Module](#)

Scheduling a runbook in Azure Automation

1/17/2017 • 7 min to read • [Edit on GitHub](#)

To schedule a runbook in Azure Automation to start at a specified time, you link it to one or more schedules. A schedule can be configured to either run once or on a reoccurring hourly or daily schedule for runbooks in the Azure classic portal and for runbooks in the Azure portal, you can additionally schedule them for weekly, monthly, specific days of the week or days of the month, or a particular day of the month. A runbook can be linked to multiple schedules, and a schedule can have multiple runbooks linked to it.

NOTE

Schedules do not currently support Azure Automation DSC configurations.

Windows PowerShell Cmdlets

The cmdlets in the following table are used to create and manage schedules with Windows PowerShell in Azure Automation. They ship as part of the [Azure PowerShell module](#).

CMDLETS	DESCRIPTION
Azure Resource Manager cmdlets	
Get-AzureRmAutomationSchedule	Retrieves a schedule.
New-AzureRmAutomationSchedule	Creates a new schedule.
Remove-AzureRmAutomationSchedule	Removes a schedule.
Set-AzureRmAutomationSchedule	Sets the properties for an existing schedule.
Get-AzureRmAutomationScheduledRunbook	Retrieves scheduled runbooks.
Register-AzureRmAutomationScheduledRunbook	Associates a runbook with a schedule.
Unregister-AzureRmAutomationScheduledRunbook	Dissociates a runbook from a schedule.
Azure Service Management cmdlets	
Get-AzureAutomationSchedule	Retrieves a schedule.
New-AzureAutomationSchedule	Creates a new schedule.
Remove-AzureAutomationSchedule	Removes a schedule.
Set-AzureAutomationSchedule	Sets the properties for an existing schedule.
Get-AzureAutomationScheduledRunbook	Retrieves scheduled runbooks.

CMDLETS	DESCRIPTION
Register-AzureAutomationScheduledRunbook	Associates a runbook with a schedule.
Unregister-AzureAutomationScheduledRunbook	Dissociates a runbook from a schedule.

Creating a schedule

You can create a new schedule for runbooks in the Azure portal, in the classic portal, or with Windows PowerShell. You also have the option of creating a new schedule when you link a runbook to a schedule using the Azure classic or Azure portal.

NOTE

When you associate a schedule with a runbook, Automation stores the current versions of the modules in your account and links them to that schedule. This means that if you had a module with version 1.0 in your account when you created a schedule and then update the module to version 2.0, the schedule will continue to use 1.0. In order to use the updated module version, you must create a new schedule.

To create a new schedule in the Azure portal

1. In the Azure portal, from your automation account, click the **Assets** tile to open the **Assets** blade.
2. Click the **Schedules** tile to open the **Schedules** blade.
3. Click **Add a schedule** at the top of the blade.
4. On the **New schedule** blade, type a **Name** and optionally a **Description** for the new schedule.
5. Select whether the schedule will run one time, or on a reoccurring schedule by selecting **Once** or **Recurrence**. If you select **Once** specify a **Start time** and then click **Create**. If you select **Recurrence**, specify a **Start time** and the frequency for how often you want the runbook to repeat - by **hour**, **day**, **week**, or by **month**. If you select **week** or **month** from the drop-down list, the **Recurrence option** will appear in the blade and upon selection, the **Recurrence option** blade will be presented and you can select the day of week if you selected **week**. If you selected **month**, you can choose by **week days** or specific days of the month on the calendar and finally, do you want to run it on the last day of the month or not and then click **OK**.

To create a new schedule in the Azure classic portal

1. In the Azure classic portal, select Automation and then then select the name of an automation account.
2. Select the **Assets** tab.
3. At the bottom of the window, click **Add Setting**.
4. Click **Add Schedule**.
5. Type a **Name** and optionally a **Description** for the new schedule.your schedule will run **One Time**, **Hourly**, **Daily**, **Weekly**, or **Monthly**.
6. Specify a **Start Time** and other options depending on the type of schedule that you selected.

To create a new schedule with Windows PowerShell

You can use the [New-AzureAutomationSchedule](#) cmdlet to create a new schedule in Azure Automation for classic runbooks, or [New-AzureRmAutomationSchedule](#) cmdlet for runbooks in the Azure portal. You must specify the start time for the schedule and the frequency it should run.

The following sample commands shows how to create a schedule for the 15th and 30th of every month using an Azure Resource Manager cmdlet.

```
$automationAccountName = "MyAutomationAccount"
$scheduleName = "Sample-MonthlyDaysOfMonthSchedule"
New-AzureRMAutomationSchedule -AutomationAccountName $automationAccountName -Name ` 
$scheduleName -StartTime "7/01/2016 15:30:00" -MonthInterval 1 ` 
-DaysOfMonth Fifteenth,Thirtieth -ResourceGroupName "ResourceGroup01"
```

The following sample commands show how to create a new schedule that runs each day at 3:30 PM starting on January 20, 2015 with an Azure Service Management cmdlet.

```
$automationAccountName = "MyAutomationAccount"
$scheduleName = "Sample-DailySchedule"
New-AzureAutomationSchedule -AutomationAccountName $automationAccountName -Name ` 
$scheduleName -StartTime "1/20/2016 15:30:00" -DayInterval 1
```

Linking a schedule to a runbook

A runbook can be linked to multiple schedules, and a schedule can have multiple runbooks linked to it. If a runbook has parameters, then you can provide values for them. You must provide values for any mandatory parameters and may provide values for any optional parameters. These values will be used each time the runbook is started by this schedule. You can attach the same runbook to another schedule and specify different parameter values.

To link a schedule to a runbook with the Azure portal

1. In the Azure portal, from your automation account, click the **Runbooks** tile to open the **Runbooks** blade.
2. Click on the name of the runbook to schedule.
3. If the runbook is not currently linked to a schedule, then you will be given the option to create a new schedule or link to an existing schedule.
4. If the runbook has parameters, you can select the option **Modify run settings (Default:Azure)** and the **Parameters** blade is presented where you can enter the information accordingly.

To link a schedule to a runbook with the Azure classic portal

1. In the Azure classic portal, select **Automation** and then click the name of an automation account.
2. Select the **Runbooks** tab.
3. Click on the name of the runbook to schedule.
4. Click the **Schedule** tab.
5. If the runbook is not currently linked to a schedule, then you will be given the option to **Link to a New Schedule** or **Link to an Existing Schedule**. If the runbook is currently linked to a schedule, click **Link** at the bottom of the window to access these options.
6. If the runbook has parameters, you will be prompted for their values.

To link a schedule to a runbook with Windows PowerShell

You can use the [Register-AzureAutomationScheduledRunbook](#) to link a schedule to a classic runbook or [Register-AzureRmAutomationScheduledRunbook](#) cmdlet for runbooks in the Azure portal. You can specify values for the runbook's parameters with the Parameters parameter. See [Starting a Runbook in Azure Automation](#) for more information on specifying parameter values.

The following sample commands show how to link a schedule to a runbook using an Azure Resource Manager cmdlet with parameters.

```

$automationAccountName = "MyAutomationAccount"
$runbookName = "Test-Runbook"
$scheduleName = "Sample-DailySchedule"
$params = @{"FirstName"="Joe";"LastName"="Smith";"RepeatCount"=2;"Show"=$true}
Register-AzureRmAutomationScheduledRunbook -AutomationAccountName $automationAccountName ` 
-Name $runbookName -ScheduleName $scheduleName -Parameters $params ` 
-ResourceGroupName "ResourceGroup01"

```

The following sample commands show how to link a schedule using an Azure Service Management cmdlet with parameters.

```

$automationAccountName = "MyAutomationAccount"
$runbookName = "Test-Runbook"
$scheduleName = "Sample-DailySchedule"
$params = @{"FirstName"="Joe";"LastName"="Smith";"RepeatCount"=2;"Show"=$true}
Register-AzureAutomationScheduledRunbook -AutomationAccountName $automationAccountName ` 
-Name $runbookName -ScheduleName $scheduleName -Parameters $params

```

Disabling a schedule

When you disable a schedule, any runbooks linked to it will no longer run on that schedule. You can manually disable a schedule or set an expiration time for schedules with a frequency when you create them. When the expiration time is reached, the schedule will be disabled.

To disable a schedule from the Azure portal

1. In the Azure portal, from your automation account, click the **Assets** tile to open the **Assets** blade.
2. Click the **Schedules** tile to open the **Schedules** blade.
3. Click the name of a schedule to open the details blade.
4. Change **Enabled** to **No**.

To disable a schedule from the Azure classic portal

You can disable a schedule in the Azure classic portal from the Schedule Details page for the schedule.

1. In the Azure classic portal, select Automation and then click the name of an automation account.
2. Select the Assets tab.
3. Click the name of a schedule to open its detail page.
4. Change **Enabled** to **No**.

To disable a schedule with Windows PowerShell

You can use the [Set-AzureAutomationSchedule](#) cmdlet to change the properties of an existing schedule for a classic runbook or [Set-AzureRmAutomationSchedule](#) cmdlet for runbooks in the Azure portal. To disable the schedule, specify **false** for the **IsEnabled** parameter.

The following sample commands show how to disable a schedule for a runbook using an Azure Resource Manager cmdlet.

```

$automationAccountName = "MyAutomationAccount"
$scheduleName = "Sample-MonthlyDaysOfMonthSchedule"
Set-AzureRmAutomationSchedule -AutomationAccountName $automationAccountName ` 
-Name $scheduleName -IsEnabled $false -ResourceGroupName "ResourceGroup01"

```

The following sample commands show how to disable a schedule using the Azure Service Management cmdlet.

```
$automationAccountName = "MyAutomationAccount"  
$scheduleName = "Sample-DailySchedule"  
Set-AzureAutomationSchedule -AutomationAccountName $automationAccountName `  
-Name $scheduleName -IsEnabled $false
```

Next steps

- To get started with runbooks in Azure Automation, see [Starting a Runbook in Azure Automation](#)

Variable assets in Azure Automation

1/17/2017 • 7 min to read • [Edit on GitHub](#)

Variable assets are values that are available to all runbooks and DSC configurations in your automation account. They can be created, modified, and retrieved from the Azure portal, Windows PowerShell, and from within a runbook or DSC configuration. Automation variables are useful for the following scenarios:

- Share a value between multiple runbooks or DSC configurations.
- Share a value between multiple jobs from the same runbook or DSC configuration.
- Manage a value from the portal or from the Windows PowerShell command line that is used by runbooks or DSC configurations, such as a set of common configuration items like specific list of VM names, a specific resource group, an AD domain name, etc.

Automation variables are persisted so that they continue to be available even if the runbook or DSC configuration fails. This also allows a value to be set by one runbook that is then used by another, or is used by the same runbook or DSC configuration the next time that it is run.

When a variable is created, you can specify that it be stored encrypted. When a variable is encrypted, it is stored securely in Azure Automation, and its value cannot be retrieved from the [Get-AzureRmAutomationVariable](#) cmdlet that ships as part of the Azure PowerShell module. The only way that an encrypted value can be retrieved is from the **Get-AutomationVariable** activity in a runbook or DSC configuration.

NOTE

Secure assets in Azure Automation include credentials, certificates, connections, and encrypted variables. These assets are encrypted and stored in the Azure Automation using a unique key that is generated for each automation account. This key is encrypted by a master certificate and stored in Azure Automation. Before storing a secure asset, the key for the automation account is decrypted using the master certificate and then used to encrypt the asset.

Variable types

When you create a variable with the Azure portal, you must specify a data type from the drop-down list so the portal can display the appropriate control for entering the variable value. The variable is not restricted to this data type, but you must set the variable using Windows PowerShell if you want to specify a value of a different type. If you specify **Not defined**, then the value of the variable will be set to **\$null**, and you must set the value with the [Set-AzureAutomationVariable](#) cmdlet or **Set-AutomationVariable** activity. You cannot create or change the value for a complex variable type in the portal, but you can provide a value of any type using Windows PowerShell. Complex types will be returned as a [PSCustomObject](#).

You can store multiple values to a single variable by creating an array or hashtable and saving it to the variable.

The following are a list of variable types available in Automation:

- String
- Integer
- DateTime
- Boolean
- Null

Cmdlets and workflow activities

The cmdlets in the following table are used to create and manage Automation variables with Windows PowerShell. They ship as part of the [Azure PowerShell module](#) which is available for use in Automation runbooks and DSC configuration.

CMDLETS	DESCRIPTION
Get-AzureRmAutomationVariable	Retrieves the value of an existing variable.
New-AzureRmAutomationVariable	Creates a new variable and sets its value.
Remove-AzureRmAutomationVariable	Removes an existing variable.
Set-AzureRmAutomationVariable	Sets the value for an existing variable.

The workflow activities in the following table are used to access Automation variables in a runbook. They are only available for use in a runbook or DSC configuration, and do not ship as part of the Azure PowerShell module.

WORKFLOW ACTIVITIES	DESCRIPTION
Get-AutomationVariable	Retrieves the value of an existing variable.
Set-AutomationVariable	Sets the value for an existing variable.

NOTE

You should avoid using variables in the `-Name` parameter of **Get-AutomationVariable** in a runbook or DSC configuration since this can complicate discovering dependencies between runbooks or DSC configuration, and Automation variables at design time.

Creating a new Automation variable

To create a new variable with the Azure portal

1. From your automation account, click **Assets** at the top of the window.
2. At the bottom of the window, click **Add Setting**.
3. Click **Add Variable**.
4. Complete the wizard and click the checkbox to save the new variable.

To create a new variable with the Azure portal

1. From your automation account, click the **Assets** part to open the **Assets** blade.
2. Click the **Variables** part to open the **Variables** blade.
3. Click **Add a variable** at the top of the blade.
4. Complete the form and click **Create** to save the new variable.

To create a new variable with Windows PowerShell

The [New-AzureRmAutomationVariable](#) cmdlet creates a new variable and sets its initial value. You can retrieve the value using [Get-AzureRmAutomationVariable](#). If the value is a simple type, then that same type is returned. If it's a complex type, then a **PSCustomObject** is returned.

The following sample commands show how to create a variable of type string and then return its value.

```

New-AzureRmAutomationVariable -ResourceGroupName "ResouceGroup01"
    -AutomationAccountName "MyAutomationAccount" -Name 'MyStringVariable' ` 
    -Encrypted $false -Value 'My String'
$string = (Get-AzureRmAutomationVariable -ResourceGroupName "ResouceGroup01" ` 
    -AutomationAccountName "MyAutomationAccount" -Name 'MyStringVariable').Value

```

The following sample commands show how to create a variable with a complex type and then return its properties. In this case, a virtual machine object from **Get-AzureRmVm** is used.

```

$vm = Get-AzureRmVm -ResourceGroupName "ResourceGroup01" -Name "VM01"
New-AzureRmAutomationVariable -AutomationAccountName "MyAutomationAccount" -Name "MyComplexVariable" - 
Encrypted $false -Value $vm

$vmValue = (Get-AzureRmAutomationVariable -ResourceGroupName "ResourceGroup01" ` 
    -AutomationAccountName "MyAutomationAccount" -Name "MyComplexVariable").Value
$vmName = $vmValue.Name
$vmIpAddress = $vmValue.IpAddress

```

Using a variable in a runbook or DSC configuration

Use the **Set-AutomationVariable** activity to set the value of an Automation variable in a runbook or DSC configuration, and the **Get-AutomationVariable** to retrieve it. You shouldn't use the **Set-AzureAutomationVariable** or **Get-AzureAutomationVariable** cmdlets in a runbook or DSC configuration since they are less efficient than the workflow activities. You also cannot retrieve the value of secure variables with **Get-AzureAutomationVariable**. The only way to create a new variable from within a runbook or DSC configuration is to use the [New-AzureAutomationVariable](#) cmdlet.

Textual runbook samples

Setting and retrieving a simple value from a variable

The following sample commands show how to set and retrieve a variable in a textual runbook. In this sample, it is assumed that variables of type integer named *NumberOfIterations* and *NumberOfRunnings* and a variable of type string named *SampleMessage* have already been created.

```

$NumberOfIterations = Get-AutomationVariable -Name 'NumberOfIterations'
$NumberOfRunnings = Get-AutomationVariable -Name 'NumberOfRunnings'
$SampleMessage = Get-AutomationVariable -Name 'SampleMessage'

Write-Output "Runbook has been run $NumberOfRunnings times."

for ($i = 1; $i -le $NumberOfIterations; $i++) {
    Write-Output "$i`: $SampleMessage"
}
Set-AutomationVariable -Name NumberOfRunnings -Value ($NumberOfRunnings += 1)

```

Setting and retrieving a complex object in a variable

The following sample code shows how to update a variable with a complex value in a textual runbook. In this sample, an Azure virtual machine is retrieved with **Get-AzureVM** and saved to an existing Automation variable. As explained in [Variable types](#), this is stored as a PSCustomObject.

```

$vm = Get-AzureVM -ServiceName "MyVM" -Name "MyVM"
Set-AutomationVariable -Name "MyComplexVariable" -Value $vm

```

In the following code, the value is retrieved from the variable and used to start the virtual machine.

```
$vmObject = Get-AutomationVariable -Name "MyComplexVariable"
if ($vmObject.PowerState -eq 'Stopped') {
    Start-AzureVM -ServiceName $vmObject.ServiceName -Name $vmObject.Name
}
```

Setting and retrieving a collection in a variable

The following sample code shows how to use a variable with a collection of complex values in a textual runbook. In this sample, multiple Azure virtual machines are retrieved with **Get-AzureVM** and saved to an existing Automation variable. As explained in [Variable types](#), this is stored as a collection of PSCustomObjects.

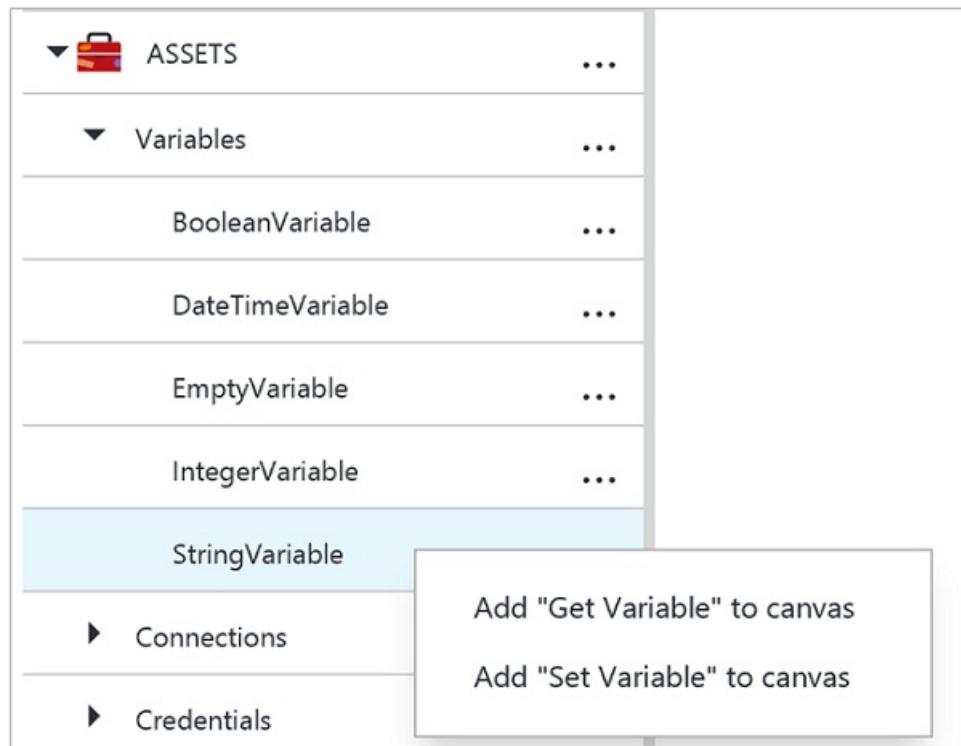
```
$vms = Get-AzureVM | Where -FilterScript {$_.Name -match "my"}
Set-AutomationVariable -Name 'MyComplexVariable' -Value $vms
```

In the following code, the collection is retrieved from the variable and used to start each virtual machine.

```
$vmValues = Get-AutomationVariable -Name "MyComplexVariable"
ForEach ($vmValue in $vmValues)
{
    if ($vmValue.PowerState -eq 'Stopped') {
        Start-AzureVM -ServiceName $vmValue.ServiceName -Name $vmValue.Name
    }
}
```

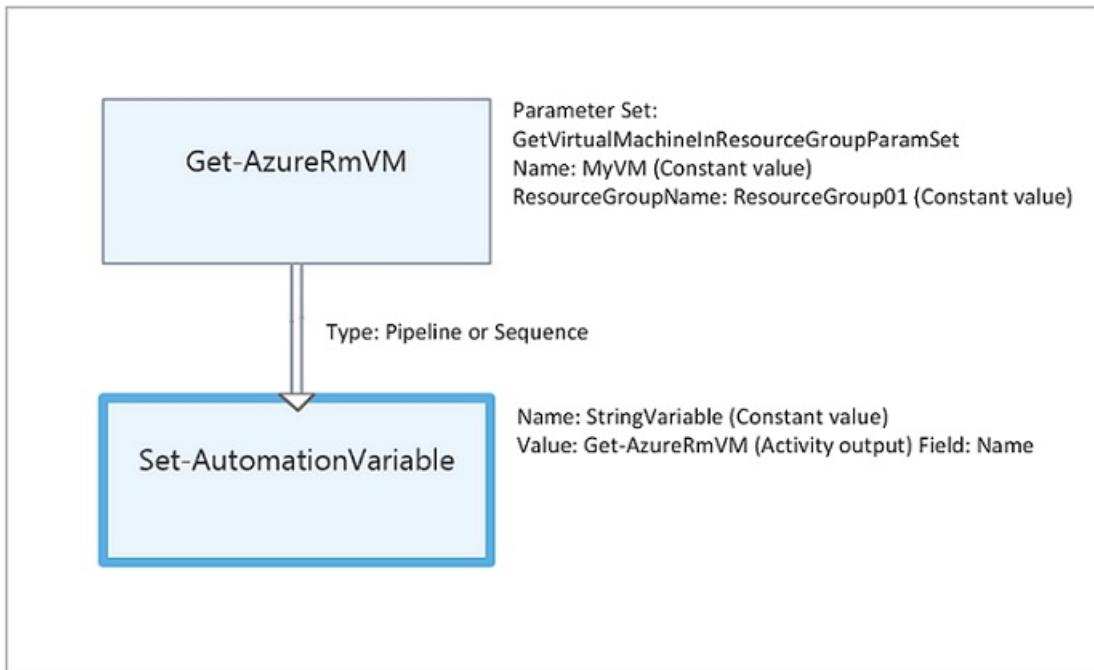
Graphical runbook samples

In a graphical runbook, you add the **Get-AutomationVariable** or **Set-AutomationVariable** by right-clicking on the variable in the Library pane of the graphical editor and selecting the activity you want.



Setting values in a variable

The following image shows sample activities to update a variable with a simple value in a graphical runbook. In this sample, a single Azure virtual machine is retrieved with **Get-AzureRmVM** and the computer name is saved to an existing Automation variable with a type of String. It doesn't matter whether the [link is a pipeline or sequence](#) since we only expect a single object in the output.



Next Steps

- To learn more about connecting activities together in graphical authoring, see [Links in graphical authoring](#)
- To get started with Graphical runbooks, see [My first graphical runbook](#)

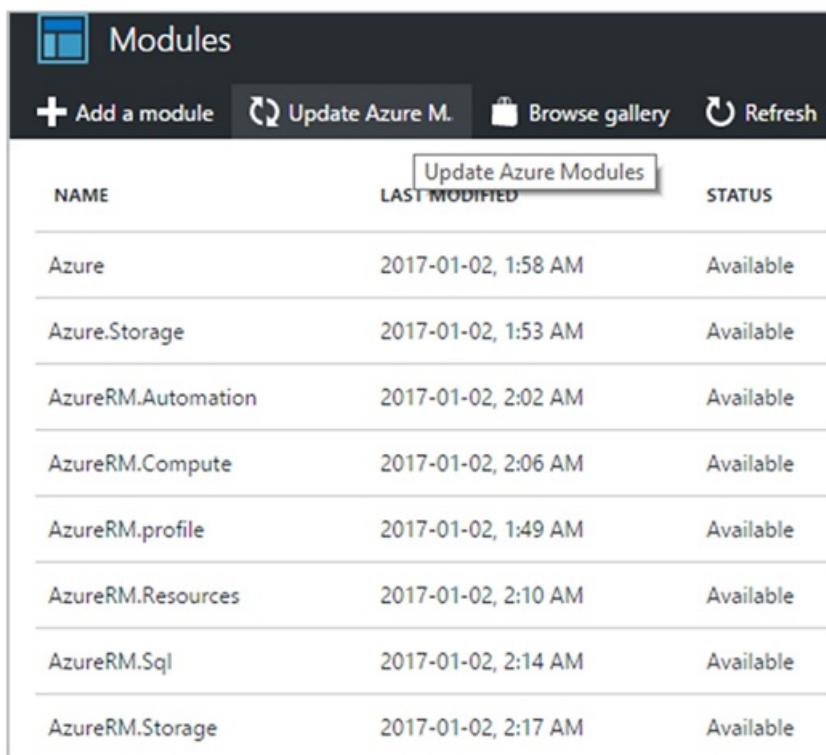
How to update Azure PowerShell modules in Azure Automation

2/14/2017 • 1 min to read • [Edit on GitHub](#)

The most common Azure PowerShell modules are provided by default in each Automation account. The Azure team updates the Azure modules regularly, so in the Automation account we provide a way for you to update the modules in the account when new versions are available.

Updating Azure Modules

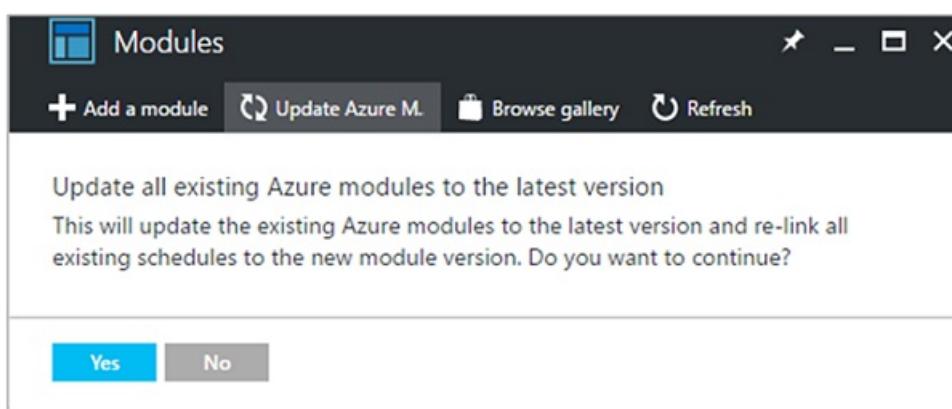
1. In the Modules blade of your Automation account there is an option called **Update Azure Modules**. It is always enabled.



A screenshot of the Azure Automation Modules blade. The top navigation bar includes 'Add a module', 'Update Azure M.', 'Browse gallery', and 'Refresh'. The main area displays a table of modules:

NAME	LAST MODIFIED	STATUS
Azure	2017-01-02, 1:58 AM	Available
Azure.Storage	2017-01-02, 1:53 AM	Available
AzureRM.Automation	2017-01-02, 2:02 AM	Available
AzureRM.Compute	2017-01-02, 2:06 AM	Available
AzureRM.profile	2017-01-02, 1:49 AM	Available
AzureRM.Resources	2017-01-02, 2:10 AM	Available
AzureRM.Sql	2017-01-02, 2:14 AM	Available
AzureRM.Storage	2017-01-02, 2:17 AM	Available

2. Click **Update Azure Modules** and you will be presented with a confirmation notification that asks you if you want to continue.



A screenshot of a confirmation dialog box. The title bar says 'Modules'. The message area contains:

Update all existing Azure modules to the latest version
This will update the existing Azure modules to the latest version and re-link all existing schedules to the new module version. Do you want to continue?

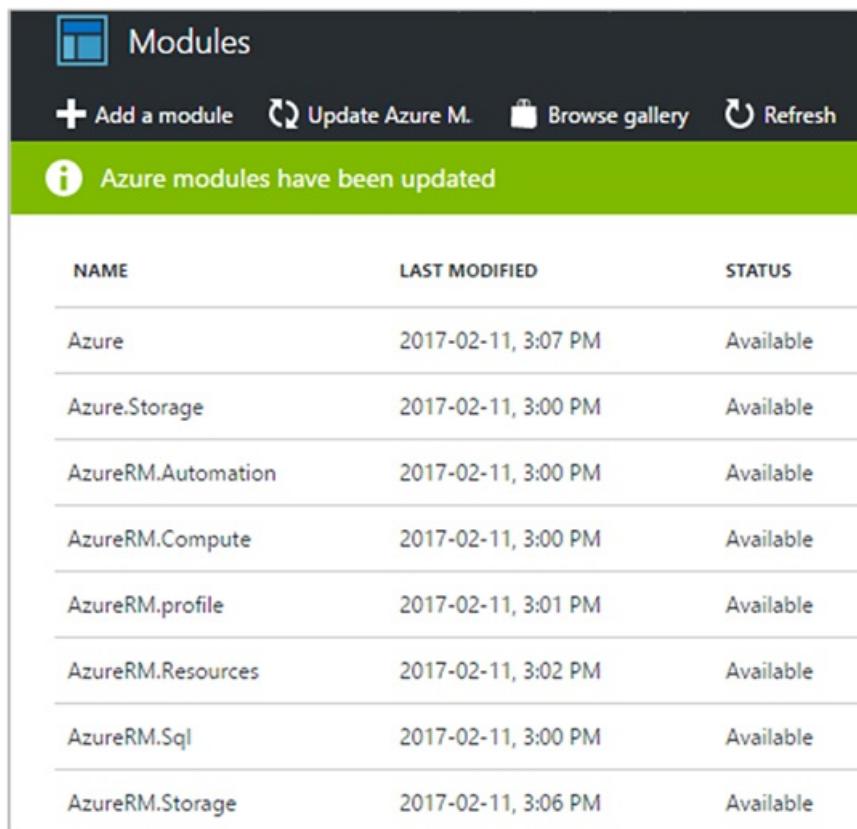
At the bottom are two buttons: 'Yes' and 'No'.

3. Click **Yes** and the module update process will begin. The update process takes about 15-20 minutes to

update the following modules:

- Azure
- Azure.Storage
- AzureRm.Automation
- AzureRm.Compute
- AzureRm.Profile
- AzureRm.Resources
- AzureRm.Sql
- AzureRm.Storage

If the modules are already up to date, then the process will complete in a few seconds. When the update process completes you will be notified.



The screenshot shows the Azure Modules management interface. At the top, there's a navigation bar with icons for 'Add a module', 'Update Azure M.', 'Browse gallery', and 'Refresh'. Below the navigation bar, a green header bar displays the message 'Azure modules have been updated' with an information icon. The main area is a table with three columns: 'NAME', 'LAST MODIFIED', and 'STATUS'. The table lists eight modules, all of which are marked as 'Available':

NAME	LAST MODIFIED	STATUS
Azure	2017-02-11, 3:07 PM	Available
Azure.Storage	2017-02-11, 3:00 PM	Available
AzureRM.Automation	2017-02-11, 3:00 PM	Available
AzureRM.Compute	2017-02-11, 3:00 PM	Available
AzureRM.profile	2017-02-11, 3:01 PM	Available
AzureRM.Resources	2017-02-11, 3:02 PM	Available
AzureRM.Sql	2017-02-11, 3:00 PM	Available
AzureRM.Storage	2017-02-11, 3:06 PM	Available

As part of the update process, the schedules for any scheduled runbooks will be updated to use the latest module version.

If you use cmdlets from these Azure PowerShell modules in your runbooks to manage Azure resources, then you will want to perform this update process every month or so to assure that you have the latest modules.

Next steps

To learn more about Integration Modules and how to create custom modules to further integrate Automation with other systems, services, or solutions, see [Integration Modules](#).

Runbook and module galleries for Azure Automation

1/17/2017 • 5 min to read • [Edit on GitHub](#)

Rather than creating your own runbooks and modules in Azure Automation, you can access a variety of scenarios that have already been built by Microsoft and the community. You can either use these scenarios without modification or you can use them as a starting point and edit them for your specific requirements.

You can get runbooks from the [Runbook Gallery](#) and modules from the [PowerShell Gallery](#). You can also contribute to the community by sharing scenarios that you develop.

Runbooks in Runbook Gallery

The [Runbook Gallery](#) provides a variety of runbooks from Microsoft and the community that you can import into Azure Automation. You can either download a runbook from the gallery which is hosted in the [TechNet Script Center](#), or you can directly import runbooks from the gallery from either the Azure classic portal or Azure portal.

You can only import directly from the Runbook Gallery using the Azure classic portal or Azure portal. You cannot perform this function using Windows PowerShell.

NOTE

You should validate the contents of any runbooks that you get from the Runbook Gallery and use extreme caution in installing and running them in a production environment.]

To import a runbook from the Runbook Gallery with the Azure classic portal

1. In the Azure Portal, click, **New, App Services, Automation, Runbook, From Gallery**.
2. Select a category to view related runbooks, and select a runbook to view its details. When you select the runbook you want, click the right arrow button.

IMPORT RUNBOOK FROM GALLERY

Select a runbook

All (195)	Provision Azure Environment Resources from Gallery
Featured (25)	Provision Azure Environment Resources from Uploaded VHD
Tutorial (12)	Run tasks on Azure Virtual Machines using the Custom Script Extension
Utility (28)	Deploy a fully customizable MySQL Percona XtraDB Cluster
Provisioning (17)	Automated Active Directory Test Domain Deployment Runbook
Monitoring (6)	
Remediation (3)	
Patching (0)	
Backup (10)	
Disaster Recovery (13)	
VM Lifecycle Management (15)	
Change Control (2)	
Capacity Management (5)	
Compliance (0)	
Dev / Test Environments (5)	
Other (143)	

MICROSOFT POWERSHELL WORKFLOWS
 COMMUNITY POWERSHELL SCRIPTS

Provision Azure Environment Resources From Uploaded VHD

★★★★★ (4)

AUTHOR Charles Joy [MSFT]

DOWNLOADS 1169

UPDATED 10/17/2014

TAGS Microsoft Azure, Building Clouds, Provisioning

TYPE PowerShell Workflow

DESCRIPTION

This runbook creates a number of Azure Environment Resources (in sequence): Azure Affinity Group, Azure Cloud Service, Azure Storage Account, Azure Storage Container, Azure VM Image, and Azure VM. It also requires the Upload of a VHD to a specified storage container mid-process.

2 3

3. Review the contents of the runbook and note any requirements in the description. Click the right arrow button when you're done.
4. Enter the runbook details and then click the checkmark button. The runbook name will already be filled in.
5. The runbook will appear on the **Runbooks** tab for the Automation Account.

To import a runbook from the Runbook Gallery with the Azure portal

1. In the Azure Portal, open your Automation account.
2. Click on the **Runbooks** tile to open the list of runbooks.
3. Click **Browse gallery** button.



4. Locate the gallery item you want and select it to view its details.

Runbook Title	Description	Created By	Ratings	Downloads	Last Update
Connect to an Azure Virtual Machine	PowerShell Workflow Runbook This runbook sets up a connection to an Azure virtual machine. Tags: Utility, VM Lifecycle Management	SC Automation Product Team	4.5 of 5	5,382	12/22/2014
Start Windows Azure Virtual Machines on a Schedule	PowerShell Runbook Demonstrates starting a single Virtual Machine or set of Virtual Machines (using a wildcard pattern) within a Cloud Service. It does this by creating scheduled tasks to start the Virtual Machine(s) on a schedule at the time specified. Tags: VM Lifecycle Management	Windows Azure Product Team	4.69 of 5	5,499	10/16/2014
Stop Azure Virtual Machine using Azure Automation Runbook	PowerShell Workflow Runbook Demonstrates stopping all Microsoft Azure Virtual Machine in a specific Azure subscription. The script could be associated with the new Azure Automation Scheduler to stop Virtual Machines at specific time. Great for developers for saving on Azure Compute, if they forget to shut Tags: Powershell, Automation, Windows Azure Virtual Machines	Peter Selch Dahl - ProActive A	4.5 of 5	4,854	4/17/2014
Configures Secure Remote PowerShell Access to Windows Azure Virtual Machines	PowerShell Runbook This script downloads and installs the automatically generated self-signed certificate created by Windows Azure for secure Remote PowerShell access to virtual machines. Tags: Windows Azure Virtual Machines, Remote PowerShell	michaelwasham	4.88 of 5	4,850	11/4/2013
Start Azure Virtual Machine using Azure Automation Runbook	PowerShell Workflow Runbook Demonstrates starting all Microsoft Azure Virtual Machine in a specific Azure subscription by starting with the Microsoft Active Directory Domain Controllers. The script could be associated with the new Azure Automation Scheduler to start the Virtual Machines at specific time Tags: Powershell, Automation, Windows Azure Virtual Machines	Peter Selch Dahl - ProActive A	3.29 of 5	3,409	4/17/2014

5. Click on **View source project** to view the item in the [TechNet Script Center](#).
6. To import an item, click on it to view its details and then click the **Import** button.

The screenshot shows a Windows Azure Virtual Machines on a Schedule runbook in the Runbook Gallery. The title bar says "Start Windows Azure Virtual Machines on a Schedule". Below it are "View Source" and "Import" buttons. A red box highlights the "Import" button. The main content area has a summary: "Demonstrates starting a single Virtual Machine or set of Virtual Machines (using a wildcard pattern) within a Cloud Service. It does this by creating scheduled tasks to start the Virtual Machine(s) on a schedule at the time specified." It was created by "Windows Azure Product Team Scripts - Microsoft" and has a rating of 4.69 of 5, 5,499 downloads, and was last updated on 10/16/2014. A red box highlights the "View Source Project" link. The code listing starts with:

```
1 <#
2 .SYNOPSIS
3     Creates scheduled tasks to start Virtual Machines.
4 .DESCRIPTION
5     Creates scheduled tasks to start a single Virtual Machine or a set of Virtual Machines
6     (using
7     wildcard pattern syntax for the Virtual Machine name).
8 .EXAMPLE
9     Start-AzureVMsOnSchedule.ps1 -ServiceName "MyServiceName" -VMName "testmachine1" ` 
10    -TaskName "Start Test Machine 1" -At 8AM
11
12 Start-AzureVMsOnSchedule.ps1 -ServiceName "MyServiceName" -VMName "test*" ` 
13    -TaskName "Start All Test Machines" -At 8:15AM
14
15
16 param(
17     # The name of the VM(s) to start on schedule.  Can be wildcard pattern.
18     [Parameter(Mandatory = $true)]
19     [string]$VMName,
20
21     # The service name that $VMName belongs to.
22     [Parameter(Mandatory = $true)]
```

7. Optionally, change the name of the runbook and then click **OK** to import the runbook.
8. The runbook will appear on the **Runbooks** tab for the Automation Account.

Adding a runbook to the runbook gallery

Microsoft encourages you to add runbooks to the Runbook Gallery that you think would be useful to other customers. You can add a runbook by [uploading it to the Script Center](#) taking into account the following details.

- You must specify *Windows Azure* for the **Category** and *Automation* for the **Subcategory** for the runbook to be displayed in the wizard.
- The upload must be a single .ps1 or .graphrunbook file. If the runbook requires any modules, child runbooks, or assets, then you should list those in the description of the submission and in the comments section of the runbook. If you have a scenario requiring multiple runbooks, then upload each separately and list the names of the related runbooks in each of their descriptions. Make sure that you use the same tags so that they will show up in the same category. A user will have to read the description to know that other runbooks are required for the scenario to work.
- Add the tag "GraphicalPS" if you are publishing a **Graphical runbook** (not a Graphical Workflow).
- Insert either a PowerShell or PowerShell Workflow code snippet into the description using **Insert code section** icon.
- The Summary for the upload will be displayed in the Runbook Gallery results so you should provide detailed information that will help a user identify the functionality of the runbook.
- You should assign one to three of the following Tags to the upload. The runbook will be listed in the wizard under the categories that match its tags. Any tags not on this list will be ignored by the wizard. If you don't specify any matching tags, the runbook will be listed under the Other category.
 - Backup
 - Capacity Management
 - Change Control
 - Compliance
 - Dev / Test Environments

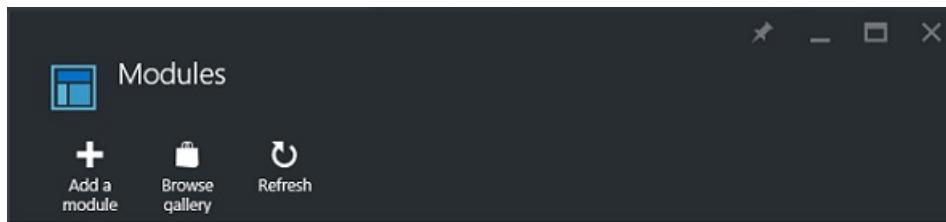
- Disaster Recovery
- Monitoring
- Patching
- Provisioning
- Remediation
- VM Lifecycle Management
- Automation updates the Gallery once an hour, so you won't see your contributions immediately.

Modules in PowerShell Gallery

PowerShell modules contain cmdlets that you can use in your runbooks, and existing modules that you can install in Azure Automation are available in the [PowerShell Gallery](#). You can launch this gallery from the Azure portal and install them directly into Azure Automation or you can download them and install them manually. You cannot install the modules directly from the Azure classic portal, but you can download them and install them as you would any other module.

To import a module from the Automation Module Gallery with the Azure portal

1. In the Azure Portal, open your Automation account.
2. Click on the **Assets** tile to open the list of assets.
3. Click on the **Modules** tile to open the list of modules.
4. Click on the **Browse gallery** button and the Browse gallery blade is launched.



5. After you have launched the Browse gallery blade, you can search by the following fields:

- Module Name
- Tags
- Author
- Cmdlet/DSC resource name

6. Locate a module that you're interested in and select it to view its details.

When you drill into a specific module, you can view more information about the module, including a link back to the PowerShell Gallery, any required dependencies, and all of the cmdlets and/or DSC resources that the module contains.

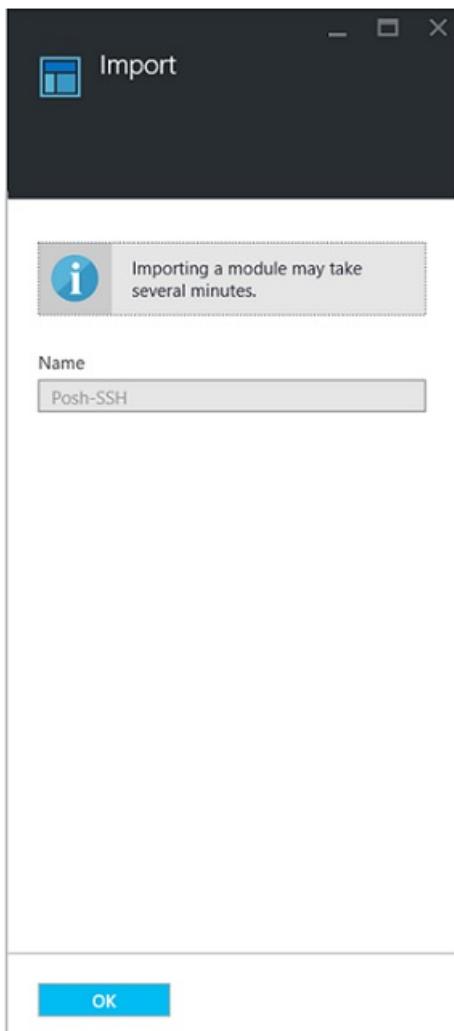
The screenshot shows the PowerShell Gallery page for the 'Posh-SSH' module. At the top, there's a dark header with the module name and a 'View Source' link. Below the header is a large 'Import' button with a downward arrow icon. The main content area has a light gray background. It starts with a brief description: 'Provide SSH functionality for executing commands against remote hosts.' To the right of this description are three status metrics: 'Version: 1.7.3', '16,273 downloads', and 'Last update: 3/1/2016'. Below the description, there's a 'Learn more' section with a 'View in PowerShell Gallery' link and a 'Licensing Information (PowerShell Gallery Default)' link. The next section is titled 'Content' and contains a table with two columns: 'TYPE' and 'NAME'. The table lists seven cmdlets:

TYPE	NAME
Cmdlet	Get-SCPFile
Cmdlet	Get-SCPFolder
Cmdlet	Get-SFTPFile
Cmdlet	Set-SFTPFile
Cmdlet	New-SFTPSession
Cmdlet	New-SSHSession
Cmdlet	Set-SCPFile

7. To install the module directly into Azure Automation, click the **Import** button.

This screenshot shows a close-up of the 'Import' button from the previous screenshot. The button is dark with a white downward arrow icon and the word 'Import' in white text. It is centered on the page below the module's description and content table.

8. When you click the Import button, you will see the module name that you are about to import. If all the dependencies are installed, the **OK** button will be active. If you are missing dependencies, you need to import those before you can import this module.
9. Click **OK** to import the module, and the module blade will launch. When Azure Automation imports a module to your account, it extracts metadata about the module and the cmdlets.



This may take a couple of minutes since each activity needs to be extracted.

10. You will receive a notification that the module is being deployed and a notification when it has completed.
11. After the module is imported, you will see the available activities, and you can use its resources in your runbooks and Desired State Configuration.

Requesting a runbook or module

You can send requests to [User Voice](#). If you need help writing a runbook or have a question about PowerShell, post a question to our [forum](#).

Next Steps

- To get started with runbooks, see [Creating or importing a runbook in Azure Automation](#)
- To understand the differences between PowerShell and PowerShell Workflow with runbooks, see [Learning PowerShell workflow](#)

Azure Automation scenario - starting and stopping virtual machines

6 min to read •

This Azure Automation scenario includes runbooks to start and stop classic virtual machines. You can use this scenario for any of the following:

- Use the runbooks without modification in your own environment.
- Modify the runbooks to perform customized functionality.
- Call the runbooks from another runbook as part of an overall solution.
- Use the runbooks as tutorials to learn runbook authoring concepts.

This is the graphical runbook version of this scenario. It is also available using [PowerShell Workflow runbooks](#).

Getting the scenario

This scenario consists of two graphical runbooks that you can download from the following links. See the [PowerShell Workflow version](#) of this scenario for links to the PowerShell Workflow runbooks.

RUNBOOK	LINK	TYPE	DESCRIPTION
StartAzureClassicVM	Start Azure Classic VM Graphical Runbook	Graphical	Starts all classic virtual machines in an Azure subscription or all virtual machines with a particular service name.
StopAzureClassicVM	Stop Azure Classic VM Graphical Runbook	Graphical	Stops all virtual machines in an automation account or all virtual machines with a particular service name.

Installing and configuring the scenario

1. Install the runbooks

After downloading the runbooks, you can import them using the procedure in [Graphical runbook procedures](#).

2. Review the description and requirements

The runbooks include an activity called **Read Me** that includes a description and required assets. You can view this information by selecting the **Read Me** activity and then the **Workflow Script** parameter. You can also get the same information from this article.

3. Configure assets

The runbooks require the following assets that you must create and populate with appropriate values. The names are default. You can use assets with different names if you specify those names in the [input parameters](#) when you start the runbook.

ASSET TYPE	DEFAULT NAME	DESCRIPTION
Credential	AzureCredential	Contains credentials for an account that has authority to start and stop virtual machines in the Azure subscription.
Variable	AzureSubscriptionId	Contains the subscription ID of your Azure subscription.

Using the scenario

Parameters

The runbooks each have the following [input parameters](#). You must provide values for any mandatory parameters and can optionally provide values for other parameters depending on your requirements.

PARAMETER	TYPE	MANDATORY	DESCRIPTION
ServiceName	string	No	If a value is provided, then all virtual machines with that service name are started or stopped. If no value is provided, then all classic virtual machines in the Azure subscription are started or stopped.
AzureSubscriptionIdAssetName	string	No	Contains the name of the variable asset that contains the subscription ID of your Azure subscription. If you don't specify a value, <i>AzureSubscriptionId</i> is used.
AzureCredentialAssetName	string	No	Contains the name of the credential asset that contains the credentials for the runbook to use. If you don't specify a value, <i>AzureCredential</i> is used.

Starting the runbooks

You can use any of the methods in [Starting a runbook in Azure Automation](#) to start either of the runbooks in this article.

The following sample commands uses Windows PowerShell to run **StartAzureClassicVM** to start all virtual machines with the service name *MyVMService*.

```
$params = @{"ServiceName"="MyVMService"}
Start-AzureAutomationRunbook -AutomationAccountName "MyAutomationAccount" -Name "StartAzureClassicVM" -
Parameters $params
```

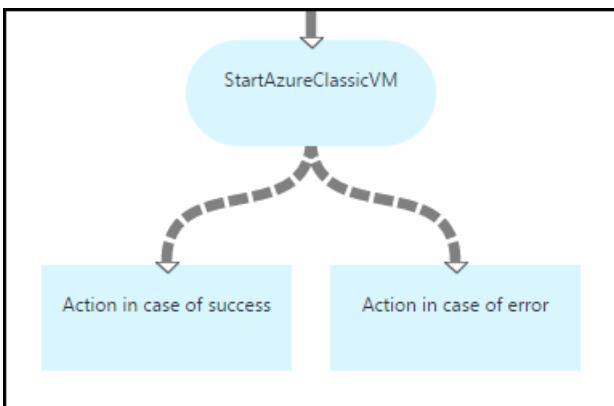
Output

The runbooks will [output a message](#) for each virtual machine indicating whether or not the start or stop instruction was successfully submitted. You can look for a specific string in the output to determine the result for each runbook. The possible output strings are listed in the following table.

RUNBOOK	CONDITION	MESSAGE
StartAzureClassicVM	Virtual machine is already running	MyVM is already running
StartAzureClassicVM	Start request for virtual machine successfully submitted	MyVM has been started
StartAzureClassicVM	Start request for virtual machine failed	MyVM failed to start
StopAzureClassicVM	Virtual machine is already running	MyVM is already stopped
StopAzureClassicVM	Start request for virtual machine successfully submitted	MyVM has been started
StopAzureClassicVM	Start request for virtual machine failed	MyVM failed to start

Following is an image of using the **StartAzureClassicVM** as a [child runbook](#) in a sample graphical runbook. This uses the conditional links in the following table.

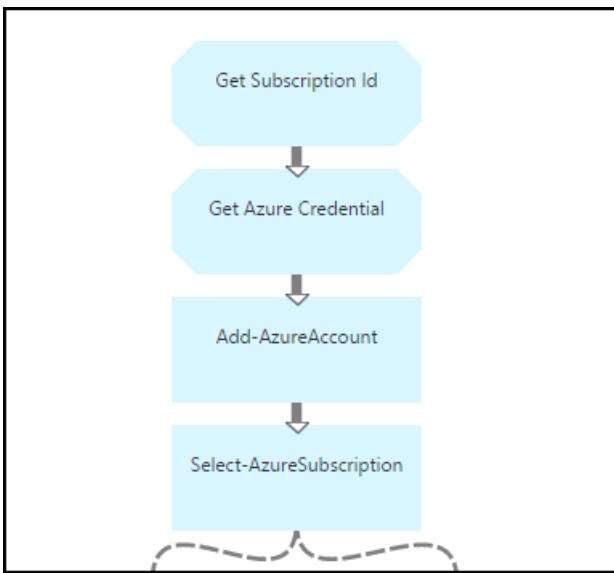
LINK	CRITERIA
Success link	\$ActivityOutput['StartAzureClassicVM'] -like "* has been started"
Error link	\$ActivityOutput['StartAzureClassicVM'] -notlike "* has been started"



Detailed breakdown

Following is a detailed breakdown of the runbooks in this scenario. You can use this information to either customize the runbooks or just to learn from them for authoring your own automation scenarios.

Authentication



The runbook starts with activities to set the [credentials](#) and Azure subscription that will be used for the rest of the runbook.

The first two activities, **Get Subscription Id** and **Get Azure Credential**, retrieve the [assets](#) that are used by the next two activities. Those activities could directly specify the assets, but they need the asset names. Since we are allowing the user to specify those names in the [input parameters](#), we need these activities to retrieve the assets with a name specified by an input parameter.

Add-AzureAccount sets the credentials that will be used for the rest of the runbook. The credential asset that it retrieves from **Get Azure Credential** must have access to start and stop virtual machines in the Azure subscription. The subscription that's used is selected by **Select-AzureSubscription** which uses the subscription Id from **Get Subscription Id**.

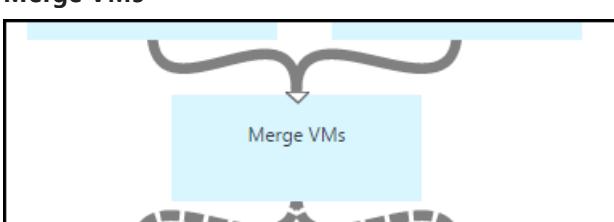
Get virtual machines



The runbook needs to determine which virtual machines it will be working with and whether they are already started or stopped (depending on the runbook). One of two activities will retrieve the VMs. **Get VMs in Service** will run if the *ServiceName* input parameter for the runbook contains a value. **Get All VMs** will run if the *ServiceName* input parameter for the runbook does not contain a value. This logic is performed by the conditional links preceding each activity.

Both activities use the **Get-AzureVM** cmdlet. **Get All VMs** uses the **ListAllVMs** parameter set to return all virtual machines. **Get VMs in Service** uses the **GetVMByServiceAndVMName** parameter set and provides the **ServiceName** input parameter for the **ServiceName** parameter.

Merge VMs



The **Merge VMs** activity is required to provide input to **Start-AzureVM** which needs the name and service name of the vm(s) to start. That input could come from either **Get All VMs** or **Get VMs in Service**, but **Start-AzureVM**

can only specify one activity for its input.

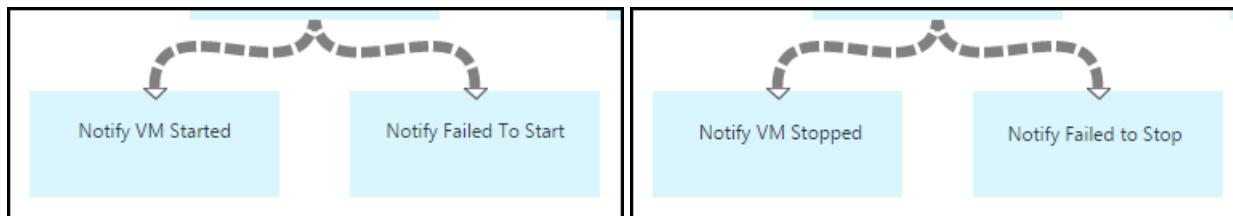
The scenario is to create **Merge VMs** which runs the **Write-Output** cmdlet. The **InputObject** parameter for that cmdlet is a PowerShell Expression that combines the input of the previous two activities. Only one of those activities will run, so only one set of output is expected. **Start-AzureVM** can use that output for its input parameters.

Start/Stop virtual machines



Depending on the runbook, the next activities attempt to start or stop the runbook using **Start-AzureVM** or **Stop-AzureVM**. Since the activity is preceded by a pipeline link, it will run once for each object returned from **Merge VMs**. The link is conditional so that the activity will only run if the *RunningState* of the virtual machine is *Stopped* for **Start-AzureVM** and *Started* for **Stop-AzureVM**. If this condition is not met, then **Notify Already Started** or **Notify Already Stopped** is run to send a message using **Write-Output**.

Send output



The final step in the runbook is to send output whether the start or stop request for each virtual machine was successfully submitted. There is a separate **Write-Output** activity for each, and we determine which one to run with conditional links. **Notify VM Started** or **Notify VM Stopped** is run if *OperationStatus* is *Succeeded*. If *OperationStatus* is any other value, then **Notify Failed To Start** or **Notify Failed to Stop** is run.

Next steps

- [Graphical authoring in Azure Automation](#)
- [Child runbooks in Azure Automation](#)
- [Runbook output and messages in Azure Automation](#)

Azure Automation scenario - starting and stopping virtual machines

6 min to read •

This Azure Automation scenario includes runbooks to start and stop classic virtual machines. You can use this scenario for any of the following:

- Use the runbooks without modification in your own environment.
- Modify the runbooks to perform customized functionality.
- Call the runbooks from another runbook as part of an overall solution.
- Use the runbooks as tutorials to learn runbook authoring concepts.

This is the PowerShell Workflow runbook version of this scenario. It is also available using [graphical runbooks](#).

Getting the scenario

This scenario consists of two PowerShell Workflow runbooks that you can download from the following links. See the [graphical version](#) of this scenario for links to the graphical runbooks.

RUNBOOK	LINK	TYPE	DESCRIPTION
Start-AzureVMs	Start Azure Classic VMs	PowerShell Workflow	Starts all classic virtual machines in an Azure subscription or all virtual machines with a particular service name.
Stop-AzureVMs	Stop Azure Classic VMs	PowerShell Workflow	Stops all virtual machines in an automation account or all virtual machines with a particular service name.

Installing and configuring the scenario

1. Install the runbooks

After downloading the runbooks, you can import them using the procedure in [Importing a Runbook](#).

2. Review the description and requirements

The runbooks include commented help text that includes a description and required assets. You can also get the same information from this article.

3. Configure assets

The runbooks require the following assets that you must create and populate with appropriate values.

ASSET TYPE	ASSET NAME	DESCRIPTION

ASSET TYPE	ASSET NAME	DESCRIPTION
Credential	AzureCredential	Contains credentials for an account that has authority to start and stop virtual machines in the Azure subscription. Alternatively, you can specify another credential asset in the Credential parameter of the Add-AzureAccount activity.
Variable	AzureSubscriptionId	Contains the subscription ID of your Azure subscription.

Using the scenario

Parameters

The runbooks each have the following parameters. You must provide values for any mandatory parameters and can optionally provide values for other parameters depending on your requirements.

PARAMETER	TYPE	MANDATORY	DESCRIPTION
ServiceName	string	No	If a value is provided, then all virtual machines with that service name are started or stopped. If no value is provided, then all classic virtual machines in the Azure subscription are started or stopped.
AzureSubscriptionIdAssetName	string	No	Contains the name of the variable asset that contains the subscription ID of your Azure subscription. If you don't specify a value, <i>AzureSubscriptionId</i> is used.
AzureCredentialAssetName	string	No	Contains the name of the credential asset that contains the credentials for the runbook to use. If you don't specify a value, <i>AzureCredential</i> is used.

Starting the runbooks

You can use any of the methods in [Starting a runbook in Azure Automation](#) to start either of the runbooks in this scenario.

The following sample commands uses Windows PowerShell to run **StartAzureVMs** to start all virtual machines with the service name *MyVMService*.

```
$params = @{"ServiceName"="MyVMService"}
Start-AzureAutomationRunbook -AutomationAccountName "MyAutomationAccount" -Name "Start-AzureVMs" -Parameters
$params
```

Output

The runbooks will [output a message](#) for each virtual machine indicating whether or not the start or stop instruction was successfully submitted. You can look for a specific string in the output to determine the result for each runbook. The possible output strings are listed in the following table.

RUNBOOK	CONDITION	MESSAGE
Start-AzureVMs	Virtual machine is already running	MyVM is already running
Start-AzureVMs	Start request for virtual machine successfully submitted	MyVM has been started
Start-AzureVMs	Start request for virtual machine failed	MyVM failed to start
Stop-AzureVMs	Virtual machine is already stopped	MyVM is already stopped
Stop-AzureVMs	Stop request for virtual machine successfully submitted	MyVM has been stopped
Stop-AzureVMs	Stop request for virtual machine failed	MyVM failed to stop

For example, the following code snippet from a runbook attempts to start all virtual machines with the service name *MyServiceName*. If any of the start requests fail, then error actions can be taken.

```
$results = Start-AzureVMs -ServiceName "MyServiceName"
foreach ($result in $results) {
    if ($result -like "* has been started" ) {
        # Action to take in case of success.
    }
    else {
        # Action to take in case of error.
    }
}
```

Detailed breakdown

Following is a detailed breakdown of the runbooks in this scenario. You can use this information to either customize the runbooks or just to learn from them for authoring your own automation scenarios.

Parameters

```
param (
    [Parameter(Mandatory=$false)]
    [String] $AzureCredentialAssetName = 'AzureCredential',
    [Parameter(Mandatory=$false)]
    [String] $AzureSubscriptionIdAssetName = 'AzureSubscriptionId',
    [Parameter(Mandatory=$false)]
    [String] $ServiceName
)
```

The workflow starts by getting the values for the [input parameters](#). If the asset names are not provided then default names are used.

Output

```
# Returns strings with status messages
[OutputType([String])]
```

This line declares that the output of the runbook will be a string. This is not required but is a best practice for when the runbook is used as a [child runbook](#) so that a parent runbook will know the output type to expect.

Authentication

```
# Connect to Azure and select the subscription to work against
$Cred = Get-AutomationPSCredential -Name $AzureCredentialAssetName
)null = Add-AzureAccount -Credential $Cred -ErrorAction Stop
$SubId = Get-AutomationVariable -Name $AzureSubscriptionIdAssetName
>null = Select-AzureSubscription -SubscriptionId $SubId -ErrorAction Stop
```

The next lines set the [credentials](#) and Azure subscription that will be used for the rest of the runbook. First we use **Get-AutomationPSCredential** to get the asset that holds the credentials with access to start and stop virtual machines in the Azure subscription. **Add-AzureAccount** then uses this asset to set the credentials. The output is assigned to a dummy variable so that it isn't included in the runbook output.

The variable asset with the subscription ID is then retrieved with **Get-AutomationVariable** and the subscription set with **Select-AzureSubscription**.

Get VMs

```
# If there is a specific cloud service, then get all VMs in the service,
# otherwise get all VMs in the subscription.
if ($ServiceName)
{
    $VMs = Get-AzureVM -ServiceName $ServiceName
}
else
{
    $VMs = Get-AzureVM
}
```

Get-AzureVM is used to retrieve the virtual machines the runbook will work with. If a value is provided in the **ServiceName** input variable, then only the virtual machines with that service name are retrieved. If **ServiceName** is empty, then all virtual machines are retrieved.

Start/Stop virtual machines and send output

```

# Start each of the stopped VMs
foreach ($VM in $VMs)
{
    if ($VM.PowerState -eq "Started")
    {
        # The VM is already started, so send notice
        Write-Output ($VM.InstanceName + " is already running")
    }
    else
    {
        # The VM needs to be started
        $StartRtn = Start-AzureVM -Name $VM.Name -ServiceName $VM.ServiceName -ErrorAction Continue

        if ($StartRtn.OperationStatus -ne 'Succeeded')
        {
            # The VM failed to start, so send notice
            Write-Output ($VM.InstanceName + " failed to start")
        }
        else
        {
            # The VM started, so send notice
            Write-Output ($VM.InstanceName + " has been started")
        }
    }
}

```

The next lines step through each virtual machine. First the **PowerState** of the virtual machine is checked to see if it is already running or stopped, depending on the runbook. If it is already in the target state, then a message is sent to output, and the runbook ends. If not, then **Start-AzureVM** or **Stop-AzureVM** is used to attempt to start or stop the virtual machine with the result of the request stored to a variable. A message is then sent to output specifying whether the request to start or stop was submitted successfully.

Next steps

- To learn more about working with child runbooks, see [Child runbooks in Azure Automation](#)
- To learn more about output messages during runbook execution and logging to help troubleshoot, see [Runbook output and messages in Azure Automation](#)

Azure Automation scenario - provision an AWS virtual machine

1/17/2017 • 5 min to read • [Edit on GitHub](#)

In this article, we demonstrate how you can leverage Azure Automation to provision a virtual machine in your Amazon Web Service (AWS) subscription and give that VM a specific name – which AWS refers to as “tagging” the VM.

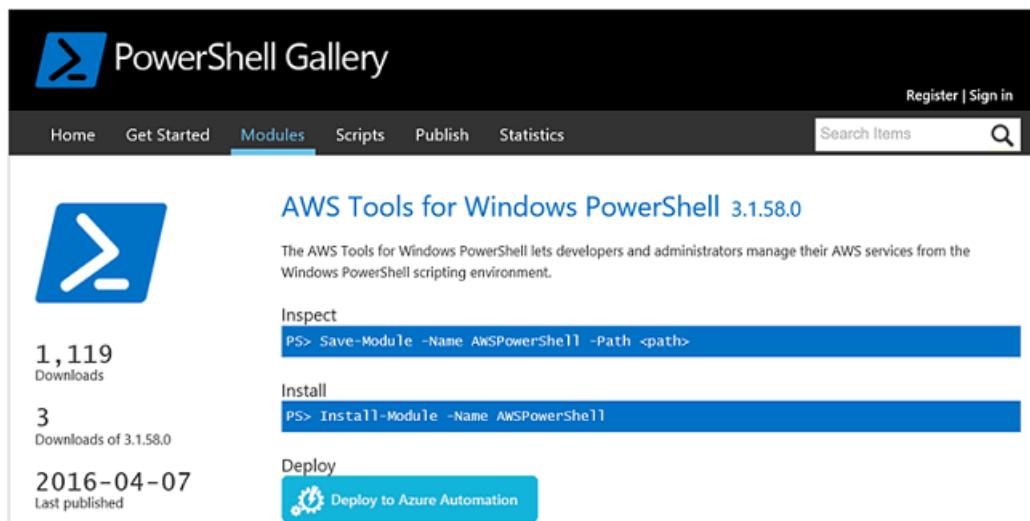
Prerequisites

For the purposes of this article, you need to have an Azure Automation account and an AWS subscription. For more information on setting up an Azure Automation account and configuring it with your AWS subscription credentials, review [Configure Authentication with Amazon Web Services](#). This account should be created or updated with your AWS subscription credentials before proceeding, as we will reference this account in the steps below.

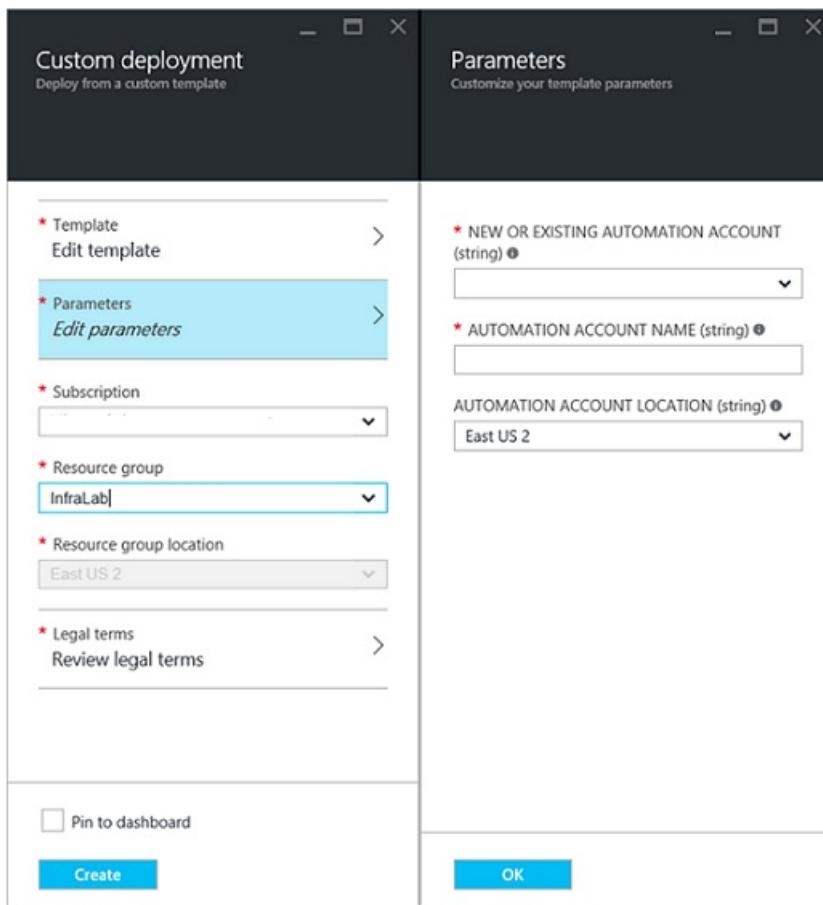
Deploy Amazon Web Services PowerShell Module

Our VM provisioning runbook will leverage the AWS PowerShell module to do its work. Perform the following steps to add the module to your Automation account that is configured with your AWS subscription credentials.

1. Open your web browser and navigate to the [PowerShell Gallery](#) and click on the **Deploy to Azure Automation button**.



2. You are taken to the Azure login page and after authenticating, you will be routed to the Azure Portal and presented with the following blade.



3. Select the Resource Group from the **Resource Group** drop-down list and on the Parameters blade, provide the following information:

- From the **New or Existing Automation Account (string)** drop-down list select **Existing**.
- In the **Automation Account Name (string)** box, type in the exact name of the Automation account that includes the credentials for your AWS subscription. For example, if you created a dedicated account named **AWSAutomation**, then that is what you type in the box.
- Select the appropriate region from the **Automation Account Location** drop-down list.

4. When you have completed entering the required information, click **Create**.

NOTE

While importing a PowerShell module into Azure Automation, it is also extracting the cmdlets and these activities will not appear until the module has completely finished importing and extracting the cmdlets. This process can take a few minutes.

5. In the Azure Portal, open your Automation account referenced in step 3.
6. Click on the **Assets** tile and on the **Assets** blade, select the **Modules** tile.
7. On the **Modules** blade you will see the **AWSPowerShell** module in the list.

Create AWS deploy VM runbook

Once the AWS PowerShell Module has been deployed, we can now author a runbook to automate provisioning a virtual machine in AWS using a PowerShell script. The steps below will demonstrate how to leverage native PowerShell script in Azure Automation.

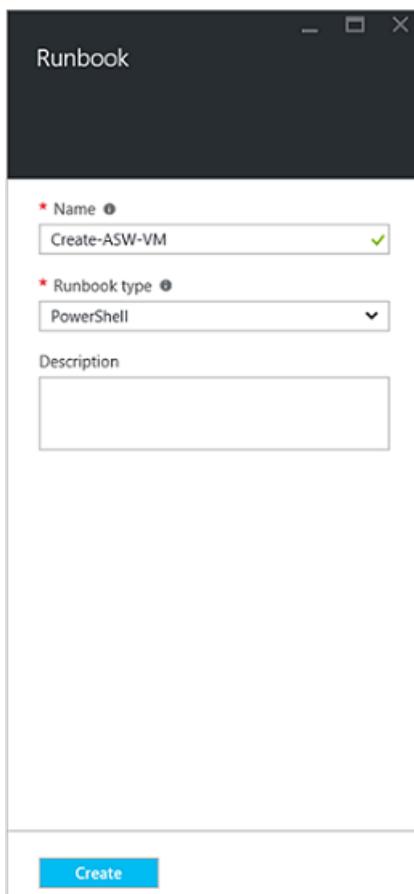
NOTE

For further options and information regarding this script, please visit the [PowerShell Gallery](#).

1. Download the PowerShell script New-AwsVM from the PowerShell Gallery by opening a PowerShell session and typing the following:

```
Save-Script -Name New-AwsVM -Path <path>
```

2. From the Azure Portal, open your Automation account and click the **Runbooks** tile.
3. From the **Runbooks** blade, select **Add a runbook**.
4. On the **Add a runbook** blade, select **Quick Create** (Create a new runbook).
5. On the **Runbook** properties blade, type a name in the Name box for your runbook and from the **Runbook type** drop-down list select **PowerShell**, and then click **Create**.



6. When the Edit PowerShell Runbook blade appears, copy and paste the PowerShell script into the runbook authoring canvas.

```

34 #>
35 #TODO:
36 #Change the default values for the following parameters if they do not match with yours:
37 $AWSRegion, $EC2ImageName, $MinCount, $MaxCount, $InstanceType
38 #Create an Azure Automation Asset called "AwsCred"
39 #Turn on Log verbose records and optionally Log progress records under the runbook settings to see verbose
40
41 param (
42     [Parameter(Mandatory=$true)]
43     [string]$VMname,
44     [ValidateNotNullOrEmpty()])
45     [string]$AWSRegion = "us-west-2",
46     [ValidateNotNullOrEmpty()])
47     [string]$EC2ImageName = "WINDOWS_2012R2_BASE",
48     [ValidateNotNullOrEmpty()])
49     [string]$MinCount = 1,
50     [ValidateNotNullOrEmpty()])
51     [string]$MaxCount = 1,
52     [ValidateNotNullOrEmpty()])
53     [string]$InstanceType = "t2.micro"
54 )
55
56 # Get credentials to authenticate against AWS
57 $AwsCred = Get-AutomationPSCredential -Name "AwsCred"
58 $AwsAccessKeyId = $AwsCred.UserName
59 $AwsSecretKey = $AwsCred.GetNetworkCredential().Password

```

NOTE

Please note the following when working with the example PowerShell script:

- The runbook contains a number of default parameter values. Please evaluate all default values and update where necessary.
- If you have stored your AWS credentials as a credential asset named differently than **AwsCred**, you will need to update the script on line 57 to match accordingly.
- When working with the AWS CLI commands in PowerShell, especially with this example runbook, you must specify the AWS region. Otherwise, the cmdlets will fail. View AWS topic [Specify AWS Region](#) in the AWS Tools for PowerShell document for further details.

7. To retrieve a list of image names from your AWS subscription, launch PowerShell ISE and import the AWS PowerShell Module. Authenticate against AWS by replacing **Get-AutomationPSCredential** in your ISE environment with **AwsCred = Get-Credential**. This will prompt you for your credentials and you can provide your **Access Key ID** for the username and **Secret Access Key** for the password. See the example below:

```

#Sample to get the AWS VM available images
#Please provide the path where you have downloaded the AWS PowerShell module
Import-Module AWSPowerShell
$AwsRegion = "us-west-2"
$AwsCred = Get-Credential
$AwsAccessKeyId = $AwsCred.UserName
$AwsSecretKey = $AwsCred.GetNetworkCredential().Password

# Set up the environment to access AWS
Set-AwsCredentials -AccessKey $AwsAccessKeyId -SecretKey $AwsSecretKey -StoreAs AWSProfile
Set-DefaultAWSRegion -Region $AwsRegion

Get-EC2ImageByName -ProfileName AWSProfile

```

The following output is returned:

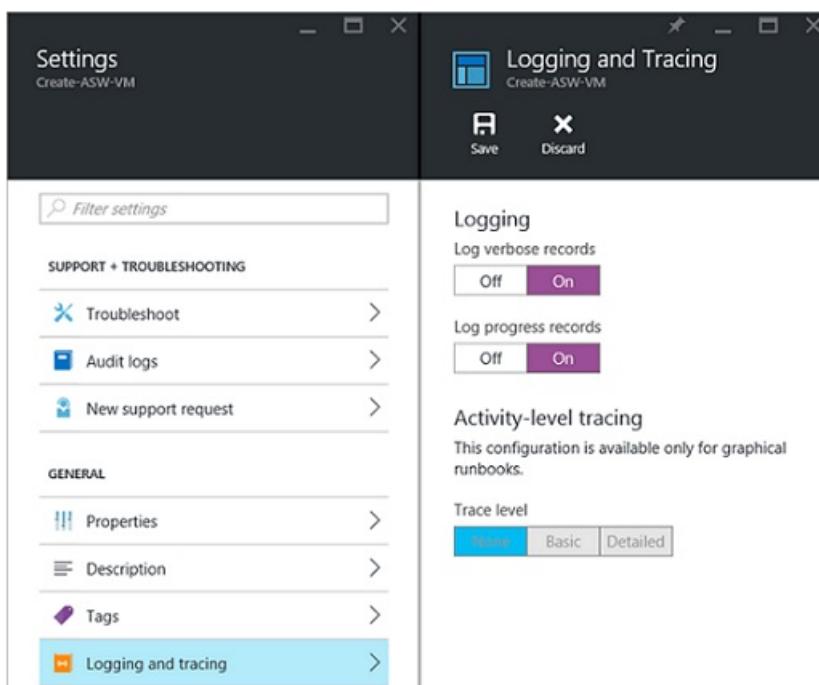
```
PS C:\> Get-EC2ImageByName -ProfileName AWSProfile
WINDOWS_2012R2_BASE
WINDOWS_2012R2_SQL_SERVER_EXPRESS_2014
WINDOWS_2012R2_SQL_SERVER_STANDARD_2014
WINDOWS_2012R2_SQL_SERVER_WEB_2014
WINDOWS_2012_BASE
WINDOWS_2012_SQL_SERVER_EXPRESS_2014
WINDOWS_2012_SQL_SERVER_STANDARD_2014
WINDOWS_2012_SQL_SERVER_WEB_2014
WINDOWS_2012_SQL_SERVER_EXPRESS_2012
WINDOWS_2012_SQL_SERVER_STANDARD_2012
WINDOWS_2012_SQL_SERVER_WEB_2012
WINDOWS_2012_SQL_SERVER_EXPRESS_2008
WINDOWS_2012_SQL_SERVER_STANDARD_2008
WINDOWS_2012_SQL_SERVER_WEB_2008
WINDOWS_2008R2_BASE
WINDOWS_2008R2_SQL_SERVER_EXPRESS_2012
WINDOWS_2008R2_SQL_SERVER_STANDARD_2012
WINDOWS_2008R2_SQL_SERVER_WEB_2012
WINDOWS_2008R2_SQL_SERVER_EXPRESS_2008
WINDOWS_2008R2_SQL_SERVER_STANDARD_2008
WINDOWS_2008R2_SQL_SERVER_WEB_2008
WINDOWS_2008RTM_BASE
WINDOWS_2008RTM_SQL_SERVER_EXPRESS_2008
WINDOWS_2008RTM_SQL_SERVER_STANDARD_2008
WINDOWS_2008_BEANSTALK_IIS75
WINDOWS_2012_BEANSTALK_IIS8
VPC_NAT
```

8. Copy and paste the one of the image names in an Automation variable as referenced in the runbook as **\$InstanceType**. Since in this example we are using the free AWS tiered subscription, we'll use **t2.micro** for our runbook example.
9. Save the runbook, then click **Publish** to publish the runbook and then **Yes** when prompted.

Testing the AWS VM runbook

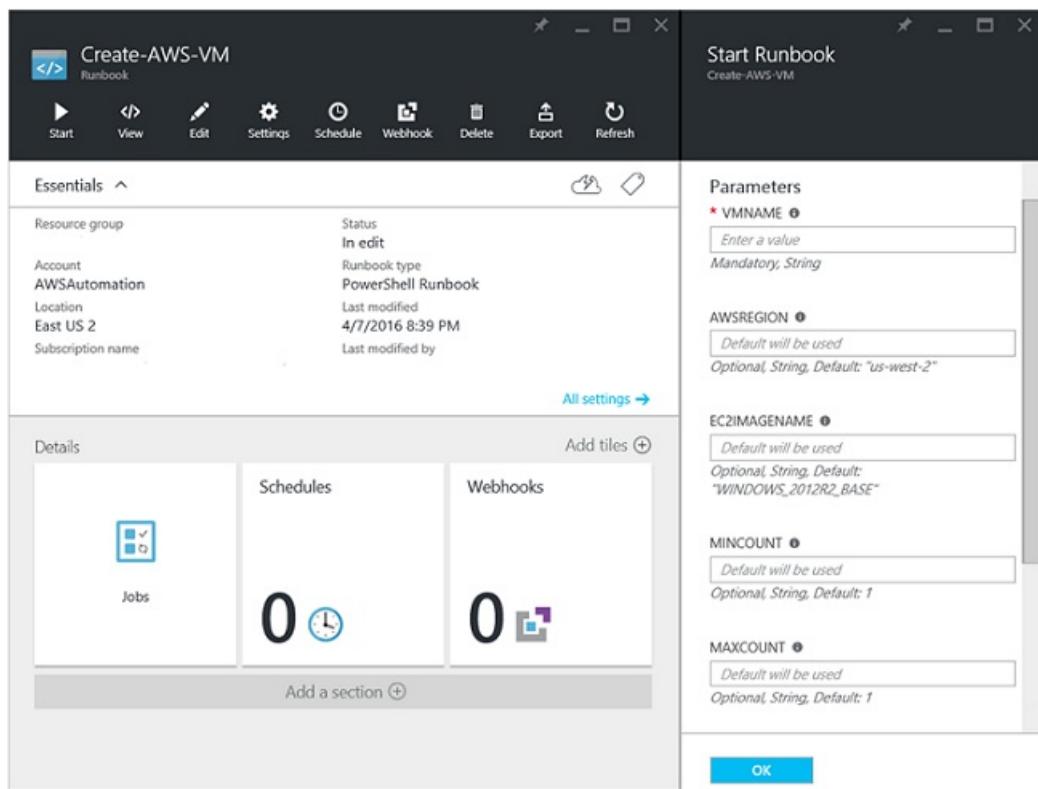
Before we proceed with testing the runbook, we need to verify a few things. Specifically:

- An asset for authenticating against AWS has been created called **AWScred** or the script has been updated to reference the name of your credential asset.
- The AWS PowerShell module has been imported in Azure Automation
- A new runbook has been created and parameter values have been verified and updated where necessary
- **Log verbose records** and optionally **Log progress records** under the runbook setting **Logging and tracing** have been set to **On**.

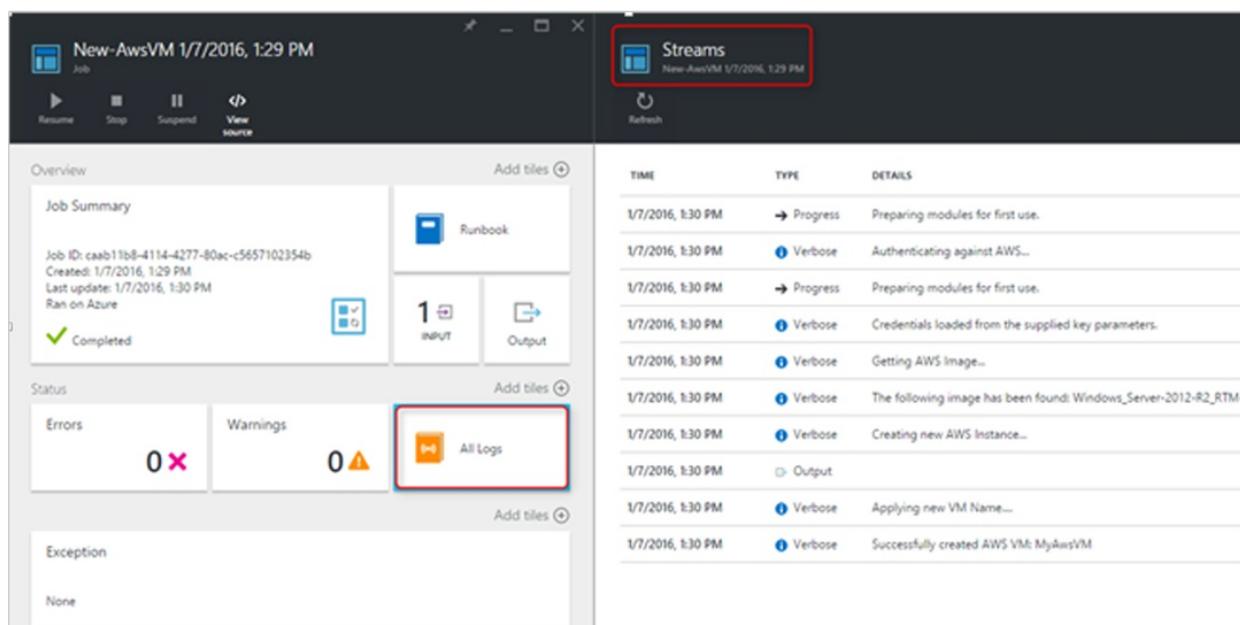


1. We want to start the runbook, so click **Start** and then click **OK** when the Start Runbook blade opens.

2. On the Start Runbook blade, provide a **VMname**. Accept the default values for the other parameters that you preconfigured in the script earlier. Click **OK** to start the runbook job.



3. A job pane is opened for the runbook job that we just created. Close this pane.
 4. We can view progress of the job and view output **Streams** by selecting the **All Logs** tile from the runbook job blade.



5. To confirm the VM is being provisioned, log into the AWS Management Console if you are not currently logged in.

EC2 Dashboard		AWS	Services	Edit	
Events					
Tags					
Reports					
Limits					
INSTANCES					
Instances					
Spot Requests					
Reserved Instances					

Launch Instance Connect Actions

Filter by tags and attributes or search by keyword

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks
Hoeba1	i-4439b59d	m1.small	us-west-2a	terminated	
MyAWSVM1	i-66ce41bf	m1.small	us-west-2a	running	Initializing
AWS-RH1	i-a89e706c	t2.micro	us-west-2b	running	2/2 checks...

Next steps

- To get started with Graphical runbooks, see [My first graphical runbook](#)
- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)
- To know more about runbook types, their advantages and limitations, see [Azure Automation runbook types](#)
- For more information on PowerShell script support feature, see [Native PowerShell script support in Azure Automation](#)

Azure Automation scenario - remediate Azure VM alerts

1/17/2017 • 7 min to read • [Edit on GitHub](#)

Azure Automation and Azure Virtual Machines have released a new feature allowing you to configure Virtual Machine (VM) alerts to run Automation runbooks. This new capability allows you to automatically perform standard remediation in response to VM alerts, like restarting or stopping the VM.

Previously, during VM alert rule creation you were able to [specify an Automation webhook](#) to a runbook in order to run the runbook whenever the alert triggered. However, this required you to do the work of creating the runbook, creating the webhook for the runbook, and then copying and pasting the webhook during alert rule creation. With this new release, the process is much easier because you can directly choose a runbook from a list during alert rule creation, and you can choose an Automation account which will run the runbook or easily create an account.

In this article, we will show you how easy it is to set up an Azure VM alert and configure an Automation runbook to run whenever the alert triggers. Example scenarios include restarting a VM when the memory usage exceeds some threshold due to an application on the VM with a memory leak, or stopping a VM when the CPU user time has been below 1% for past hour and is not in use. We'll also explain how the automated creation of a service principal in your Automation account simplifies the use of runbooks in Azure alert remediation.

Create an alert on a VM

Perform the following steps to configure an alert to launch a runbook when its threshold has been met.

NOTE

With this release, we only support V2 virtual machines and support for classic VMs will be added soon.

1. Log in to the Azure portal and click **Virtual Machines**.
2. Select one of your virtual machines. The virtual machine dashboard blade will appear and the **Settings** blade to its right.
3. From the **Settings** blade, under the Monitoring section select **Alert rules**.
4. On the **Alert rules** blade, click **Add alert**.

This opens up the **Add an alert rule** blade, where you can configure the conditions for the alert and choose among one or all of these options: send email to someone, use a webhook to forward the alert to another system, and/or run an Automation runbook in response attempt to remediate the issue.

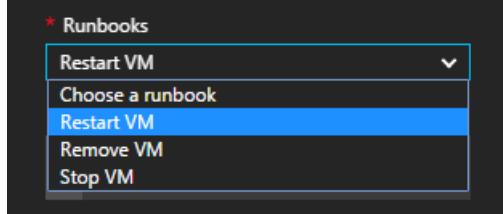
Configure a runbook

To configure a runbook to run when the VM alert threshold is met, select **Automation Runbook**. In the **Configure runbook** blade, you can select the runbook to run and the Automation account to run the runbook in.

<h3>Add an alert rule</h3> <p>* Metric ⓘ Memory usage</p> <p>* Condition greater than</p> <p>* Threshold ⓘ 75 %</p> <p>* Period ⓘ Over the last hour</p> <p>Email owners, contributors, and readers <input type="checkbox"/></p> <p>Additional administrator email(s) admin@contoso.com</p> <p>Webhook ⓘ HTTP or HTTPS endpoint to route alerts to Learn more about configuring webhooks</p> <p>Automation Runbook ⓘ Not configured</p> <p>OK</p>	<h3>Configure Runbook</h3> <p>* Run runbook Enabled</p> <p>* Runbooks Restart VM</p> <p>This runbook will restart the Azure virtual machine that triggered the alert.</p> <p>* Automation account + New</p> <p>* Automation account Create new account ></p> <p>OK</p>	<h3>Add Automation Account</h3> <p>* Name ⓘ ContosoAutomationAccount</p> <p>Subscription name Visual Studio Ultimate with MSDN</p> <p>* Resource group AutomationGroup</p> <p>* Location East US 2</p> <p>i This will create a new Azure Run As account (service principal user) in Azure Active directory and assign the Contributor role to this user at the subscription level. Learn more</p> <p>OK</p>
--	--	---

NOTE

For this release you can choose from three runbooks that the service provides – Restart VM, Stop VM, or Remove VM (delete it). The ability to select other runbooks or one of your own runbooks will be available in a future release.



After you select one of the three available runbooks, the **Automation account** drop-down list appears and you can select an automation account the runbook will run as. Runbooks need to run in the context of an **Automation account** that is in your Azure subscription. You can select an Automation account that you already created, or you can have a new Automation account created for you.

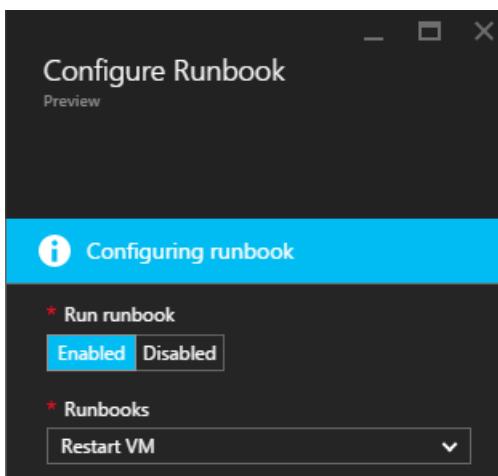
The runbooks that are provided authenticate to Azure using a service principal. If you choose to run the runbook in one of your existing Automation accounts, we will automatically create the service principal for you. If you choose to create a new Automation account, then we will automatically create the account and the service principal. In both cases, two assets will also be created in the Automation account – a certificate asset named **AzureRunAsCertificate** and a connection asset named **AzureRunAsConnection**. The runbooks will use

AzureRunAsConnection to authenticate with Azure in order to perform the management action against the VM.

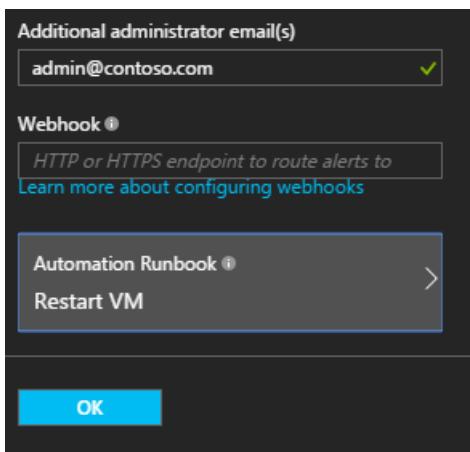
NOTE

The service principal is created in the subscription scope and is assigned the Contributor role. This role is required in order for the account to have permission to run Automation runbooks to manage Azure VMs. The creation of an Automaton account and/or service principal is a one-time event. Once they are created, you can use that account to run runbooks for other Azure VM alerts.

When you click **OK** the alert is configured and if you selected the option to create a new Automation account, it is created along with the service principal. This can take a few seconds to complete.



After the configuration is completed you will see the name of the runbook appear in the **Add an alert rule** blade.



Click **OK** in the **Add an alert rule** blade and the alert rule will be created and activate if the virtual machine is in a running state.

Enable or disable a runbook

If you have a runbook configured for an alert, you can disable it without removing the runbook configuration. This allows you to keep the alert running and perhaps test some of the alert rules and then later re-enable the runbook.

Create a runbook that works with an Azure alert

When you choose a runbook as part of an Azure alert rule, the runbook needs to have logic in it to manage the alert data that is passed to it. When a runbook is configured in an alert rule, a webhook is created for the runbook; that webhook is then used to start the runbook each time the alert triggers. The actual call to start the runbook is an HTTP POST request to the webhook URL. The body of the POST request contains a JSON-formatted object that contains useful properties related to the alert. As you can see below, the alert data contains details like `subscriptionID`, `resourceGroupName`, `resourceName`, and `resourceType`.

Example of Alert data

```
{  
    "WebhookName": "AzureAlertTest",  
    "RequestBody": "{  
        \"status\": \"Activated\",  
        \"context\": {  
  
            \"id\": \"/subscriptions/<subscriptionId>/resourceGroups/MyResourceGroup/providers/microsoft.insights/alertrules/AlertTest\",  
            \"name\": \"AlertTest\",  
            \"description\": \"\",  
            \"condition\": {  
                \"metricName\": \"CPU percentage guest OS\",  
                \"metricUnit\": \"Percent\",  
                \"metricValue\": \"4.26337916666667\",  
                \"threshold\": \"1\",  
                \"windowSize\": \"60\",  
                \"timeAggregation\": \"Average\",  
                \"operator\": \"GreaterThan\",  
                \"subscriptionId\": <subscriptionID>,  
                \"resourceGroupName\": \"TestResourceGroup\",  
                \"timestamp\": \"2016-04-24T23:19:50.1440170Z\",  
                \"resourceName\": \"TestVM\",  
                \"resourceType\": \"microsoft.compute/virtualmachines\",  
                \"resourceRegion\": \"westus\",  
  
                \"resourceId\": \"/subscriptions/<subscriptionId>/resourceGroups/TestResourceGroup/providers/Microsoft.Compute/virtualMachines/TestVM\",  
  
                \"portalLink\": \"https://portal.azure.com/#resource/subscriptions/<subscriptionId>/resourceGroups/TestResourceGroup/providers/Microsoft.Compute/virtualMachines/TestVM\"  
            },  
            \"properties\": {},  
            \"RequestHeader\": {  
                \"Connection\": \"Keep-Alive\",  
                \"Host\": <webhookURL>  
            }  
        }  
    }  
}
```

When the Automation webhook service receives the HTTP POST it extracts the alert data and passes it to the runbook in the WebhookData runbook input parameter. Below is a sample runbook that shows how to use the WebhookData parameter and extract the alert data and use it to manage the Azure resource that triggered the alert.

Example runbook

```

# This runbook will restart an ARM (V2) VM in response to an Azure VM alert.

[OutputType("PSAzureOperationResponse")]

param ( [object] $WebhookData )

if ($WebhookData)
{
    # Get the data object from WebhookData
    $WebhookBody = (ConvertFrom-Json -InputObject $WebhookData.RequestBody)

    # Assure that the alert status is 'Activated' (alert condition went from false to true)
    # and not 'Resolved' (alert condition went from true to false)
    if ($WebhookBody.status -eq "Activated")
    {
        # Get the info needed to identify the VM
        $AlertContext = [object] $WebhookBody.context
        $ResourceName = $AlertContext.resourceName
        $ResourceType = $AlertContext.resourceType
        $ResourceGroupName = $AlertContext.resourceGroupName
        $SubId = $AlertContext.subscriptionId

        # Assure that this is the expected resource type
        Write-Verbose "ResourceType: $ResourceType"
        if ($ResourceType -eq "microsoft.compute/virtualmachines")
        {
            # This is an ARM (V2) VM

            # Authenticate to Azure with service principal and certificate
            $ConnectionAssetName = "AzureRunAsConnection"
            $Conn = Get-AutomationConnection -Name $ConnectionAssetName
            if ($Conn -eq $null) {
                throw "Could not retrieve connection asset: $ConnectionAssetName. Check that this asset exists in the Automation account."
            }
            Add-AzureRMAccount -ServicePrincipal -Tenant $Conn.TenantID -ApplicationId $Conn.ApplicationID -CertificateThumbprint $Conn.CertificateThumbprint | Write-Verbose
            Set-AzureRmContext -SubscriptionId $SubId -ErrorAction Stop | Write-Verbose

            # Restart the VM
            Restart-AzureRmVM -Name $ResourceName -ResourceGroupName $ResourceGroupName
        } else {
            Write-Error "$ResourceType is not a supported resource type for this runbook."
        }
    } else {
        # The alert status was not 'Activated' so no action taken
        Write-Verbose ("No action taken. Alert status: " + $WebhookBody.status)
    }
} else {
    Write-Error "This runbook is meant to be started from an Azure alert only."
}

```

Summary

When you configure an alert on an Azure VM, you now have the ability to easily configure an Automation runbook to automatically perform remediation action when the alert triggers. For this release, you can choose from runbooks to restart, stop, or delete a VM depending on your alert scenario. This is just the beginning of enabling scenarios where you control the actions (notification, troubleshooting, remediation) that will be taken automatically when an alert triggers.

Next Steps

- To get started with Graphical runbooks, see [My first graphical runbook](#)

- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#)
- To learn more about runbook types, their advantages and limitations, see [Azure Automation runbook types](#)

Azure Automation scenario: Using JSON-formatted tags to create a schedule for Azure VM startup and shutdown

1/23/2017 • 8 min to read • [Edit on GitHub](#)

Customers often want to schedule the startup and shutdown of virtual machines to help reduce subscription costs or support business and technical requirements.

The following scenario enables you to set up automated startup and shutdown of your VMs by using a tag called Schedule at a resource group level or virtual machine level in Azure. This schedule can be configured from Sunday to Saturday with a startup time and shutdown time.

We do have some out-of-the-box options. These include:

- [Virtual machine scale sets](#) with autoscale settings that enable you to scale in or out.
- [DevTest Labs](#) service, which has the built-in capability of scheduling startup and shutdown operations.

However, these options only support specific scenarios and cannot be applied to infrastructure-as-a-service (IaaS) VMs.

When the Schedule tag is applied to a resource group, it's also applied to all virtual machines inside that resource group. If a schedule is also directly applied to a VM, the last schedule takes precedence in the following order:

1. Schedule applied to a resource group
2. Schedule applied to a resource group and virtual machine in the resource group
3. Schedule applied to a virtual machine

This scenario essentially takes a JSON string with a specified format and adds it as the value for a tag called Schedule. Then a runbook lists all resource groups and virtual machines and identifies the schedules for each VM based on the scenarios listed earlier. Next it loops through the VMs that have schedules attached and evaluates what action should be taken. For example, it determines which VMs need to be stopped, shut down, or ignored.

These runbooks authenticate by using the [Azure Run As account](#).

Download the runbooks for the scenario

This scenario consists of four PowerShell Workflow runbooks that you can download from the [TechNet Gallery](#) or the [GitHub](#) repository for this project.

RUNBOOK	DESCRIPTION
Test-ResourceSchedule	Checks each virtual machine schedule and performs shutdown or startup depending on the schedule.
Add-ResourceSchedule	Adds the Schedule tag to a VM or resource group.
Update-ResourceSchedule	Modifies the existing Schedule tag by replacing it with a new one.
Remove-ResourceSchedule	Removes the Schedule tag from a VM or resource group.

Install and configure this scenario

Install and publish the runbooks

After downloading the runbooks, you can import them by using the procedure in [Creating or importing a runbook in Azure Automation](#). Publish each runbook after it has been successfully imported into your Automation account.

Add a schedule to the Test-ResourceSchedule runbook

Follow these steps to enable the schedule for the Test-ResourceSchedule runbook. This is the runbook that verifies which virtual machines should be started, shut down, or left as is.

1. From the Azure portal, open your Automation account, and then click the **Runbooks** tile.
2. On the **Test-ResourceSchedule** blade, click the **Schedules** tile.
3. On the **Schedules** blade, click **Add a schedule**.
4. On the **Schedules** blade, select **Link a schedule to your runbook**. Then select **Create a new schedule**.
5. On the **New schedule** blade, type in the name of this schedule, for example: *HourlyExecution*.
6. For the schedule **Start**, set the start time to an hour increment.
7. Select **Recurrence**, and then for **Recur every interval**, select **1 hour**.
8. Verify that **Set expiration** is set to **No**, and then click **Create** to save your new schedule.
9. On the **Schedule Runbook** options blade, select **Parameters and run settings**. In the Test-ResourceSchedule **Parameters** blade, enter the name of your subscription in the **SubscriptionName** field. This is the only parameter that's required for the runbook. When you're finished, click **OK**.

The runbook schedule should look like the following when it's completed:

The screenshot shows the 'Schedules' blade for the 'Test-ResourceSchedule' runbook. At the top, there are two buttons: 'Add a schedule.' (with a clock icon) and 'Refresh' (with a circular arrow icon). Below this is a table with three columns: 'NAME', 'NEXT RUN', and 'STATUS'. A single row is present, showing 'HourlyExecution' in the NAME column, '7/11/2016 7:00 PM (ET)' in the NEXT RUN column, and a green checkmark and the word 'On' in the STATUS column.

NAME	NEXT RUN	STATUS
HourlyExecution	7/11/2016 7:00 PM (ET)	✓ On

Format the JSON string

This solution basically takes a JSON string with a specified format and adds it as the value for a tag called Schedule. Then a runbook lists all resource groups and virtual machines and identifies the schedules for each virtual machine.

The runbook loops over the virtual machines that have schedules attached and checks what actions should be taken. The following is an example of how the solutions should be formatted:

```
{
    "TzId": "Eastern Standard Time",
    "0": {
        "S": "11",
        "E": "17"
    },
    "1": {
        "S": "9",
        "E": "19"
    },
    "2": {
        "S": "9",
        "E": "19"
    }
}
```

Here is some detailed information about this structure:

1. The format of this JSON structure is optimized to work around the 256-character limitation of a single tag value in Azure.
2. *TzId* represents the time zone of the virtual machine. This ID can be obtained by using the `TimeZoneInfo` .NET class in a PowerShell session--**[System.TimeZoneInfo]::GetSystemTimeZones()**.

```
PS C:\> [System.TimeZoneInfo]::GetSystemTimeZones()

Id                               : Dateline Standard Time
DisplayName                      : (UTC-12:00) International Date Line West
StandardName                     : Dateline Standard Time
DaylightName                     : Dateline Daylight Time
BaseUtcOffset                    : -12:00:00
SupportsDaylightSavingTime     : False

Id                               : UTC-11
DisplayName                      : (UTC-11:00) Coordinated Universal Time-11
StandardName                     : UTC-11
DaylightName                     : UTC-11
BaseUtcOffset                    : -11:00:00
SupportsDaylightSavingTime     : False
```

- Weekdays are represented with a numeric value of zero to six. The value zero equals Sunday.
- The start time is represented with the **S** attribute, and its value is in a 24-hour format.
- The end or shutdown time is represented with the **E** attribute, and its value is in a 24-hour format.

If the **S** and **E** attributes each have a value of zero (0), the virtual machine will be left in its present state at the time of evaluation.

3. If you want to skip evaluation for a specific day of the week, don't add a section for that day of the week. In the following example, only Monday is evaluated, and the other days of the week are ignored:

```
{
    "TzId": "Eastern Standard Time",
    "1": {
        "S": "11",
        "E": "17"
    }
}
```

Tag resource groups or VMs

To shut down VMs, you need to tag either the VMs or the resource groups in which they're located. Virtual machines that don't have a Schedule tag are not evaluated. Therefore, they aren't started or shut down.

There are two ways to tag resource groups or VMs with this solution. You can do it directly from the portal. Or you can use the Add-ResourceSchedule, Update-ResourceSchedule, and Remove-ResourceSchedule runbooks.

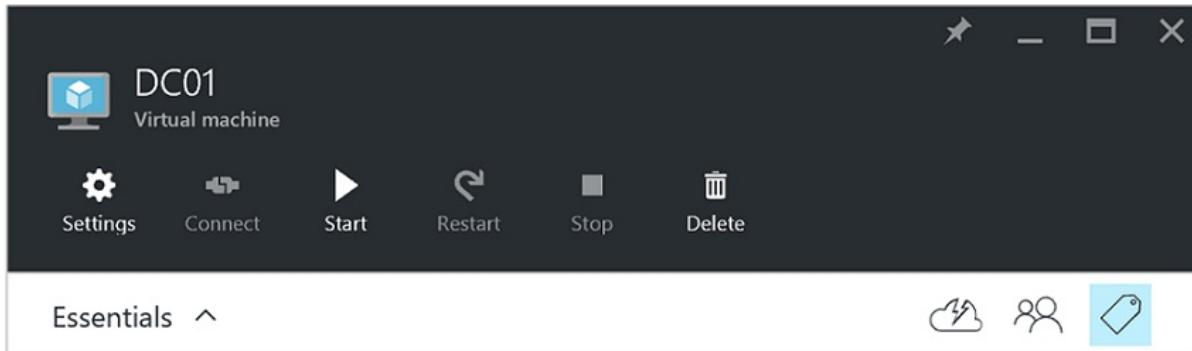
Tag through the portal

Follow these steps to tag a virtual machine or resource group in the portal:

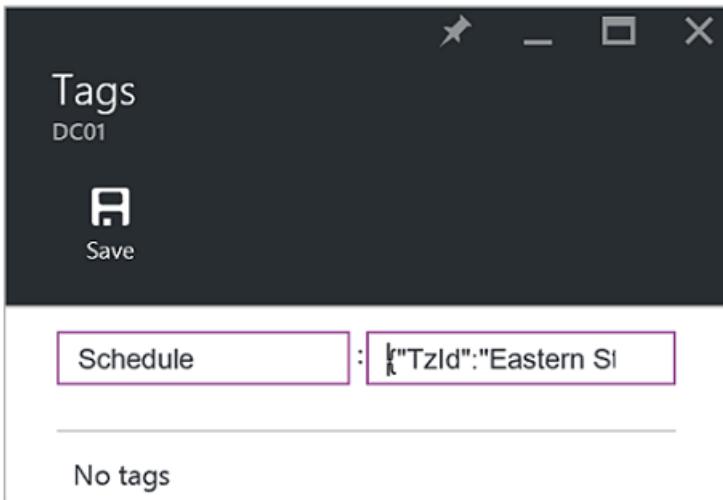
1. Flatten the JSON string and verify that there aren't any spaces. Your JSON string should look like this:

```
{"TzId": "Eastern Standard Time", "0": {"S": "11", "E": "17"}, "1": {"S": "9", "E": "19"}, "2": {"S": "9", "E": "19"}, "3": {"S": "9", "E": "19"}, "4": {"S": "9", "E": "19"}, "5": {"S": "9", "E": "19"}, "6": {"S": "11", "E": "17"}}
```

2. Select the **Tag** icon for a VM or resource group to apply this schedule.



3. Tags are defined following a key/value pair. Type **Schedule** in the **Key** field, and then paste the JSON string into the **Value** field. Click **Save**. Your new tag should now appear in the list of tags for your resource.



Tag from PowerShell

All imported runbooks contain help information at the beginning of the script that describes how to execute the runbooks directly from PowerShell. You can call the Add-ScheduleResource and Update-ScheduleResource runbooks from PowerShell. You do this by passing required parameters that enable you to create or update the Schedule tag on a VM or resource group outside of the portal.

To create, add, and delete tags through PowerShell, you first need to [set up your PowerShell environment for Azure](#). After you complete the setup, you can proceed with the following steps.

Create a schedule tag with PowerShell

1. Open a PowerShell session. Then use the following example to authenticate with your Run As account and to specify a subscription:

```
$Conn = Get-AutomationConnection -Name AzureRunAsConnection
Add-AzureRMAccount -ServicePrincipal -Tenant $Conn.TenantID ` 
-ApplicationId $Conn.ApplicationID -CertificateThumbprint $Conn.CertificateThumbprint
Select-AzureRmSubscription -SubscriptionName "MySubscription"
```

2. Define a schedule hash table. Here is an example of how it should be constructed:

```
$schedule= @{ "TzId"="Eastern Standard Time"; "0"= @{$S="11";$E="17"};"1"= @{$S="9";$E="19"};"2"= 
 @{$S="9";$E="19"};"3"= @{$S="9";$E="19"};"4"= @{$S="9";$E="19"};"5"= @{$S="9";$E="19"};"6"= 
 @{$S="11";$E="17"}}
```

3. Define the parameters that are required by the runbook. In the following example, we are targeting a VM:

```
$params = @{"SubscriptionName"="MySubscription";"ResourceGroupName"="ResourceGroup01";
"VmName"="VM01";"Schedule"=$schedule}
```

If you're tagging a resource group, remove the *VMName* parameter from the \$params hash table as follows:

```
$params = @{"SubscriptionName"="MySubscription";"ResourceGroupName"="ResourceGroup01";
"Schedule"=$schedule}
```

4. Run the Add-ResourceSchedule runbook with the following parameters to create the Schedule tag:

```
Start-AzureRmAutomationRunbook -Name "Add-ResourceSchedule" -Parameters $params ` 
-AutomationAccountName "AutomationAccount" -ResourceGroupName "ResourceGroup01"
```

5. To update a resource group or virtual machine tag, execute the **Update-ResourceSchedule** runbook with the following parameters:

```
Start-AzureRmAutomationRunbook -Name "Update-ResourceSchedule" -Parameters $params ` 
-AutomationAccountName "AutomationAccount" -ResourceGroupName "ResourceGroup01"
```

Remove a schedule tag with PowerShell

1. Open a PowerShell session and run the following to authenticate with your Run As account and to select and specify a subscription:

```
Conn = Get-AutomationConnection -Name AzureRunAsConnection
Add-AzureRMAccount -ServicePrincipal -Tenant $Conn.TenantID ` 
-ApplicationId $Conn.ApplicationID -CertificateThumbprint $Conn.CertificateThumbprint
Select-AzureRmSubscription -SubscriptionName "MySubscription"
```

2. Define the parameters that are required by the runbook. In the following example, we are targeting a VM:

```
$params = @{"SubscriptionName"="MySubscription";"ResourceGroupName"="ResourceGroup01";"VmName"="VM01"}
```

If you're removing a tag from a resource group, remove the *VMName* parameter from the \$params hash table as follows:

```
$params = @{"SubscriptionName"="MySubscription";"ResourceGroupName"="ResourceGroup01"}
```

3. Execute the Remove-ResourceSchedule runbook to remove the Schedule tag:

```
Start-AzureRmAutomationRunbook -Name "Remove-ResourceSchedule" -Parameters $params  
-AutomationAccountName "AutomationAccount" -ResourceGroupName "ResourceGroup01"
```

4. To update a resource group or virtual machine tag, execute the Remove-ResourceSchedule runbook with the following parameters:

```
Start-AzureRmAutomationRunbook -Name "Remove-ResourceSchedule" -Parameters $params  
-AutomationAccountName "AutomationAccount" -ResourceGroupName "ResourceGroup01"
```

NOTE

We recommend that you proactively monitor these runbooks (and the virtual machine states) to verify that your virtual machines are being shut down and started accordingly.

To view the details of the Test-ResourceSchedule runbook job in the Azure portal, select the **Jobs** tile of the runbook. The job summary displays the input parameters and the output stream, in addition to general information about the job and any exceptions if they occurred.

The **Job Summary** includes messages from the output, warning, and error streams. Select the **Output** tile to view detailed results from the runbook execution.

The screenshot shows the Azure portal interface for a runbook job. On the left, the 'Job' tab is selected for the 'Test-ResourceSchedule' job, which was created on 7/12/2016 at 10:00 PM. The 'Overview' section shows the job ID, creation date, last update date, and that it ran on Azure. The 'Status' section indicates 0 errors and 0 warnings. The 'Exception' section shows 'None'. On the right, the 'Output' tab is selected, displaying the log stream. The logs show the process of getting a list of resource groups, building a schedule list for VMs containing the 'Schedule Tag', and getting VMs from the 'InfraLab' resource group. It also lists VMs to be evaluated (DC01, FS01, SQLCL01, SQLCL02) and their corresponding resource schedules. The final log entry shows 'vmList Count => 1'.

```
Test-ResourceSchedule 7/12/2016 10:00 PM
Job
Resume Stop Suspend View source
Overview Job Summary
Job ID: 8b9a1259-f1b5-4196-8362-388752d669d0
Created: 7/12/2016 10:00 PM
Last updated: 7/12/2016 10:07 PM
Ran on Azure
Completed
Status Errors 0 Warnings 0 All Logs
Add tiles +
Exception None
Add a section +
Output
Test-ResourceSchedule 7/12/2016 10:00 PM
Getting list of resource groups
Resource group count: 3
Building Schedule List for VMs that contains the Schedule Tag
Getting VMs from Resource Group InfraLab
VM to be evaluated: DC01
Resource Schedule for vm DC01 is {"TzId":"Eastern Standard Time","0":{"S":"1","E":"9","M":{},"5":{"S":"9","E":"19"}, "6":{"S":"11","E":"17"}}, "1":{}, "2":{}, "3":{}, "4":{}, "7":{}, "8":{}, "9":{}, "10":{}, "11":{}, "12":{}, "13":{}, "14":{}, "15":{}, "16":{}, "17":{}, "18":{}, "19":{}, "20":{}, "21":{}, "22":{}, "23":{}}
VM to be evaluated: FS01
VM to be evaluated: SQLCL01
VM to be evaluated: SQLCL02
Getting VMs from Resource Group SQL-HA
Getting VMs from Resource Group SQL-HA-RG
VM to be evaluated: DC01
VM to be evaluated: DC02
VM to be evaluated: FS01
VM to be evaluated: SQLCL01
VM to be evaluated: SQLCL02
vmList Count => 1
```

Next steps

- To get started with PowerShell workflow runbooks, see [My first PowerShell workflow runbook](#).
- To learn more about runbook types, and their advantages and limitations, see [Azure Automation runbook types](#).
- For more information about PowerShell script support features, see [Native PowerShell script support in Azure Automation](#).
- To learn more about runbook logging and output, see [Runbook output and messages in Azure Automation](#).

- To learn more about an Azure Run As account and how to authenticate your runbooks by using it, see [Authenticate runbooks with Azure Run As account](#).

Azure Automation scenario - automate removal of resource groups

1/17/2017 • 2 min to read • [Edit on GitHub](#)

Many customers create more than one resource group. Some might be used for managing production applications, and others might be used as development, testing, and staging environments. Automating the deployment of these resources is one thing, but being able to decommission a resource group with a click of the button is another. You can streamline this common management task by using Azure Automation. This is helpful if you are working with an Azure subscription that has a spending limit through a member offer like MSDN or the Microsoft Partner Network Cloud Essentials program.

This scenario is based on a PowerShell runbook and is designed to remove one or more resource groups that you specify from your subscription. The default setting of the runbook is to test before proceeding. This ensures that you don't accidentally delete the resource group before you're ready to complete this procedure.

Getting the scenario

This scenario consists of a PowerShell runbook that you can download from the [PowerShell Gallery](#). You can also import it directly from the [Runbook Gallery](#) in the Azure portal.

RUNBOOK	DESCRIPTION
Remove-ResourceGroup	Removes one or more Azure resource groups and associated resources from the subscription.

The following input parameters are defined for this runbook:

PARAMETER	DESCRIPTION
NameFilter (Required)	Specifies a name filter to limit the resource groups that you intend on deleting. You can pass multiple values using a comma-separated list. The filter is not case-sensitive and will match any resource group that contains the string.
PreviewMode (Optional)	Executes the runbook to see which resource groups would be deleted, but takes no action. The default is true to help avoid accidental deletion of one or more resource groups passed to the runbook.

Install and configure this scenario

Prerequisites

This runbook authenticates using the [Azure Run As account](#).

Install and publish the runbooks

After you download the runbook, you can import it by using the procedure in [Importing runbook procedures](#). Publish the runbook after it has been successfully imported into your Automation account.

Using the runbook

The following steps will walk you through the execution of this runbook and help you become familiar with how it works. You will only be testing the runbook in this example, not actually deleting the resource group.

1. From the Azure portal, open your Automation account and click **Runbooks**.
2. Select the **Remove-ResourceGroup** runbook and click **Start**.
3. When you start the runbook, the **Start Runbook** blade opens and you can configure the parameters. Enter the names of resource groups in your subscription that you can use for testing and will cause no harm if accidentally deleted.

The screenshot shows the 'Start Runbook' blade for the 'Remove-ResourceGroup' runbook. At the top, there are window control buttons (minimize, maximize, close) and the title 'Start Runbook' followed by the runbook name 'Remove-ResourceGroup'. Below the title, the 'Parameters' section is visible, containing a mandatory string parameter named 'NAMEFILTER' with a placeholder 'Enter a value'. The 'PREVIEWMODE' section shows a dropdown set to 'Default will be used' with an optional boolean default of '\$true'. In the 'Run Settings' section, 'Run on Azure' is selected. A note at the bottom of the blade states: 'Make sure Previewmode is set to true to avoid deleting the selected resource groups. Note that this runbook will not remove the resource group that contains the Automation account that is running this runbook.'

NOTE
Make sure **Previewmode** is set to **true** to avoid deleting the selected resource groups. **Note** that this runbook will not remove the resource group that contains the Automation account that is running this runbook.

4. After you have configured all the parameter values, click **OK**, and the runbook will be queued for execution.

To view the details of the **Remove-ResourceGroup** runbook job in the Azure portal, select **Jobs** in the runbook. The job summary displays the input parameters and the output stream in addition to general information about the job and any exceptions that occurred.

The screenshot shows the 'Job' tab for a runbook named 'Remove-ResourceGroup'. The job was created on 9/23/2016 at 9:37 AM and last updated at 9:39 AM. It ran on Azure and completed successfully. The runbook has 1 input and no output. There are 0 errors and 0 warnings. The exception log is empty.

Job Summary

Job ID: 96652b3c-1327-47e1-94c9-382be7666ae0
Created: 9/23/2016 9:37 AM
Last updated: 9/23/2016 9:39 AM
Ran on Azure

Completed

Status

Errors: 0 ✗ Warnings: 0 ! All Logs

Exception

None

The **Job Summary** includes messages from the output, warning, and error streams. Select **Output** to view detailed results from the runbook execution.

The output page shows the results of the 'Remove-ResourceGroup' runbook. It indicates that the resource group will not be removed due to preview mode. It lists the resources that would be removed in preview mode, categorized by resource type and location.

Output
Remove-ResourceGroup 9/20/2016 2:53 PM

The resource group for this runbook job will not be removed. Resource group: Lab

Preview Mode: The following resource groups would be removed:

Development

Preview Mode: The following resources would be removed:

```
Name : AzureAutomation-Dev
ResourceId : /subscriptions/68627f8c-032a81f8cf0/resourceGroups/Development/providers/Microsoft.Automation/automationAccounts/AzureAutomation-Dev
ResourceName : AzureAutomation-Dev
ResourceType : Microsoft.Automation/automationAccounts
ResourceGroupName : Development
Location : eastus2
SubscriptionId : 68627f8c-032a81f8cf0
Tags : {}

Name : AzureAutomation-Dev/AzureAutomationTutorial
ResourceId : /subscriptions/68627f8c-032a81f8cf0/resourceGroups/Development/providers/Microsoft.Automation/automationAccounts/AzureAutomation-Dev/runbooks/AzureAutomationTutorial
ResourceName : AzureAutomation-Dev/AzureAutomationTutorial
ResourceType : Microsoft.Automation/automationAccounts/runbooks
ResourceGroupName : Development
Location : eastus2
SubscriptionId : 68627f8c-032a81f8cf0
Tags : {}
```

Next steps

- To get started creating your own runbook, see [Creating or importing a runbook in Azure Automation](#).
- To get started with PowerShell Workflow runbooks, see [My first PowerShell Workflow runbook](#).

Start/Stop VMs during off-hours [Preview] solution in Automation

2/14/2017 • 13 min to read • [Edit on GitHub](#)

The Start/Stop VMs during off-hours [Preview] solution starts and stops your Azure Resource Manager virtual machines on a user-defined schedule and provides insight into the success of the Automation jobs that start and stop your virtual machines with OMS Log Analytics.

Prerequisites

- The runbooks work with an [Azure Run As account](#). The Run As account is the preferred authentication method since it uses certificate authentication instead of a password that may expire or change frequently.
- This solution can only manage VMs which are in the same subscription and resource group as where the Automation account resides.
- This solution only deploys to the following Azure regions - Australia Southeast, East US, Southeast Asia, and West Europe. The runbooks that manage the VM schedule can target VMs in any region.
- To send email notifications when the start and stop VM runbooks complete, an Office 365 business-class subscription is required.

Solution components

This solution consists of the following resources that will be imported and added to your Automation account.

Runbooks

RUNBOOK	DESCRIPTION
CleanSolution-MS-Mgmt-VM	This runbook will remove all contained resources, and schedules when you go to delete the solution from your subscription.
SendMailO365-MS-Mgmt	This runbook sends an email through Office 365 Exchange.
StartByResourceGroup-MS-Mgmt-VM	This runbook is intended to start VMs (both classic and ARM based VMs) that resides in a given list of Azure resource group(s).
StopByResourceGroup-MS-Mgmt-VM	This runbook is intended to stop VMs (both classic and ARM based VMs) that resides in a given list of Azure resource group(s).

Variables

VARIABLE	DESCRIPTION
SendMailO365-MS-Mgmt Runbook	

VARIABLE	DESCRIPTION
SendMailO365-IsSendEmail-MS-Mgmt	Specifies if StartByResourceGroup-MS-Mgmt-VM and StopByResourceGroup-MS-Mgmt-VM runbooks can send email notification upon completion. Select True to enable and False to disable email alerting. Default value is False .
StartByResourceGroup-MS-Mgmt-VM Runbook	
StartByResourceGroup-ExcludeList-MS-Mgmt-VM	Enter VM names to be excluded from management operation; separate names by using semi-colon(;). Values are case-sensitive and wildcard (asterisk) is supported.
StartByResourceGroup-SendMailO365-EmailBodyPreFix-MS-Mgmt	Text that can be appended to the beginning of the email message body.
StartByResourceGroup-SendMailO365-EmailRunBookAccount-MS-Mgmt	Specifies the name of the Automation Account that contains the Email runbook. Do not modify this variable.
StartByResourceGroup-SendMailO365-EmailRunbookName-MS-Mgmt	Specifies the name of the email runbook. This is used by the StartByResourceGroup-MS-Mgmt-VM and StopByResourceGroup-MS-Mgmt-VM runbooks to send email. Do not modify this variable.
StartByResourceGroup-SendMailO365-EmailRunbookResourceGroup-MS-Mgmt	Specifies the name of the Resource group that contains the Email runbook. Do not modify this variable.
StartByResourceGroup-SendMailO365-EmailSubject-MS-Mgmt	Specifies the text for the subject line of the email.
StartByResourceGroup-SendMailO365-EmailToAddress-MS-Mgmt	Specifies the recipient(s) of the email. Enter separate names by using semi-colon(;).
StartByResourceGroup-TargetResourceGroups-MS-Mgmt-VM	Enter VM names to be excluded from management operation; separate names by using semi-colon(;). Values are case-sensitive and wildcard (asterisk) is supported. Default value (asterisk) will include all resource groups in the subscription.
StartByResourceGroup-TargetSubscriptionID-MS-Mgmt-VM	Specifies the subscription that contains VMs to be managed by this solution. This must be the same subscription where the Automation account of this solution resides.
StopByResourceGroup-MS-Mgmt-VM Runbook	
StopByResourceGroup-ExcludeList-MS-Mgmt-VM	Enter VM names to be excluded from management operation; separate names by using semi-colon(;). Values are case-sensitive and wildcard (asterisk) is supported.
StopByResourceGroup-SendMailO365-EmailBodyPreFix-MS-Mgmt	Text that can be appended to the beginning of the email message body.
StopByResourceGroup-SendMailO365-EmailRunBookAccount-MS-Mgmt	Specifies the name of the Automation Account that contains the Email runbook. Do not modify this variable.
StopByResourceGroup-SendMailO365-EmailRunbookResourceGroup-MS-Mgmt	Specifies the name of the Resource group that contains the Email runbook. Do not modify this variable.

VARIABLE	DESCRIPTION
StopByResourceGroup-SendMailO365-EmailSubject-MS-Mgmt	Specifies the text for the subject line of the email.
StopByResourceGroup-SendMailO365-EmailToAddress-MS-Mgmt	Specifies the recipient(s) of the email. Enter separate names by using semi-colon(;).
StopByResourceGroup-TargetResourceGroups-MS-Mgmt-VM	Enter VM names to be excluded from management operation; separate names by using semi-colon(;). Values are case-sensitive and wildcard (asterisk) is supported. Default value (asterisk) will include all resource groups in the subscription.
StopByResourceGroup-TargetSubscriptionID-MS-Mgmt-VM	Specifies the subscription that contains VMs to be managed by this solution. This must be the same subscription where the Automation account of this solution resides.

Schedules

SCHEDULE	DESCRIPTION
StartByResourceGroup-Schedule-MS-Mgmt	Schedule for StartByResourceGroup runbook, which performs the startup of VMs managed by this solution. When created, it defaults to UTC time zone.
StopByResourceGroup-Schedule-MS-Mgmt	Schedule for StopByResourceGroup runbook, which performs the shutdown of VMs managed by this solution. When created, it defaults to UTC time zone.

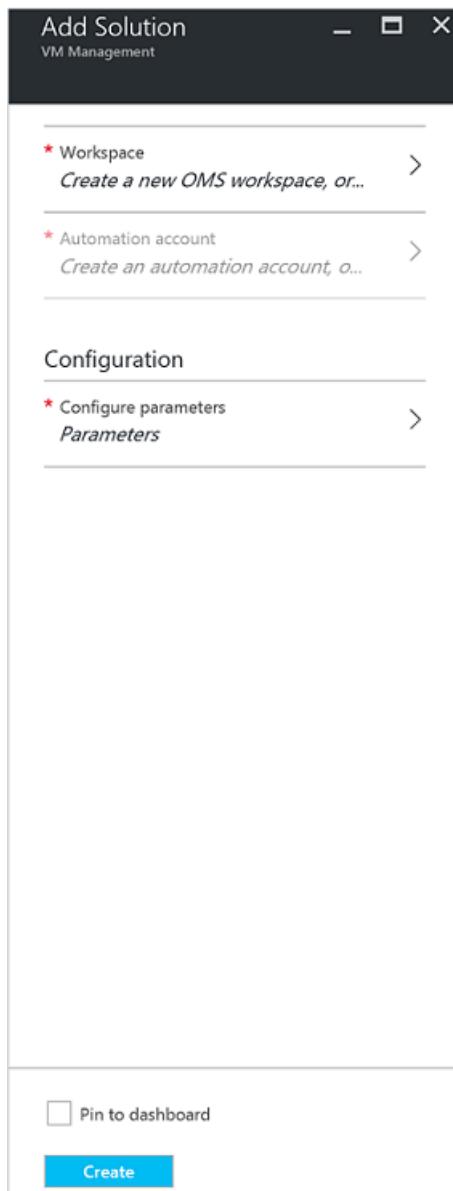
Credentials

CREDENTIAL	DESCRIPTION
O365Credential	Specifies a valid Office 365 user account to send email. Only required if variable SendMailO365-IsSendEmail-MS-Mgmt is set to True .

Configuration

Perform the following steps to add the Start/Stop VMs during off-hours [Preview] solution to your Automation account and then configure the variables to customize the solution.

- From the home-screen in the Azure portal, select the **Marketplace** tile. If the tile is no longer pinned to your home-screen, from the left navigation pane, select **New**.
- In the Marketplace blade, type **Start VM** in the search box, and then select the solution **Start/Stop VMs during off-hours [Preview]** from the search results.
- In the **Start/Stop VMs during off-hours [Preview]** blade for the selected solution, review the summary information and then click **Create**.
- The **Add Solution** blade appears where you are prompted to configure the solution before you can import it into your Automation subscription.



5. On the **Add Solution** blade, select **Workspace** and here you select an OMS workspace that is linked to the same Azure subscription that the Automation account is in or create a new OMS workspace. If you do not have an OMS workspace, you can select **Create New Workspace** and on the **OMS Workspace** blade perform the following:

- Specify a name for the new **OMS Workspace**.
- Select a **Subscription** to link to by selecting from the drop-down list if the default selected is not appropriate.
- For **Resource Group**, you can create a new resource group or select an existing resource group.
- Select a **Location**. Currently the only locations provided for selection are **Australia Southeast**, **East US**, **Southeast Asia**, and **West Europe**.
- Select a **Pricing tier**. The solution is offered in two tiers: free and OMS paid tier. The free tier has a limit on the amount of data collected daily, retention period, and runbook job runtime minutes. The OMS paid tier does not have a limit on the amount of data collected daily.

NOTE

While the Standalone paid tier is displayed as an option, it is not applicable. If you select it and proceed with the creation of this solution in your subscription, it will fail. This will be addressed when this solution is officially released.

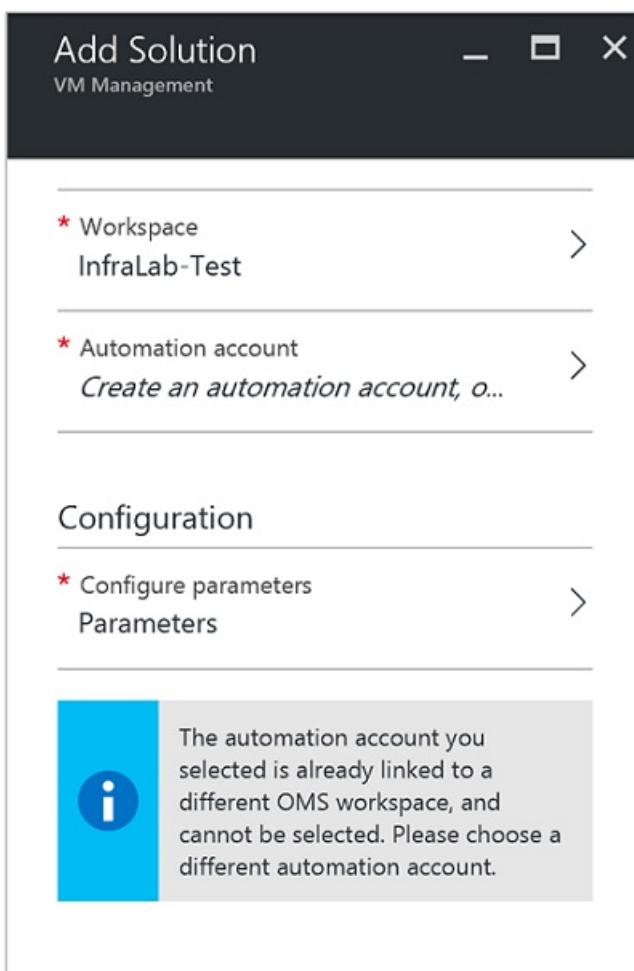
If you use this solution, it will only use automation job minutes and log ingestion. The solution does not add additional OMS nodes to your environment.

6. After providing the required information on the **OMS workspace** blade, click **Create**. While the information is verified and the workspace is created, you can track its progress under **Notifications** from the menu. You will be returned to the **Add Solution** blade.
7. On the **Add Solution** blade, select **Automation Account**. If you are creating a new OMS workspace, you will be required to also create a new Automation account that will be associated with the new OMS workspace specified earlier, including your Azure subscription, resource group and region. You can select **Create an Automation account** and on the **Add Automation account** blade, provide the following:

- In the **Name** field, enter the name of the Automation account.

All other options are automatically populated based on the OMS workspace selected and these options cannot be modified. An Azure Run As account is the default authentication method for the runbooks included in this solution. Once you click **OK**, the configuration options are validated and the Automation account is created. You can track its progress under **Notifications** from the menu.

Otherwise, you can select an existing Automation Run As account. Note that the account you select cannot already be linked to another OMS workspace, otherwise a message will be presented in the blade to inform you. If it is already linked, you will need to select a different Automation Run As account or create a new one.



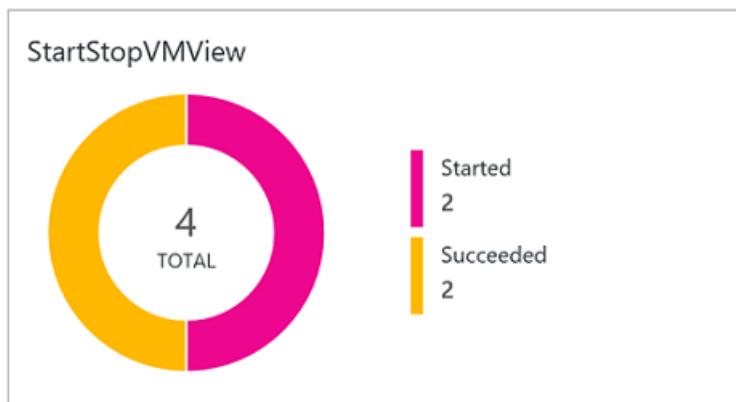
8. Finally on the **Add Solution** blade, select **Configuration** and the **Parameters** blade appears. On the **Parameters** blade, you are prompted to:
 - Specify the **Target ResourceGroup Names**, which is a resource group name that contains VMs to be managed by this solution. You can enter more than one name and separate each using a semi-colon (values are case-sensitive). Using a wildcard is supported if you want to target VMs in all resource groups in the subscription.
 - Select a **Schedule** which is a recurring date and time for starting and stopping the VM's in the target resource group(s). By default, the schedule is configured to the UTC time zone and selecting a different region is not available. If you wish to configure the schedule to your specific time zone after configuring the solution, see [Modifying the startup and shutdown schedule](#) below.
9. Once you have completed configuring the initial settings required for the solution, select **Create**. All settings will be validated and then it will attempt to deploy the solution in your subscription. This process can take several seconds to complete and you can track its progress under **Notifications** from the menu.

Collection frequency

Automation job log and job stream data is ingested into the OMS repository every five minutes.

Using the solution

When you add the VM Management solution, in your OMS workspace the **StartStopVM View** tile will be added to your OMS dashboard. This tile displays a count and graphical representation of the runbooks jobs for the solution that have started and have completed successfully.



In your Automation account, you can access and manage the solution by selecting the **Solutions** tile and then from the **Solutions** blade, selecting the solution **Start-Stop-VM[Workspace]** from the list.

The screenshot shows the Azure portal interface. On the left, the 'Test-Automation' resource group is selected under the 'InfraLab-Test' subscription. The 'Essentials' section displays basic information: Resource group (InfraLab-Test), Location (East US 2), and Subscription name (Microsoft Azure). It also shows the status as Active, last modified on 9/26/2016 at 1:29 PM, and last modified by AutomationAcct@onmicrosoft.com. A 'Solutions' blade is open on the right, showing a search bar for 'Solution Filter...' and a list of solutions. The first solution listed is 'Start-Stop-VM[InfraLab-Test]'.

Selecting the solution will display the **Start-Stop-VM[Workspace]** solution blade, where you can review important details such as the **StartStopVM** tile, like in your OMS workspace, which displays a count and graphical representation of the runbooks jobs for the solution that have started and have completed successfully.

The screenshot shows the 'Start-Stop-VM[InfraLab-Test]' solution blade. At the top, it displays the solution name, type (Microsoft.OperationsManagement/solution), workspace name (InfraLab-Test), and management services (Operations logs). Below this, the 'Summary' section features a 'StartStopVMView SOLUTION VIEW' chart. The chart shows a total of 4 jobs, with 2 Started and 2 Succeeded. The 'Solution Resources' section lists four resources: StartByResourceGroup-MS-Mgmt-VM and StopByResourceGroup-MS-Mgmt-VM, along with three other items represented by ellipses (...).

From here you can also open your OMS workspace and perform further analysis of the job records. Just click **All settings**, and in the **Settings** blade, select **Quick Start** and then in the **Quick Start** blade select **OMS Portal**. This will open a new tab or new browser session and present your OMS workspace associated with your Automation account and subscription.

Configuring e-mail notifications

To enable email notifications when the start and stop VM runbooks complete, you will need to modify the **O365Credential** credential and at a minimum, the following variables:

- SendMailO365-IsSendEmail-MS-Mgmt
- StartByResourceGroup-SendMailO365-EmailToAddress-MS-Mgmt
- StopByResourceGroup-SendMailO365-EmailToAddress-MS-Mgmt

To configure the **O365Credential** credential, perform the following steps:

1. From your automation account, click **All Settings** at the top of the window.
2. On the **Settings** blade under the section **Automation Resources**, select **Assets**.
3. On the **Assets** blade, select the **Credential** tile and from the **Credential** blade, select the **O365Credential**.
4. Enter a valid Office 365 username and password and then click **Save** to save your changes.

To configure the variables highlighted earlier, perform the following steps:

1. From your automation account, click **All Settings** at the top of the window.
2. On the **Settings** blade under the section **Automation Resources**, select **Assets**.
3. On the **Assets** blade, select the **Variables** tile and from the **Variables** blade, select the variable listed above and then modify its value following the description for it specified in the **variable** section earlier.
4. Click **Save** to save the changes to the variable.

Modifying the startup and shutdown schedule

Managing the startup and shutdown schedule in this solution follows the same steps as outlined in [Scheduling a runbook in Azure Automation](#). Remember, you cannot modify the schedule configuration. You will need to disable the existing schedule and then create a new one and then link to the **StartByResourceGroup-MS-Mgmt-VM** or **StopByResourceGroup-MS-Mgmt-VM** runbook that you want the schedule to apply to.

Log Analytics records

Automation creates two types of records in the OMS repository.

Job logs

PROPERTY	DESCRIPTION
Caller	Who initiated the operation. Possible values are either an email address or system for scheduled jobs.
Category	Classification of the type of data. For Automation, the value is JobLogs.
CorrelationId	GUID that is the Correlation Id of the runbook job.
JobId	GUID that is the Id of the runbook job.
operationName	Specifies the type of operation performed in Azure. For Automation, the value will be Job.

PROPERTY	DESCRIPTION
resourceId	Specifies the resource type in Azure. For Automation, the value is the Automation account associated with the runbook.
ResourceGroup	Specifies the resource group name of the runbook job.
ResourceProvider	Specifies the Azure service that supplies the resources you can deploy and manage. For Automation, the value is Azure Automation.
ResourceType	Specifies the resource type in Azure. For Automation, the value is the Automation account associated with the runbook.
resultType	The status of the runbook job. Possible values are: - Started - Stopped - Suspended - Failed - Succeeded
resultDescription	Describes the runbook job result state. Possible values are: - Job is started - Job Failed - Job Completed
RunbookName	Specifies the name of the runbook.
SourceSystem	Specifies the source system for the data submitted. For Automation, the value will be :OpsManager
StreamType	Specifies the type of event. Possible values are: - Verbose - Output - Error - Warning
SubscriptionId	Specifies the subscription ID of the job.
Time	Date and time when the runbook job executed.

Job streams

PROPERTY	DESCRIPTION
Caller	Who initiated the operation. Possible values are either an email address or system for scheduled jobs.
Category	Classification of the type of data. For Automation, the value is JobStreams.
JobId	GUID that is the Id of the runbook job.
operationName	Specifies the type of operation performed in Azure. For Automation, the value will be Job.

PROPERTY	DESCRIPTION
ResourceGroup	Specifies the resource group name of the runbook job.
resourceId	Specifies the resource Id in Azure. For Automation, the value is the Automation account associated with the runbook.
ResourceProvider	Specifies the Azure service that supplies the resources you can deploy and manage. For Automation, the value is Azure Automation.
ResourceType	Specifies the resource type in Azure. For Automation, the value is the Automation account associated with the runbook.
resultType	The result of the runbook job at the time the event was generated. Possible values are: - InProgress
resultDescription	Includes the output stream from the runbook.
RunbookName	The name of the runbook.
SourceSystem	Specifies the source system for the data submitted. For Automation, the value will be OpsManager
StreamType	The type of job stream. Possible values are: - Progress - Output - Warning - Error - Debug - Verbose
Time	Date and time when the runbook job executed.

When you perform any log search that returns records of category of **JobLogs** or **JobStreams**, you can select the **JobLogs** or **JobStreams** view which displays a set of tiles summarizing the updates returned by the search.

Sample log searches

The following table provides sample log searches for job records collected by this solution.

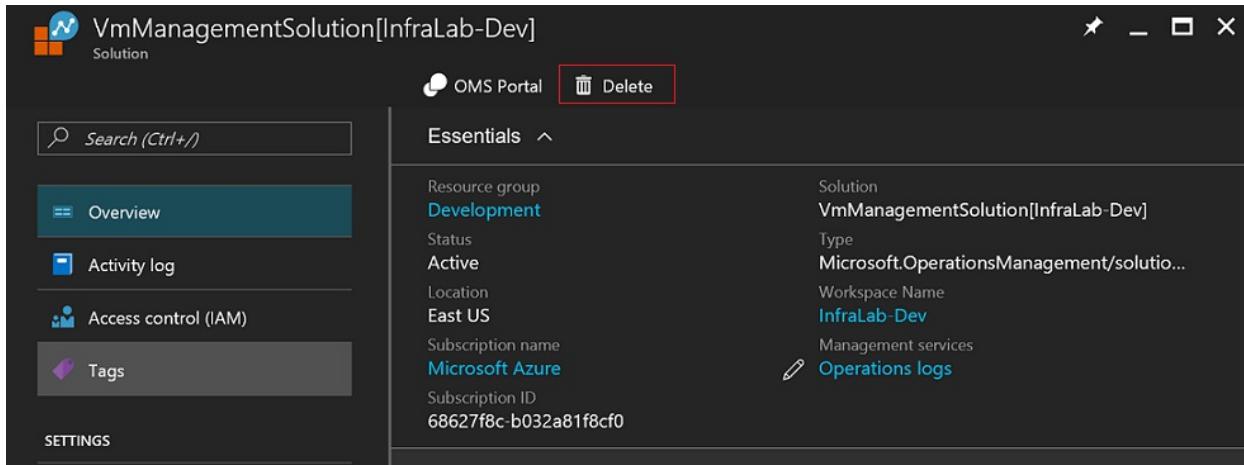
QUERY	DESCRIPTION
Find jobs for runbook StartVM that have completed successfully	Category=JobLogs RunbookName_s="StartByResourceGroup-MS-Mgmt-VM" ResultType=Succeeded measure count() by JobId_g
Find jobs for runbook StopVM that have completed successfully	Category=JobLogs RunbookName_s="StartByResourceGroup-MS-Mgmt-VM" ResultType=Failed measure count() by JobId_g
Show job status over time for StartVM and StopVM runbooks	Category=JobLogs RunbookName_s="StartByResourceGroup-MS-Mgmt-VM" OR "StopByResourceGroup-MS-Mgmt-VM" NOT(ResultType="started")

Removing the solution

If you decide you no longer need to use the solution any further, you can delete it from the Automation account. Deleting the solution will only remove the runbooks, it will not delete the schedules or variables that were created when the solution was added. Those assets you will need to delete manually if you are not using them with other runbooks.

To delete the solution, perform the following steps:

1. From your automation account, select the **Solutions** tile.
2. On the **Solutions** blade, select the solution **Start-Stop-VM[Workspace]**. On the **VMMManagementSolution[Workspace]** blade, from the menu click **Delete**.



3. In the **Delete Solution** window, confirm you want to delete the solution.
4. While the information is verified and the solution is deleted, you can track its progress under **Notifications** from the menu. You will be returned to the **VMMManagementSolution[Workspace]** blade after the process to remove solution starts.

The Automation account and OMS workspace are not deleted as part of this process. If you do not want to retain the OMS workspace, you will need to manually delete it. This can be accomplished also from the Azure portal. From the home-screen in the Azure portal, select **Log Analytics** and then on the **Log Analytics** blade, select the workspace and click **Delete** from the menu on the workspace settings blade.

Next steps

- To learn more about how to construct different search queries and review the Automation job logs with Log Analytics, see [Log searches in Log Analytics](#)
- To learn more about runbook execution, how to monitor runbook jobs, and other technical details, see [Track a runbook job](#)
- To learn more about OMS Log Analytics and data collection sources, see [Collecting Azure storage data in Log Analytics overview](#)

Azure Automation scenario - Automation source control integration with GitHub Enterprise

1/17/2017 • 4 min to read • [Edit on GitHub](#)

Automation currently supports source control integration, which allows you to associate runbooks in your Automation account to a GitHub source control repository. However, customers who have deployed [GitHub Enterprise](#) to support their DevOps practices, also want to use it to manage the lifecycle of runbooks that are developed to automate business processes and service management operations.

In this scenario, you will have a Windows computer in your data center configured as a Hybrid Runbook Worker with the Azure RM modules and Git tools installed. The Hybrid worker machine will have a clone of the local Git repository. When the runbook is run on the hybrid worker, the Git directory is synchronized and the runbook file contents are imported into the Automation account.

This article describes how to set up this configuration in your Azure Automation environment. We will start by configuring Automation with the security credentials, runbooks required to support this scenario, and deployment of a Hybrid Runbook Worker in your data center to run the runbooks and access your GitHub Enterprise repository to synchronize runbooks with your Automation account.

Getting the scenario

This scenario consists of two PowerShell runbooks that you can import directly from the [Runbook Gallery](#) in the Azure portal or download from the [PowerShell Gallery](#).

Runbooks

RUNBOOK	DESCRIPTION
Export-RunAsCertificateToHybridWorker	Runbook will export a RunAs certificate from an Automation account to a hybrid worker so that runbooks on the worker can authenticate with Azure in order to import runbooks into the Automation account.
Sync-LocalGitFolderToAutomationAccount	Runbook will sync the local Git folder on the hybrid machine and then import the runbook files (*.ps1) into the Automation account.

Credentials

CREDENTIAL	DESCRIPTION
GitHRWCredential	Credential asset you will create that contains the username and password for a user with permissions to the hybrid worker.

Installing and configuring this scenario

Prerequisites

1. The Sync-LocalGitFolderToAutomationAccount runbook authenticates using the [Azure Run As account](#).
2. A Microsoft Operations Management Suite (OMS) workspace with the Azure Automation solution enabled

and configured is also required. If you do not have one that is associated with the Automation account used to install and configure this scenario, it will be created and configured for you when you execute the **New-OnPremiseHybridWorker.ps1** script from the hybrid runbook worker.

NOTE

Currently these are the only regions supported for Automation integration with OMS - **Australia Southeast, East US 2, Southeast Asia, and West Europe**.

3. A computer that can serve as a dedicated Hybrid Runbook Worker that will also host the GitHub software and maintain the runbook files (*runbook.ps1*) in a source directory on the file system to synchronize between GitHub and your Automation account.

Import and publish the runbooks

To import the *Export-RunAsCertificateToHybridWorker* and *Sync-LocalGitFolderToAutomationAccount* runbooks from the Runbook Gallery from your Automation account in the Azure portal, please follow the procedures in [Import Runbook from the Runbook Gallery](#). Publish the runbooks after they have been successfully imported into your Automation account.

Deploy and Configure Hybrid Runbook Worker

If you do not have a Hybrid Runbook Worker already deployed in your data center, you should review the requirements and follow the automated installation steps using the procedure in [Azure Automation Hybrid Runbook Workers - Automate Install and Configuration](#). Once you have successfully installed the hybrid worker on a computer, perform the following steps to complete its configuration to support this scenario.

1. Log onto the computer hosting the Hybrid Runbook Worker role with an account that has local administrative rights and create a directory to hold the Git runbook files. Clone the internal Git repository to the directory.
2. If you do not already have a RunAs account created or you want to create a new one dedicated for this purpose, from the Azure portal navigate to Automation accounts, select your Automation account and create a [credential asset](#) that contains the username and password for a user with permissions to the hybrid worker.
3. From your Automation account, [edit the runbook Export-RunAsCertificateToHybridWorker](#) and modify the value for the variable *\$Password* with a strong password. After you modify the value, click **Publish** to have the draft version of the runbook published.
4. Start the runbook **Export-RunAsCertificateToHybridWorker**, and in the **Start Runbook** blade, under the option **Run settings** select the option **Hybrid Worker** and in the drop-down list select the Hybrid worker group you created earlier for this scenario.

This will export a certificate to the hybrid worker so that runbooks on the worker can authenticate with Azure using the Run As connection in order to manage Azure resources (in particular for this scenario - import runbooks to the Automation account).

5. From your Automation account, select the Hybrid worker group created earlier and [specify a RunAs account](#) for the for the Hybrid worker group, and chose the credential asset you just or already have created. This assures that the Sync runbook can run Git commands.
6. Start the runbook **Sync-LocalGitFolderToAutomationAccount**, provide the following required input parameter values and in the **Start Runbook** blade, under the option **Run settings** select the option **Hybrid Worker** and in the drop-down list select the Hybrid worker group you created earlier for this scenario:
 - *ResourceGroup* - the name of your resource group associated with your Automation account
 - *AutomationAccountName* - the name of your Automation account
 - *GitPath* - The local folder or file on the Hybrid Runbook Worker where Git is set up to pull latest changes into

This will sync the local Git folder on the hybrid worker computer and then import the .ps1 files from the source directory to the Automation account.

The screenshot shows the Azure portal interface with two runbooks displayed side-by-side.

Runbook Details (Left):

- Resource group: InfraLab
- Status: Published
- Runbook type: PowerShell Runbook
- Last modified: 11/23/2016 3:07 PM
- Last modified by: AutomationAcct@InfraLab ...

Runbook Parameters (Right):

- * RESOURCEGROUP: InfraLab (Mandatory, String)
- * AUTOMATIONACCOUNTNAME: AzureAutomation (Mandatory, String)
- * GITPATH: C:\GitRepo\RunbookSrc (Mandatory, String)

Run Settings (Right):

- Run on: Hybrid Worker
- Choose Hybrid Worker group: HRWG

- View job summary details for the runbook by selecting it from the **Runbooks** blade in your Automation account, and then select the **Jobs** tile. Confirm it completed successfully by selecting the **All logs** tile and reviewing the detailed log stream.

Next steps

- To know more about runbook types, their advantages and limitations, see [Azure Automation runbook types](#)
- For more information on PowerShell script support feature, see [Native PowerShell script support in Azure Automation](#)

Azure Automation scenario - Automation source control integration with Visual Studio Team Services

2/21/2017 • 3 min to read • [Edit on GitHub](#)

In this scenario, you have a Visual Studio Team Services project that you are using to manage Azure Automation runbooks or DSC configurations under source control. This article describes how to integrate VSTS with your Azure Automation environment so that continuous integration happens for each check-in.

Getting the scenario

This scenario consists of two PowerShell runbooks that you can import directly from the [Runbook Gallery](#) in the Azure portal or download from the [PowerShell Gallery](#).

Runbooks

RUNBOOK	DESCRIPTION
Sync-VSTS	Import runbooks or configurations from VSTS source control when a check-in is done. If run manually, it will import and publish all runbooks or configurations into the Automation account.
Sync-VSTSGit	Import runbooks or configurations from VSTS under Git source control when a check-in is done. If run manually, it will import and publish all runbooks or configurations into the Automation account.

Variables

VARIABLE	DESCRIPTION
VSToken	Secure variable asset you will create that contains the VSTS personal access token. You can learn how to create a VSTS personal access token on the VSTS authentication page .

Installing and configuring this scenario

Create a [personal access token](#) in VSTS that you will use to sync the runbooks or configurations into your automation account.

The screenshot shows the 'Personal access tokens' section of the VSTS settings. It displays a table with one row:

Description	Expiration	Status	Actions
ContosoDev	1/20/2018 10:01:12 PM	Active	Revoke

Create a [secure variable](#) in your automation account to hold the personal access token so that the runbook can authenticate to VSTS and sync the runbooks or configurations into the Automation account. You can name this VSToken.

Import the runbook that will sync your runbooks or configurations into the automation account. You can use the [VSTS sample runbook](#) or the [VSTS with Git sample runbook](#) from the PowerShellGallery.com depending on if you use VSTS source control or VSTS with Git and deploy to your automation account.

You can now [publish](#) this runbook so you can create a webhook.

```

1 <#
2 .SYNOPSIS
3     This Azure Automation runbook syncs runbook and configurations from VSTS source control. It requires that a
4     service hook be set up in VSTS to trigger this runbook when changes are made.
5
6 .DESCRIPTION
7     This Azure Automation runbook syncs runbook and configurations from VSTS source control. It requires that a
8     service hook be set up in VSTS to trigger this runbook when changes are made. It can also be run without a
9     service hook to force a sync of everything from VSTS folder.
10    It requires that you have the RunAs account configured in the automation service.
11
12    This enables continuous integration with VSTS source control and an automation account.
13
14 .PARAMETER WebhookData
15     Optional. This will contain the change that was made in VSTS and sent over to the runbook through a
16     service hook call.
17
18 .PARAMETER ResourceGroup
19     Required. The name of the resource group the automation account is in.
20
21 .PARAMETER AutomationAccountName
22     Required. The name of the Automation account to sync all the runbooks and configurations to
23
24 .PARAMETER VSFolder
25     Required. The name of the folder in VSTS where the runbooks and configurations exist.
26     This should look like '$/ContosoDev/AutomationScriptsConfigurations'

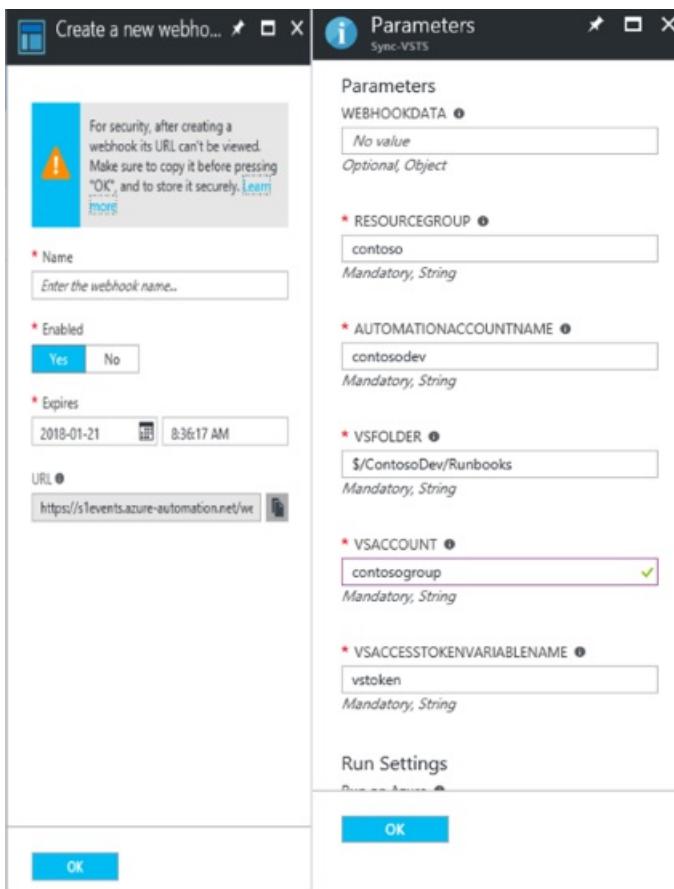
```

Create a [webhook](#) for this Sync-VSTS runbook and fill in the parameters as shown below. Make sure you copy the webhook url as you will need it for a service hook in VSTS. The VSAccessTokenVariableName is the name (VSToken) of the secure variable that you created earlier to hold the personal access token.

Integrating with VSTS (Sync-VSTS.ps1) will take the following parameters.

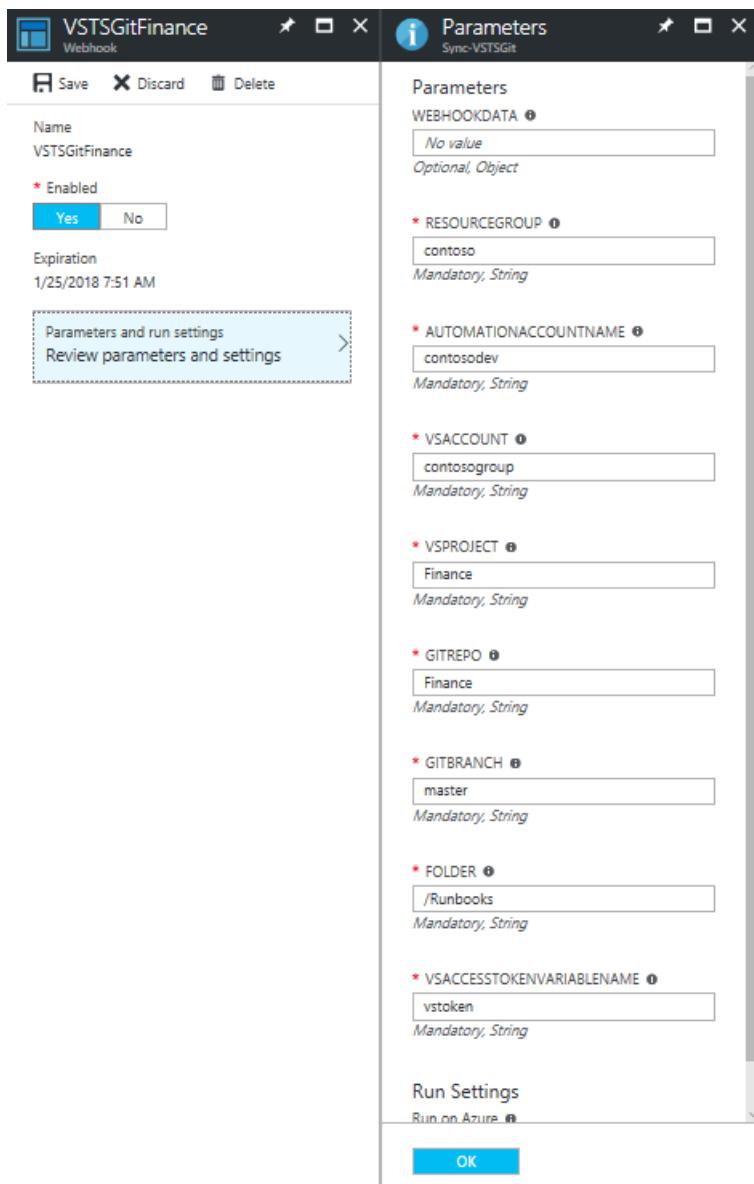
Sync-VSTS Parameters

PARAMETER	DESCRIPTION
WebhookData	This will contain the checkin information sent from the VSTS service hook. You should leave this parameter blank.
ResourceGroup	This is the name of the resource group that the automation account is in.
AutomationAccountName	The name of the automation account that will sync with VSTS.
VSFolder	The name of the folder in VSTS where the runbooks and configurations exist.
VSAccount	The name of the Visual Studio Team Services account.
VSAccessTokenVariableName	The name of the secure variable (VSToken) that holds the VSTS personal access token.



If you are using VSTS with GIT (Sync-VSTSGit.ps1) it will take the following parameters.

PARAMETER	DESCRIPTION
WebhookData	This will contain the checkin information sent from the VSTS service hook. You should leave this parameter blank.
AutomationAccountName	The name of the automation account that will sync with VSTS.
VSAccount	The name of the Visual Studio Team Services account.
VSProject	The name of the project in VSTS where the runbooks and configurations exist.
GitRepo	The name of the Git repository.
GitBranch	The name of the branch in VSTS Git repository.
Folder	The name of the folder in VSTS Git branch.
VSAccessTokenVariableName	The name of the secure variable (VSToken) that holds the VSTS personal access token.



Create a service hook in VSTS for check-ins to the folder that triggers this webhook on code check-in. Select Web Hooks as the service to integrate with when you create a new subscription. You can learn more about service hooks on [VSTS Service Hooks documentation](#).

The screenshot shows the 'Service Hooks' section in the VSTS navigation bar. A modal window titled 'EDIT SERVICE HOOKS SUBSCRIPTION' is open, specifically for the 'Action' configuration. The action type is set to 'Post via HTTP'. The URL is specified as `https://st1events.azure-automation.net/webhooks?token=token%2f%2bU3ZwDV`. Other fields like 'HTTP headers', 'Basic authentication username', and 'Basic authentication password' are optional. At the bottom of the modal are buttons for 'Previous', 'Next', 'Test', 'Finish', and 'Cancel'.

You should now be able to do all check-ins of your runbooks and configurations into VSTS and have these automatically sync'd into your automation account.

The screenshot shows the 'Sync-VSTS 1/21/2017 8:14 AM' runbook job summary. It indicates the job is completed with 6 inputs and 0 errors. The output pane shows the command 'Syncing \$/ContosoDev/Runbooks/test2.ps1' was run successfully, with the message 'Removing runbook test1'.

If you run this runbook manually without being triggered by VSTS, you can leave the webhookdata parameter empty and it will do a full sync from the VSTS folder specified.

If you wish to uninstall the scenario, remove the service hook from VSTS, delete the runbook, and the VSToken variable.

Forward job status and job streams from Automation to Log Analytics (OMS)

2/24/2017 • 7 min to read • [Edit on GitHub](#)

Automation can send runbook job status and job streams to your Microsoft Operations Management Suite (OMS) Log Analytics workspace. Job logs and job streams are visible in the Azure portal, or with PowerShell, for individual jobs and this allows you to perform simple investigations. Now with Log Analytics you can:

- Get insight on your Automation jobs
- Trigger an email or alert based on your runbook job status (for example, failed or suspended)
- Write advanced queries across your job streams
- Correlate jobs across Automation accounts
- Visualize your job history over time

Prerequisites and deployment considerations

To start sending your Automation logs to Log Analytics, you need:

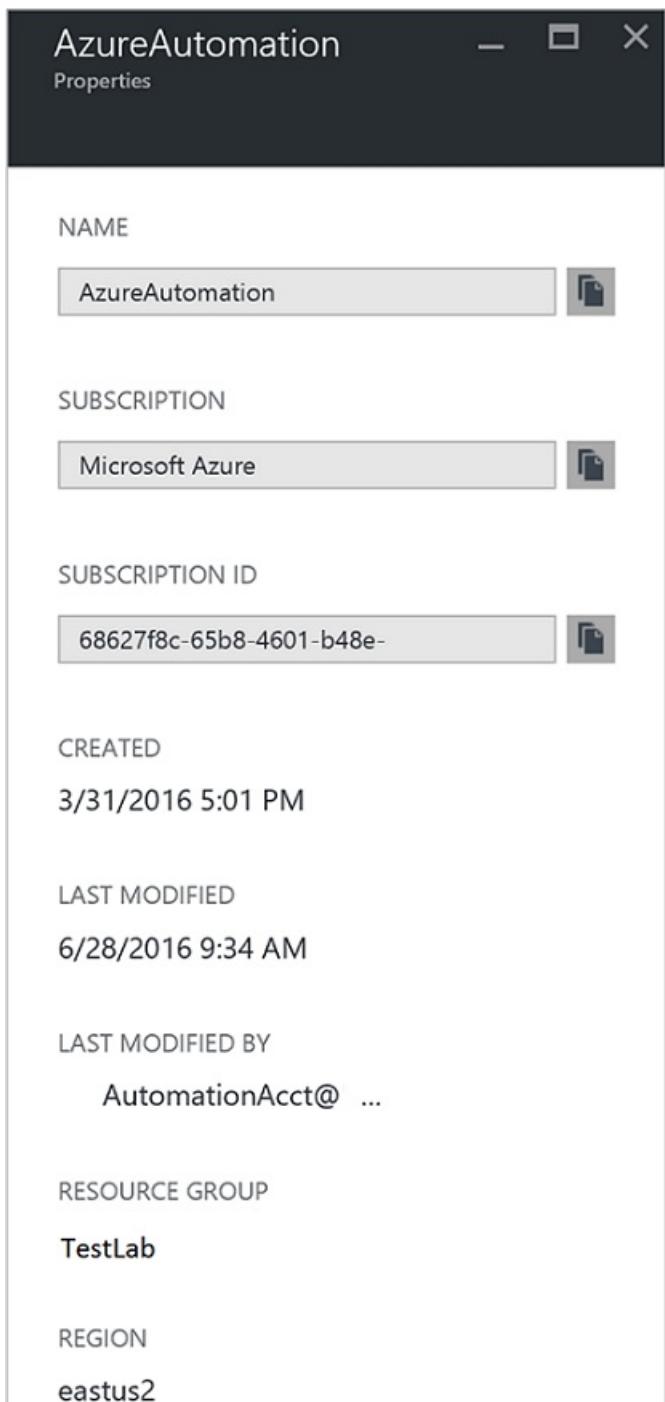
1. The November 2016 or later release of [Azure PowerShell](#) (v2.3.0).
2. A Log Analytics workspace. For more information, see [Get started with Log Analytics](#).
3. The ResourceId for your Azure Automation account

To find the ResourceId for your Azure Automation account and Log Analytics workspace, run the following PowerShell:

```
# Find the ResourceId for the Automation Account  
Find-AzureRmResource -ResourceType "Microsoft.Automation/automationAccounts"  
  
# Find the ResourceId for the Log Analytics workspace  
Find-AzureRmResource -ResourceType "Microsoft.OperationalInsights/workspaces"
```

If you have multiple Automation accounts, or workspaces, in the output of the preceding commands, find the *Name* you need to configure and copy the value for *ResourceId*.

If you need to find the *Name* of your Automation account, in the Azure portal select your Automation account from the **Automation account** blade and select **All settings**. From the **All settings** blade, under **Account Settings** select **Properties**. In the **Properties** blade, you can note these values.



Set up integration with Log Analytics

1. On your computer, start **Windows PowerShell** from the **Start** screen.
2. Copy and paste the following PowerShell, and edit the value for the `$workspaceId` and `$automationAccountId`.
For the `-Environment` parameter, valid values are *AzureCloud* or *AzureUSGovernment* depending on the cloud environment you are working in.

```

[cmdletBinding()]
Param
(
    [Parameter(Mandatory=$True)]
    [ValidateSet("AzureCloud", "AzureUSGovernment")]
    [string]$Environment="AzureCloud"
)

#Check to see which cloud environment to sign into.
Switch ($Environment)
{
    "AzureCloud" {Login-AzureRmAccount}
    "AzureUSGovernment" {Login-AzureRmAccount -EnvironmentName AzureUSGovernment}
}

# if you have one Log Analytics workspace you can use the following command to get the resource id of the
workspace
$workspaceId = (Get-AzureRmOperationalInsightsWorkspace).ResourceId

$automationAccountId = "/SUBSCRIPTIONS/ec11ca60-1234-491e-5678-
0ea07feae25c/RESOURCEGROUPS/DEMO/PROVIDERS/MICROSOFT.AUTOMATION/ACCOUNTS/DEMO"

Set-AzureRmDiagnosticSetting -ResourceId $automationAccountId -WorkspaceId $workspaceId -Enabled $true

```

After running this script, you will see records in Log Analytics within 10 minutes of new JobLogs or JobStreams being written.

To see the logs, run the following query: `Type=AzureDiagnostics ResourceProvider="MICROSOFT.AUTOMATION"`

Verify configuration

To confirm that your Automation account is sending logs to your Log Analytics workspace, check that diagnostics are set correctly on the Automation account using the following PowerShell:

```

[cmdletBinding()]
Param
(
    [Parameter(Mandatory=$True)]
    [ValidateSet("AzureCloud", "AzureUSGovernment")]
    [string]$Environment="AzureCloud"
)

#Check to see which cloud environment to sign into.
Switch ($Environment)
{
    "AzureCloud" {Login-AzureRmAccount}
    "AzureUSGovernment" {Login-AzureRmAccount -EnvironmentName AzureUSGovernment}
}

# if you have one Log Analytics workspace you can use the following command to get the resource id of the
workspace
$workspaceId = (Get-AzureRmOperationalInsightsWorkspace).ResourceId

$automationAccountId = "/SUBSCRIPTIONS/ec11ca60-1234-491e-5678-
0ea07feae25c/RESOURCEGROUPS/DEMO/PROVIDERS/MICROSOFT.AUTOMATION/ACCOUNTS/DEMO"

Get-AzureRmDiagnosticSetting -ResourceId $automationAccountId

```

In the output ensure that:

- Under *Logs*, the value for *Enabled* is *True*
- The value of *Workspaceld* is set to the *Resourceld* of your Log Analytics workspace

Log Analytics records

Diagnostics from Azure Automation creates two types of records in Log Analytics.

Job Logs

PROPERTY	DESCRIPTION
TimeGenerated	Date and time when the runbook job executed.
RunbookName_s	The name of the runbook.
Caller_s	Who initiated the operation. Possible values are either an email address or system for scheduled jobs.
Tenant_g	GUID that identifies the tenant for the Caller.
JobId_g	GUID that is the Id of the runbook job.
ResultType	The status of the runbook job. Possible values are: <ul style="list-style-type: none">- Started- Stopped- Suspended- Failed- Succeeded
Category	Classification of the type of data. For Automation, the value is JobLogs.
OperationName	Specifies the type of operation performed in Azure. For Automation, the value is Job.
Resource	Name of the Automation account
SourceSystem	How Log Analytics collected the data. Always <i>Azure</i> for Azure diagnostics.
ResultDescription	Describes the runbook job result state. Possible values are: <ul style="list-style-type: none">- Job is started- Job Failed- Job Completed
CorrelationId	GUID that is the Correlation Id of the runbook job.
ResourceId	Specifies the Azure Automation account resource id of the runbook.
SubscriptionId	The Azure subscription Id (GUID) for the Automation account.
ResourceGroup	Name of the resource group for the Automation account.
ResourceProvider	MICROSOFT.AUTOMATION
ResourceType	AUTOMATIONACCOUNTS

Job Streams

PROPERTY	DESCRIPTION
TimeGenerated	Date and time when the runbook job executed.
RunbookName_s	The name of the runbook.
Caller_s	Who initiated the operation. Possible values are either an email address or system for scheduled jobs.
StreamType_s	The type of job stream. Possible values are: -Progress - Output - Warning - Error - Debug - Verbose
Tenant_g	GUID that identifies the tenant for the Caller.
JobId_g	GUID that is the Id of the runbook job.
ResultType	The status of the runbook job. Possible values are: - In Progress
Category	Classification of the type of data. For Automation, the value is JobStreams.
OperationName	Specifies the type of operation performed in Azure. For Automation, the value is Job.
Resource	Name of the Automation account
SourceSystem	How Log Analytics collected the data. Always <i>Azure</i> for Azure diagnostics.
ResultDescription	Includes the output stream from the runbook.
CorrelationId	GUID that is the Correlation Id of the runbook job.
ResourceId	Specifies the Azure Automation account resource id of the runbook.
SubscriptionId	The Azure subscription Id (GUID) for the Automation account.
ResourceGroup	Name of the resource group for the Automation account.
ResourceProvider	MICROSOFT.AUTOMATION
ResourceType	AUTOMATIONACCOUNTS

Viewing Automation Logs in Log Analytics

Now that you have started sending your Automation job logs to Log Analytics, let's see what you can do with these logs inside Log Analytics.

To see the logs, run the following query:

`Type=AzureDiagnostics ResourceProvider="MICROSOFT.AUTOMATION"`

Send an email when a runbook job fails or suspends

One of our top customer asks is for the ability to send an email or a text when something goes wrong with a runbook job.

To create an alert rule, you start by creating a log search for the runbook job records that should invoke the alert. Click the **Alert** button to create and configure the alert rule.

1. From the Log Analytics Overview page, click **Log Search**.
2. Create a log search query for your alert by typing the following search into the query field:

`Type=AzureDiagnostics ResourceProvider="MICROSOFT.AUTOMATION" Category=JobLogs (ResultType=Failed OR ResultType=Suspended)`

You can also group by the RunbookName by using:

`Type=AzureDiagnostics ResourceProvider="MICROSOFT.AUTOMATION" Category=JobLogs (ResultType=Failed OR ResultType=Suspended) | measure Count() by RunbookName_s`

If you have set up logs from more than one Automation account or subscription to your workspace, you can group your alerts by subscription and Automation account. Automation account name can be derived from the Resource field in the search of JobLogs.

3. To open the **Add Alert Rule** screen, click **Alert** at the top of the page. For further details on the options to configure the alert, see [Alerts in Log Analytics](#).

Find all jobs that have completed with errors

In addition to alerting on failures, you can find when a runbook job has a non-terminating error. In these cases PowerShell produces an error stream, but the non-terminating errors do not cause your job to suspend or fail.

1. In your Log Analytics workspace, click **Log Search**.
2. In the query field, type

`Type=AzureDiagnostics ResourceProvider="MICROSOFT.AUTOMATION" Category=JobStreams StreamType_s=Error | measure count() by JobId_g`

and then click **Search**.

View job streams for a job

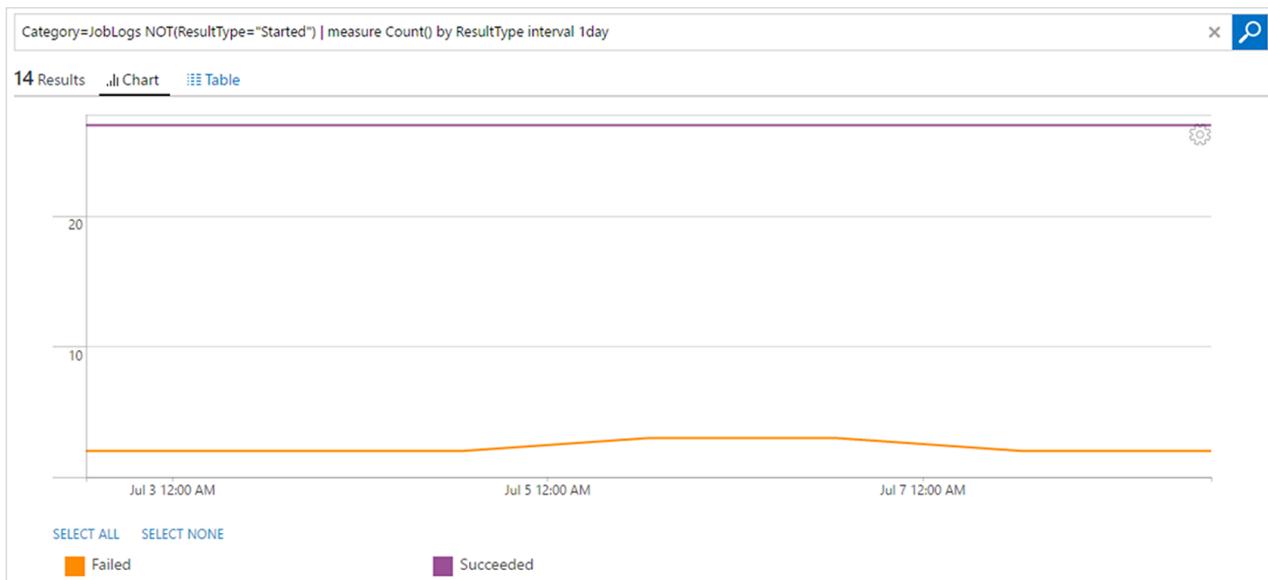
When you are debugging a job, you may also want to look into the job streams. The following query shows all the streams for a single job with GUID 2ebd22ea-e05e-4eb9-9d76-d73cbd4356e0:

`Type=AzureDiagnostics ResourceProvider="MICROSOFT.AUTOMATION" Category=JobStreams JobId_g="2ebd22ea-e05e-4eb9-9d76-d73cbd4356e0" | sort TimeGenerated | select ResultDescription`

View historical job status

Finally, you may want to visualize your job history over time. You can use this query to search for the status of your jobs over time.

`Type=AzureDiagnostics ResourceProvider="MICROSOFT.AUTOMATION" Category=JobLogs NOT(ResultType="started") | measure Count() by ResultType interval 1hour`



Summary

By sending your Automation job status and stream data to Log Analytics, you can get better insight into the status of your Automation jobs by:

- Setting up alerts to notify you when there is an issue
- Using custom views and search queries to visualize your runbook results, runbook job status, and other related key indicators or metrics.

Log Analytics provides greater operational visibility to your Automation jobs and can help address incidents quicker.

Next steps

- To learn more about how to construct different search queries and review the Automation job logs with Log Analytics, see [Log searches in Log Analytics](#)
- To understand how to create and retrieve output and error messages from runbooks, see [Runbook output and messages](#)
- To learn more about runbook execution, how to monitor runbook jobs, and other technical details, see [Track a runbook job](#)
- To learn more about OMS Log Analytics and data collection sources, see [Collecting Azure storage data in Log Analytics overview](#)

How to unlink your Automation account from a Log Analytics workspace

2/7/2017 • 1 min to read • [Edit on GitHub](#)

Azure Automation integrates with Log Analytics to not only support proactive monitoring of your runbook jobs across all of your Automation accounts, but is also required when you import the following solutions that are dependent on Log Analytics:

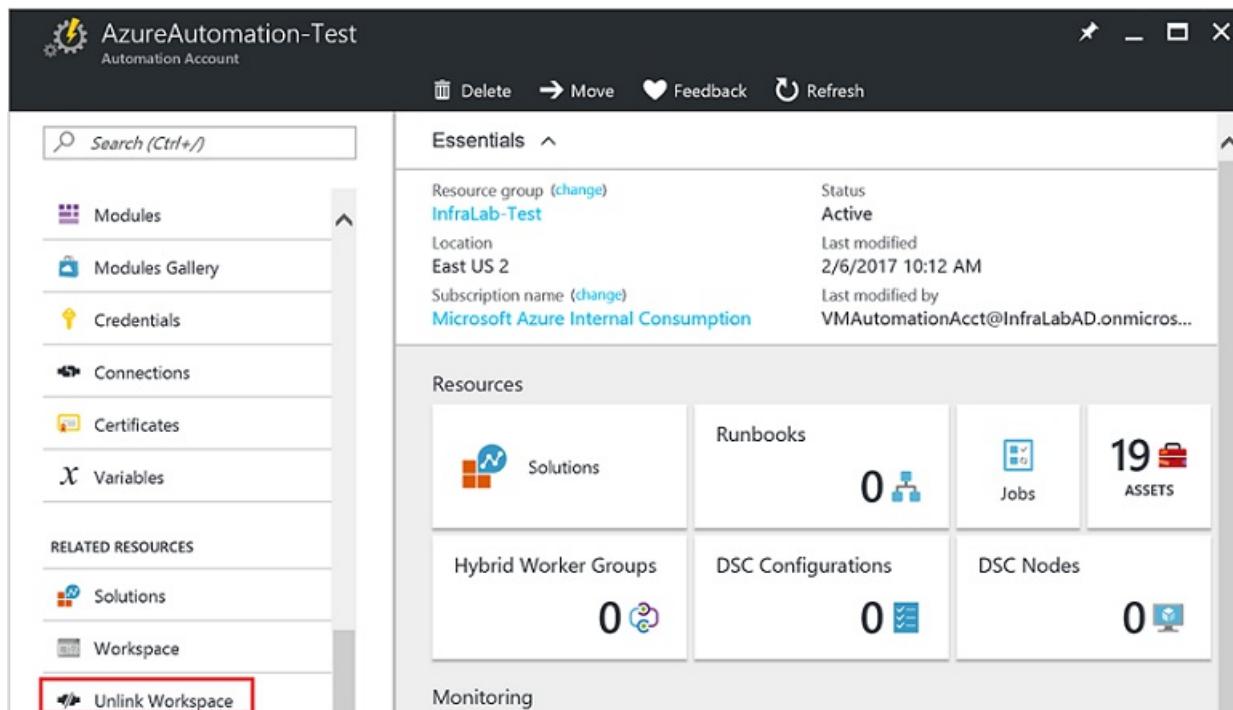
- [Update Management](#)
- [Change Tracking](#)
- [Start/Stop VMs during off-hours](#)

If you decide you no longer wish to integrate your Automation account with Log Analytics, you can unlink your account directly from the Azure portal. Before you proceed, you will first need to remove the solutions mentioned earlier, otherwise this process will be prevented from proceeding. Review the topic for the particular solution you have imported to understand the steps required to remove it.

After you remove these solutions you can perform the following steps to unlink your Automation account.

Unlink workspace

1. From the Azure portal, open your Automation account, and in the Automation account blade, in the account blade, select **Unlink workspace**.



2. On the Unlink workspace blade, click **Unlink workspace**.

The screenshot shows the Azure Automation - Unlink Workspace page. On the left, there's a sidebar with a search bar and links to Modules, Modules Gallery, Credentials, Connections, Certificates, Variables, Solutions, Workspace, and Unlink Workspace. The 'Unlink Workspace' link is highlighted with a blue background. The main content area has a heading 'Unlink Workspace' and instructions about removing dependencies before unlinking. It lists three items: Update Management, ChangeTracking, and Start/Stop VMs during off-hours. Below this, it says 'After you remove these solutions you can click **Unlink Workspace** above to complete the unlinking.' It also notes that if the Update Management solution is used, other items like update schedules and hybrid worker groups may be removed. A link 'Learn how to remove hybrid worker groups!' is provided.

You will receive a prompt verifying you wish to proceed.

3. While Azure Automation attempts to unlink the account your Log Analytics workspace, you can track the progress under **Notifications** from the menu.

If you used the Update Management solution, optionally you may want to remove the following items that are no longer needed after you remove the solution.

- Update schedules. Each will have names that match the update deployments you created)
- Hybrid worker groups created for the solution. Each will be named similarly to machine1.contoso.com_9ceb8108-26c9-4051-b6b3-227600d715c8).

If you used the Start/Stop VMs during off-hours solution, optionally you may want to remove the following items that are no longer needed after you remove the solution.

- Start and stop VM runbook schedules
- Start and stop VM runbooks
- Variables

Next steps

To reconfigure your Automation account to integrate with OMS Log Analytics, see [Forward job status and job streams from Automation to Log Analytics \(OMS\)](#).

Migrating from Orchestrator to Azure Automation (Beta)

1/17/2017 • 9 min to read • [Edit on GitHub](#)

Runbooks in [System Center Orchestrator](#) are based on activities from integration packs that are written specifically for Orchestrator while runbooks in Azure Automation are based on Windows PowerShell. [Graphical runbooks](#) in Azure Automation have a similar appearance to Orchestrator runbooks with their activities representing PowerShell cmdlets, child runbooks, and assets.

The [System Center Orchestrator Migration Toolkit](#) includes tools to assist you in converting runbooks from Orchestrator to Azure Automation. In addition to converting the runbooks themselves, you must convert the integration packs with the activities that the runbooks use to integration modules with Windows PowerShell cmdlets.

Following is the basic process for converting Orchestrator runbooks to Azure Automation. Each of these steps is described in detail in the sections below.

1. Download the [System Center Orchestrator Migration Toolkit](#) which contains the tools and modules discussed in this article.
2. Import [Standard Activities Module](#) into Azure Automation. This includes converted versions of standard Orchestrator activities that may be used by converted runbooks.
3. Import [System Center Orchestrator Integration Modules](#) into Azure Automation for those integration packs used by your runbooks that access System Center.
4. Convert custom and third party integration packs using the [Integration Pack Converter](#) and import into Azure Automation.
5. Convert Orchestrator runbooks using the [Runbook Converter](#) and install in Azure Automation.
6. Manually create required Orchestrator assets in Azure Automation since the Runbook Converter does not convert these resources.
7. Configure a [Hybrid Runbook Worker](#) in your local data center to run converted runbooks that will access local resources.

Service Management Automation

[Service Management Automation](#) (SMA) stores and runs runbooks in your local data center like Orchestrator, and it uses the same integration modules as Azure Automation. The [Runbook Converter](#) converts Orchestrator runbooks to graphical runbooks though which are not supported in SMA. You can still install the [Standard Activities Module](#) and [System Center Orchestrator Integration Modules](#) into SMA, but you must manually [rewrite your runbooks](#).

Hybrid Runbook Worker

Runbooks in Orchestrator are stored on a database server and run on runbook servers, both in your local data center. Runbooks in Azure Automation are stored in the Azure cloud and can run in your local data center using a [Hybrid Runbook Worker](#). This is how you will usually run runbooks converted from Orchestrator since they are designed to run on local servers.

Integration Pack Converter

The Integration Pack Converter converts integration packs that were created using the [Orchestrator Integration Toolkit \(OIT\)](#) to integration modules based on Windows PowerShell that can be imported into Azure Automation or

Service Management Automation.

When you run the Integration Pack Converter, you are presented with a wizard that will allow you to select an integration pack (.oip) file. The wizard then lists the activities included in that integration pack and allows you to select which will be migrated. When you complete the wizard, it creates an integration module that includes a corresponding cmdlet for each of the activities in the original integration pack.

Parameters

Any properties of an activity in the integration pack are converted to parameters of the corresponding cmdlet in the integration module. Windows PowerShell cmdlets have a set of [common parameters](#) that can be used with all cmdlets. For example, the -Verbose parameter causes a cmdlet to output detailed information about its operation. No cmdlet may have a parameter with the same name as a common parameter. If an activity does have a property with the same name as a common parameter, the wizard will prompt you to provide another name for the parameter.

Monitor activities

Monitor runbooks in Orchestrator start with a [monitor activity](#) and run continuously waiting to be invoked by a particular event. Azure Automation does not support monitor runbooks, so any monitor activities in the integration pack will not be converted. Instead, a placeholder cmdlet is created in the integration module for the monitor activity. This cmdlet has no functionality, but it allows any converted runbook that uses it to be installed. This runbook will not be able to run in Azure Automation, but it can be installed so that you can modify it.

Integration packs that cannot be converted

Integration packs that were not created with OIT cannot be converted with the Integration Pack Converter. There are also some integration packs provided by Microsoft that cannot currently be converted with this tool. Converted versions of these integration packs have been [provided for download](#) so that they can be installed in Azure Automation or Service Management Automation.

Standard Activities Module

Orchestrator includes a set of [standard activities](#) that are not included in an integration pack but are used by many runbooks. The Standard Activities module is an integration module that includes a cmdlet equivalent for each of these activities. You must install this integration module in Azure Automation before importing any converted runbooks that use a standard activity.

In addition to supporting converted runbooks, the cmdlets in the standard activities module can be used by someone familiar with Orchestrator to build new runbooks in Azure Automation. While the functionality of all of the standard activities can be performed with cmdlets, they may operate differently. The cmdlets in the converted standard activities module will work the same as their corresponding activities and use the same parameters. This can help the existing Orchestrator runbook author in their transition to Azure Automation runbooks.

System Center Orchestrator Integration Modules

Microsoft provides [integration packs](#) for building runbooks to automate System Center components and other products. Some of these integration packs are currently based on OIT but cannot currently be converted to integration modules because of known issues. [System Center Orchestrator Integration Modules](#) includes converted versions of these integration packs that can be imported into Azure Automation and Service Management Automation.

By the RTM version of this tool, updated versions of the integration packs based on OIT that can be converted with the Integration Pack Converter will be published. Guidance will also be provided to assist you in converting runbooks using activities from the integration packs not based on OIT.

Runbook Converter

The Runbook Converter converts Orchestrator runbooks into [graphical runbooks](#) that can be imported into Azure Automation.

Runbook Converter is implemented as a PowerShell module with a cmdlet called **ConvertFrom-SCORunbook** that performs the conversion. When you install the tool, it will create a shortcut to a PowerShell session that loads the cmdlet.

Following is the basic process to convert an Orchestrator runbook and import it into Azure Automation. The following sections provide further details on using the tool and working with converted runbooks.

1. Export one or more runbooks from Orchestrator.
2. Obtain integration modules for all activities in the runbook.
3. Convert the Orchestrator runbooks in the exported file.
4. Review information in logs to validate the conversion and to determine any required manual tasks.
5. Import converted runbooks into Azure Automation.
6. Create any required assets in Azure Automation.
7. Edit the runbook in Azure Automation to modify any required activities.

Using Runbook Converter

The syntax for **ConvertFrom-SCORunbook** is as follows:

```
ConvertFrom-SCORunbook -RunbookPath <string> -Module <string[]> -OutputFolder <string>
```

- RunbookPath - Path to the export file containing the runbooks to convert.
- Module - Comma delimited list of integration modules containing activities in the runbooks.
- OutputFolder - Path to the folder to create converted graphical runbooks.

The following example command converts the runbooks in an export file called **MyRunbooks.ois_export**. These runbooks use the Active Directory and Data Protection Manager integration packs.

```
ConvertFrom-SCORunbook -RunbookPath "c:\runbooks\MyRunbooks.ois_export" -Module c:\ip\SystemCenter_IntegrationModule_ActiveDirectory.zip,c:\ip\SystemCenter_IntegrationModule_DPM.zip -OutputFolder "c:\runbooks"
```

Log files

The Runbook Converter will create the following log files in the same location as the converted runbook. If the files already exist, then they will be overwritten with information from the last conversion.

FILE	CONTENTS
Runbook Converter - Progress.log	Detailed steps of the conversion including information for each activity successfully converted and warning for each activity not converted.
Runbook Converter - Summary.log	Summary of the last conversion including any warnings and follow up tasks that you need to perform such as creating a variable required for the converted runbook.

Exporting runbooks from Orchestrator

The Runbook Converter works with an export file from Orchestrator that contains one or more runbooks. It will create a corresponding Azure Automation runbook for each Orchestrator runbook in the export file.

To export a runbook from Orchestrator, right-click the name of the runbook in Runbook Designer and select **Export**. To export all runbooks in a folder, right-click the name of the folder and select **Export**.

Runbook activities

The Runbook Converter converts each activity in the Orchestrator runbook to a corresponding activity in Azure Automation. For those activities that can't be converted, a placeholder activity is created in the runbook with warning text. After you import the converted runbook into Azure Automation, you must replace any of these activities with valid activities that perform the required functionality.

Any Orchestrator activities in the [Standard Activities Module](#) will be converted. There are some standard Orchestrator activities that are not in this module though and are not converted. For example, **Send Platform Event** has no Azure Automation equivalent since the event is specific to Orchestrator.

[Monitor activities](#) are not converted since there is no equivalent to them in Azure Automation. The exception are monitor activities in [converted integration packs](#) that will be converted to the placeholder activity.

Any activity from a [converted integration pack](#) will be converted if you provide the path to the integration module with the **modules** parameter. For System Center Integration Packs, you can use the [System Center Orchestrator Integration Modules](#).

Orchestrator resources

The Runbook Converter only converts runbooks, not other Orchestrator resources such as counters, variables, or connections. Counters are not supported in Azure Automation. Variables and connections are supported, but you must create them manually. The log files will inform you if the runbook requires such resources and specify corresponding resources that you need to create in Azure Automation for the converted runbook to operate properly.

For example, a runbook may use a variable to populate a particular value in an activity. The converted runbook will convert the activity and specify a variable asset in Azure Automation with the same name as the Orchestrator variable. This will be noted in the **Runbook Converter - Summary.log** file that is created after the conversion. You will need to manually create this variable asset in Azure Automation before using the runbook.

Input parameters

Runbooks in Orchestrator accept input parameters with the **Initialize Data** activity. If the runbook being converted includes this activity, then an [input parameter](#) in the Azure Automation runbook is created for each parameter in the activity. A [Workflow Script control](#) activity is created in the converted runbook that retrieves and returns each parameter. Any activities in the runbook that use an input parameter refer to the output from this activity.

The reason that this strategy is used is to best mirror the functionality in the Orchestrator runbook. Activities in new graphical runbooks should refer directly to input parameters using a Runbook input data source.

Invoke Runbook activity

Runbooks in Orchestrator start other runbooks with the **Invoke Runbook** activity. If the runbook being converted includes this activity and the **Wait for completion** option is set, then a runbook activity is created for it in the converted runbook. If the **Wait for completion** option is not set, then a Workflow Script activity is created that uses **Start-AzureAutomationRunbook** to start the runbook. After you import the converted runbook into Azure Automation, you must modify this activity with the information specified in the activity.

Related articles

- [System Center 2012 - Orchestrator](#)
- [Service Management Automation](#)
- [Hybrid Runbook Worker](#)
- [Orchestrator Standard Activities](#)
- [Download System Center Orchestrator Migration Toolkit](#)

Migrate Automation Account and resources

1/17/2017 • 2 min to read • [Edit on GitHub](#)

For Automation accounts and its associated resources (i.e. assets, runbooks, modules, etc.) that you have created in the Azure portal and want to migrate from one resource group to another or from one subscription to another, you can accomplish this easily with the [move resources](#) feature available in the Azure portal. However, before proceeding with this action, you should first review the following [checklist before moving resources](#) and additionally, the list below specific to Automation.

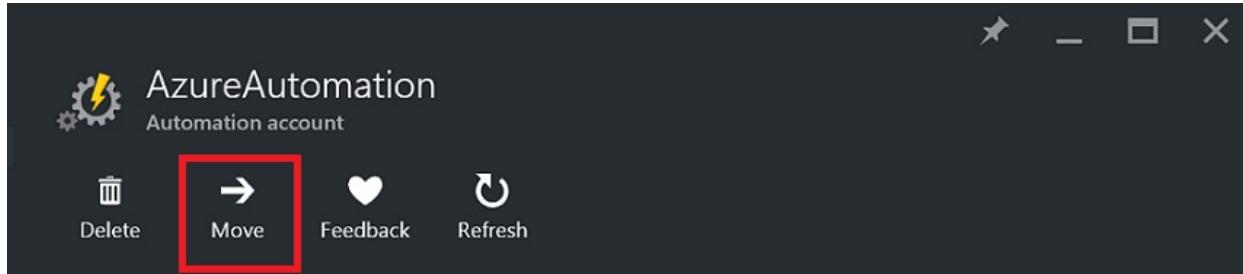
1. The destination subscription/resource group must be in same region as the source. Meaning, Automation accounts cannot be moved across regions.
2. When moving resources (e.g. runbooks, jobs, etc.), both the source group and the target group are locked for the duration of the operation. Write and delete operations are blocked on the groups until the move completes.
3. Any runbooks or variables which reference a resource or subscription ID from the existing subscription will need to be updated after migration is completed.

NOTE

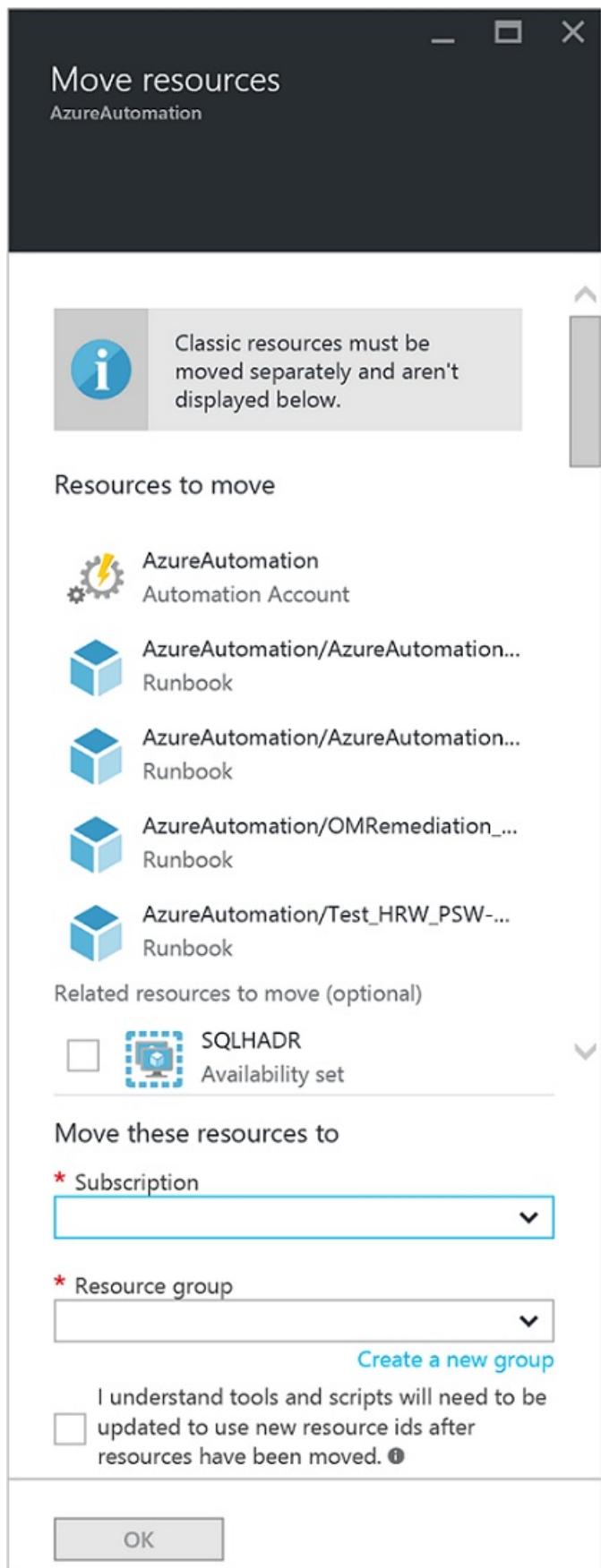
This feature does not support moving Classic automation resources.

To move the Automation Account using the portal

1. From your Automation account, click **Move** at the top of the blade.



2. On the **Move resources** blade, note that it presents resources related to both your Automation account and your resource group(s). Select the **subscription** and **resource group** from the drop-down lists, or select the option **create a new resource group** and enter a new resource group name in the field provided.
3. Review and select the checkbox to acknowledge you *understand tools and scripts will need to be updated to use new resource IDs after resources have been moved* and then click **OK**.



This action will take several minutes to complete. In **Notifications**, you will be presented with a status of each action that takes place - validation, migration, and then finally when it is completed.

To move the Automation Account using PowerShell

To move existing Automation resources to another resource group or subscription, use the **Get-AzureRmResource** cmdlet to get the specific Automation account and then **Move-AzureRmResource** cmdlet to

perform the move.

The first example shows how to move an Automation account to a new resource group.

```
$resource = Get-AzureRmResource -ResourceName "TestAutomationAccount" -ResourceGroupName "ResourceGroup01"
Move-AzureRmResource -ResourceId $resource.ResourceId -DestinationResourceGroupName "NewResourceGroup"
```

After you execute the above code example, you will be prompted to verify you want to perform this action. Once you click **Yes** and allow the script to proceed, you will not receive any notifications while it's performing the migration.

To move to a new subscription, include a value for the *DestinationSubscriptionId* parameter.

```
$resource = Get-AzureRmResource -ResourceName "TestAutomationAccount" -ResourceGroupName "ResourceGroup01"
Move-AzureRmResource -ResourceId $resource.ResourceId -DestinationResourceGroupName "NewResourceGroup" -
DestinationSubscriptionId "SubscriptionId"
```

As with the previous example, you will be prompted to confirm the move.

Next steps

- For more information about moving resources to new resource group or subscription, see [Move resources to new resource group or subscription](#)
- For more information about Role-based Access Control in Azure Automation, refer to [Role-based access control in Azure Automation](#).
- To learn about PowerShell cmdlets for managing your subscription, see [Using Azure PowerShell with Resource Manager](#)
- To learn about portal features for managing your subscription, see [Using the Azure Portal to manage resources](#).

Troubleshooting common issues in Azure Automation

1/24/2017 • 9 min to read • [Edit on GitHub](#)

This article provides help troubleshooting common errors you might experience in Azure Automation and suggests possible solutions to resolve them.

Authentication errors when working with Azure Automation runbooks

Scenario: Sign in to Azure Account failed

Error: You receive the error "Unknown_user_type: Unknown User Type" when working with the Add-AzureAccount or Login-AzureRmAccount cmdlets.

Reason for the error: This error occurs if the credential asset name is not valid or if the username and password that you used to setup the Automation credential asset are not valid.

Troubleshooting tips: In order to determine what's wrong, take the following steps:

1. Make sure that you don't have any special characters, including the @ character in the Automation credential asset name that you are using to connect to Azure.
2. Check that you can use the username and password that are stored in the Azure Automation credential in your local PowerShell ISE editor. You can do this by running the following cmdlets in the PowerShell ISE:

```
$Cred = Get-Credential  
#Using Azure Service Management  
Add-AzureAccount -Credential $Cred  
#Using Azure Resource Manager  
Login-AzureRmAccount -Credential $Cred
```

3. If your authentication fails locally, this means that you haven't set up your Azure Active Directory credentials properly. Refer to [Authenticating to Azure using Azure Active Directory](#) blog post to get the Azure Active Directory account set up correctly.

Scenario: Unable to find the Azure subscription

Error: You receive the error "The subscription named <subscription name> cannot be found" when working with the Select-AzureSubscription or Select-AzureRmSubscription cmdlets.

Reason for the error: This error occurs if the subscription name is not valid or if the Azure Active Directory user who is trying to get the subscription details is not configured as an admin of the subscription.

Troubleshooting tips: In order to determine if you have properly authenticated to Azure and have access to the subscription you are trying to select, take the following steps:

1. Make sure that you run the **Add-AzureAccount** before running the **Select-AzureSubscription** cmdlet.
2. If you still see this error message, modify your code by adding the **Get-AzureSubscription** cmdlet following the **Add-AzureAccount** cmdlet and then execute the code. Now verify if the output of Get-AzureSubscription contains your subscription details.
 - If you don't see any subscription details in the output, this means that the subscription isn't initialized yet.
 - If you do see the subscription details in the output, confirm that you are using the correct subscription name or ID with the **Select-AzureSubscription** cmdlet.

Scenario: Authentication to Azure failed because multi-factor authentication is enabled

Error: You receive the error "Add-AzureAccount: AADSTS50079: Strong authentication enrollment (proof-up) is required" when authenticating to Azure with your Azure username and password.

Reason for the error: If you have multi-factor authentication on your Azure account, you can't use an Azure Active Directory user to authenticate to Azure. Instead, you need to use a certificate or a service principal to authenticate to Azure.

Troubleshooting tips: To use a certificate with the Azure Service Management cmdlets, refer to [creating and adding a certificate to manage Azure services](#). To use a service principal with Azure Resource Manager cmdlets, refer to [creating service principal using Azure portal](#) and [authenticating a service principal with Azure Resource Manager](#).

Common errors when working with runbooks

Scenario: Runbook fails because of deserialized object

Error: Your runbook fails with the error "Cannot bind parameter <ParameterName>. Cannot convert the <ParameterType> value of type Deserialized <ParameterType> to type <ParameterType>".

Reason for the error: If your runbook is a PowerShell Workflow, it stores complex objects in a serialized format in order to persist your runbook state if the workflow is suspended.

Troubleshooting tips:

Any of the following three solutions will fix this problem:

1. If you are piping complex objects from one cmdlet to another, wrap these cmdlets in an InlineScript.
2. Pass the name or value that you need from the complex object instead of passing the entire object.
3. Use a PowerShell runbook instead of a PowerShell Workflow runbook.

Scenario: Runbook job failed because the allocated quota exceeded

Error: Your runbook job fails with the error "The quota for the monthly total job run time has been reached for this subscription".

Reason for the error: This error occurs when the job execution exceeds the 500-minute free quota for your account. This quota applies to all types of job execution tasks such as testing a job, starting a job from the portal, executing a job by using webhooks and scheduling a job to execute by using either the Azure portal or in your datacenter. To learn more about pricing for Automation see [Automation pricing](#).

Troubleshooting tips: If you want to use more than 500 minutes of processing per month you will need to change your subscription from the Free tier to the Basic tier. You can upgrade to the Basic tier by taking the following steps:

1. Sign in to your Azure subscription
2. Select the Automation account you wish to upgrade
3. Click on **Settings > Pricing tier and Usage > Pricing tier**
4. On the **Choose your pricing tier** blade, select **Basic**

Scenario: Cmdlet not recognized when executing a runbook

Error: Your runbook job fails with the error "<cmdlet name> : The term <cmdlet name> is not recognized as the name of a cmdlet, function, script file, or operable program."

Reason for the error: This error is caused when the PowerShell engine cannot find the cmdlet you are using in your runbook. This could be because the module containing the cmdlet is missing from the account, there is a name conflict with a runbook name, or the cmdlet also exists in another module and Automation cannot resolve the name.

Troubleshooting tips: Any of the following solutions will fix the problem:

- Check that you have entered the cmdlet name correctly.
- Make sure the cmdlet exists in your Automation account and that there are no conflicts. To verify if the cmdlet is present, open a runbook in edit mode and search for the cmdlet you want to find in the library or run **Get-Command** <CommandName>. Once you have validated that the cmdlet is available to the account, and that there are no name conflicts with other cmdlets or runbooks, add it to the canvas and ensure that you are using a valid parameter set in your runbook.
- If you do have a name conflict and the cmdlet is available in two different modules, you can resolve this by using the fully qualified name for the cmdlet. For example, you can use **ModuleName\CmdletName**.
- If you are executing the runbook on-premises in a hybrid worker group, then make sure that the module/cmdlet is installed on the machine that hosts the hybrid worker.

Scenario: A long running runbook consistently fails with the exception: "The job cannot continue running because it was repeatedly evicted from the same checkpoint".

Reason for the error: This is by design behavior due to the "Fair Share" monitoring of processes within Azure Automation, which automatically suspends a runbook if it executes longer than 3 hours. However, the error message returned does not provide "what next" options. A runbook can be suspended for a number of reasons. Suspensions happen mostly due to errors. For example, an uncaught exception in a runbook, a network failure, or a crash on the Runbook Worker running the runbook, will all cause the runbook to be suspended and start from its last checkpoint when resumed.

Troubleshooting tips: The documented solution to avoid this issue is to use Checkpoints in a workflow. To learn more refer to [Learning PowerShell Workflows](#). A more thorough explanation of "Fair Share" and Checkpoint can be found in this blog article [Using Checkpoints in Runbooks](#).

Common errors when importing modules

Scenario: Module fails to import or cmdlets can't be executed after importing

Error: A module fails to import or imports successfully, but no cmdlets are extracted.

Reason for the error: Some common reasons that a module might not successfully import to Azure Automation are:

- The structure does not match the structure that Automation needs it to be in.
- The module is dependent on another module that has not been deployed to your Automation account.
- The module is missing its dependencies in the folder.
- The **New-AzureRmAutomationModule** cmdlet is being used to upload the module, and you have not given the full storage path or have not loaded the module by using a publicly accessible URL.

Troubleshooting tips:

Any of the following solutions will fix the problem:

- Make sure that the module follows the following format:
ModuleName.Zip -> ModuleName or Version Number -> (ModuleName.psm1, ModuleName.psd1)
- Open the .psd1 file and see if the module has any dependencies. If it does, upload these modules to the Automation account.
- Make sure that any referenced .dlls are present in the module folder.

Common errors when working with Desired State Configuration (DSC)

Scenario: Node is in failed status with a "Not found" error

Error: The node has a report with **Failed** status and containing the error "The attempt to get the action from server https://<url>/accounts/<account-id>/Nodes(AgentId=<agent-id>)/GetDscAction failed because a valid configuration <guid> cannot be found."

Reason for the error: This error typically occurs when the node is assigned to a configuration name (e.g. ABC) instead of a node configuration name (e.g. ABC.WebServer).

Troubleshooting tips:

- Make sure that you are assigning the node with "node configuration name" and not the "configuration name".
- You can assign a node configuration to a node using Azure portal or with a PowerShell cmdlet.
 - In order to assign a node configuration to a node using Azure portal, open the **DSC Nodes** blade, then select a node and click on **Assign node configuration** button.
 - In order to assign a node configuration to a node using PowerShell cmdlet, use **Set-AzureRmAutomationDscNode** cmdlet

Scenario: No node configurations (MOF files) were produced when a configuration is compiled

Error: Your DSC compilation job suspends with the error: "Compilation completed successfully, but no node configuration .mofs were generated".

Reason for the error: When the expression following the **Node** keyword in the DSC configuration evaluates to \$null then no node configurations will be produced.

Troubleshooting tips:

Any of the following solutions will fix the problem:

- Make sure that the expression next to the **Node** keyword in the configuration definition is not evaluating to \$null.
- If you are passing ConfigurationData when compiling the configuration, make sure that you are passing the expected values that the configuration requires from [ConfigurationData](#).

Scenario: The DSC node report becomes stuck "in progress" state

Error: DSC Agent outputs "No instance found with given property values."

Reason for the error: You have upgraded your WMF version and have corrupted WMI.

Troubleshooting tips: Follow the instructions in the [DSC known issues and limitations](#) to fix the issue.

Scenario: Unable to use a credential in a DSC configuration

Error: Your DSC compilation job was suspended with the error: "System.InvalidOperationException error processing property 'Credential' of type <some resource name> : Converting and storing an encrypted password as plaintext is allowed only if PSDscAllowPlainTextPassword is set to true".

Reason for the error: You have used a credential in a configuration but didn't provide proper **ConfigurationData** to set **PSDscAllowPlainTextPassword** to true for each node configuration.

Troubleshooting tips:

- Make sure to pass in the proper **ConfigurationData** to set **PSDscAllowPlainTextPassword** to true for each node configuration mentioned in the configuration. For more information, refer to [assets in Azure Automation DSC](#).

Next steps

If you have followed the troubleshooting steps above and can't find the answer, you can review the additional support options below.

- Get help from Azure experts. Submit your issue to the [MSDN Azure or Stack Overflow forums..](#)
- File an Azure support incident. Go to the [Azure Support site](#) and click **Get support** under **Technical and billing support**.

- Post a Script Request on [Script Center](#) if you are looking for an Azure Automation runbook solution or an integration module.
- Post feedback or feature requests for Azure Automation on [User Voice](#).

Troubleshooting tips for Hybrid Runbook Worker

1/24/2017 • 2 min to read • [Edit on GitHub](#)

This article provides help troubleshooting errors you might experience with Automation Hybrid Runbook Workers and suggests possible solutions to resolve them.

Hybrid Runbook Worker: A runbook job terminates with a status of Suspended

Your runbook is suspended shortly after attempting to execute it three times. There are conditions which may interrupt the runbook from completing successfully and the related error message does not include any additional information indicating why. This article provides troubleshooting steps for issues related to the Hybrid Runbook Worker runbook execution failures.

If your Azure issue is not addressed in this article, visit the Azure forums on [MSDN and the Stack Overflow](#). You can post your issue on these forums or to [@AzureSupport on Twitter](#). Also, you can file an Azure support request by selecting **Get support** on the [Azure support](#) site.

Symptom

Runbook execution fails and the error returned is, "The job action 'Activate' cannot be run, because the process stopped unexpectedly. The job action was attempted 3 times."

There are several possible causes for the error:

1. The hybrid worker is behind a proxy or firewall
2. The computer the hybrid worker is running on has less than the minimum hardware [requirements](#)
3. The runbooks cannot authenticate with local resources

Cause 1: Hybrid Runbook Worker is behind proxy or firewall

The computer the Hybrid Runbook Worker is running on is behind a firewall or proxy server and outbound network access may not be permitted or configured correctly.

Solution

Verify the computer has outbound access to *.cloudapp.net on ports 443, 9354, and 30000-30199.

Cause 2: Computer has less than minimum hardware requirements

Computers running the Hybrid Runbook Worker should meet the minimum hardware requirements before designating it to host this feature. Otherwise, depending on the resource utilization of other background processes and contention caused by runbooks during execution, the computer will become over utilized and cause runbook job delays or timeouts.

Solution

First confirm the computer designated to run the Hybrid Runbook Worker feature meets the minimum hardware requirements. If it does, monitor CPU and memory utilization to determine any correlation between the performance of Hybrid Runbook Worker processes and Windows. If there is memory or CPU pressure, this may indicate the need to upgrade or add additional processors, or increase memory to address the resource bottleneck and resolve the error. Alternatively, select a different compute resource that can support the minimum requirements and scale when workload demands indicate an increase is necessary.

Cause 3: Runbooks cannot authenticate with local resources

Solution

Check the **Microsoft-SMA** event log for a corresponding event with description *Win32 Process Exited with code*

[4294967295]. The cause of this error is you haven't configured authentication in your runbooks or specified the Run As credentials for the Hybrid worker group. Please review [Runbook permissions](#) to confirm you have correctly configured authentication for your runbooks.

Next steps

For help troubleshooting other issues in Automation, see [Troubleshooting common Azure Automation issues](#)