# Authoring Azure Automation Runbooks in the PowerShell ISE – Tools and Tips

While Azure Automation provides a cool [web-based runbook authoring experience](http://azure.microsoft.com/blog/2014/07/03/azure-automation-in-depth-runbook-authoring/) in the Azure portal, many folks are more comfortable using a good old-fashioned tool they’re already familiar with to write PowerShell – the Windows PowerShell ISE. While in theory it should be very easy to write Azure Automation runbooks in the PowerShell ISE since runbooks are essentially PowerShell workflows, in reality there are a few hiccups one needs to overcome. This blog post will outline those hiccups and explain how you can overcome them to make your PowerShell ISE into a lean, mean, runbook-authoring machine. It also provides information on the new **Azure Automation Authoring Toolkit**, a PowerShell module which, among other things, lets you use the Get-Automation\* activities (which were previously available only in Azure Automation runbooks) in regular PowerShell.

**The Process**

Before we discuss the specifics of authoring runbooks in the PowerShell ISE, let’s talk about the basic process you’ll want to follow when authoring your runbooks outside of Azure Automation’s web-based runbook authoring tool.

For a new runbook, follows these steps:

1. Write a PowerShell Workflow in the ISE.
2. Test the workflow on the local machine.
3. When fully tested, save the workflow as a .ps1 file.
4. Import the .ps1 file into Azure Automation as a runbook.
5. Test the runbook you just imported in the Azure Automation authoring page, using the Test button.
6. When fully tested, publish the runbook in the Azure Automation authoring page using the Publish button.

If you are attempting to modify an existing Azure Automation runbook rather than write a new one from scratch, the steps looks like this:

1. Put the runbook you want to modify into edit mode using the Automation authoring pane.
2. Copy / paste the runbook from the Automation runbook editor to a blank window in the PowerShell ISE to edit is as a PowerShell Workflow
3. Update the PowerShell Workflow as needed in the PowerShell ISE.
4. Test the workflow on the local machine.
5. When fully tested, copy / paste the workflow from the ISE back into the Automation runbook editor, overwriting the old runbook draft with this updated runbook draft.
6. Test the updated runbook you just copied into Azure Automation in the authoring page using the Test button.
7. When fully tested, publish the runbook in the Automation authoring page using the Publish button.

Many of these steps can be done automatically using the Azure Automation Authoring Toolkit (see “6: Moving between local runbook authoring and Azure Automation” section below), but we’ll talk more on that later.

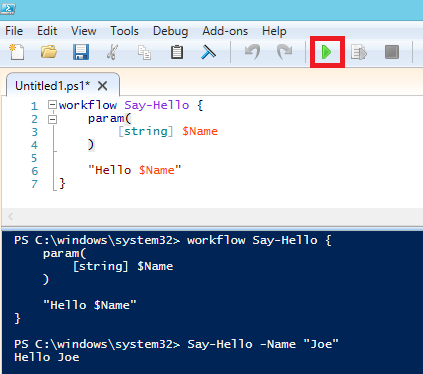
**Making It Work**

There are a few things you will need to do / understand to fully take advantage of the PowerShell ISE as an Azure Automation runbook authoring tool.

1: Starting a runbook

The first thing to be aware of is a tiny detail that people tend to forget which can cause some headaches. This detail is the difference between running a workflow as a runbook in the Azure Automation authoring experience versus running it as a PowerShell workflow in the PowerShell ISE. On the Azure Automation authoring page, running a runbook is as simple as hitting the Test button. This will compile and start the runbook. However, in the PowerShell ISE, there is no equivalent “compile and run” button. You must first compile the workflow by defining its definition in PowerShell, and then run it by typing out the workflow name.

When you want to run the workflow in the ISE, first compile the workflow by running the script. This can be done by hitting the green “play” button, as shown below. Then you can start the workflow by calling the workflow name directly in the PowerShell console attached to the ISE. From below, you can see two PowerShell commands are executed. The first defines the PowerShell workflow, easily done by clicking the “play” button, and the second runs the workflow:



**Note**    *As you continually update and test the workflow in the ISE, remember to click the “play” button before each test to run the script, causing a recompile of the workflow to include your latest changes. If you run the workflow without recompiling it, you will still be running the version of the workflow that was last compiled.*

As a **best practice**, make sure the script containing your workflow contains **only**the workflow, and no commands outside the workflow (comments above the workflow are fine, though). This is because when you import the script into Azure Automation as a runbook, only the workflow for the runbook can be present in the script for import to be successful.

2: Working with child runbooks

Let’s say you want to write a runbook in the ISE that relies on some [child (nested) runbooks](http://azure.microsoft.com/blog/2014/08/12/azure-automation-runbook-input-output-and-nested-runbooks/). These could be new child runbooks you wish to create, also in the ISE, or existing runbooks in Azure Automation that you want to leverage. You may even have child runbooks that call other child runbooks, forming a large dependency hierarchy.

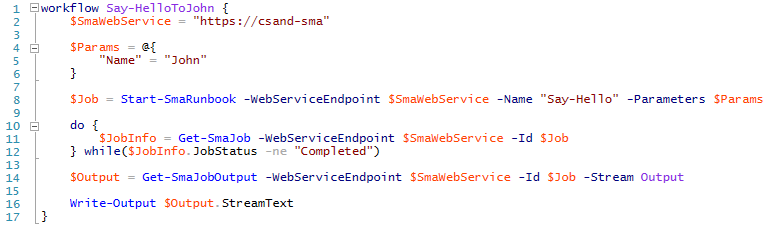
The **best practice** here, if you want to write these runbooks in the ISE, is to start from the innermost child runbook, and work your way outwards until you are at your overall parent runbook, which you would start in order to kick off all child runbooks you wrote, and any child runbooks those child runbooks call, etc. This way, you are writing runbooks in an order where the dependent (child) runbooks of any runbook you write already exist.

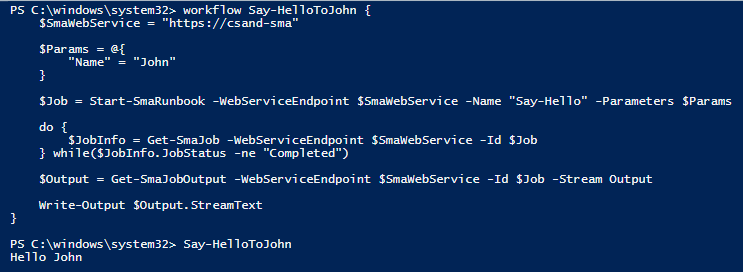
In terms of the details, there’s a number of different scenarios for writing parent / child runbooks in the ISE, depending on whether you are calling the nested runbooks synchronously or asynchronously, and whether the nested runbooks already exist in Azure Automation or not. Below I’ll outline the important aspects of each scenario and how you can handle them.

If you want to start a child runbook **asynchronously** from a runbook you are writing in the ISE, and the child **has been published in Azure Automation** **already,** you should use the [Start-AzureAutomationRunbook](https://msdn.microsoft.com/en-us/library/dn690259.aspx) Azure PowerShell cmdlet to start the child runbook via the Azure API. The [Azure PowerShell module](http://azure.microsoft.com/en-us/documentation/articles/install-configure-powershell/) can be installed locally, allowing you to use the Start-AzureAutomationRunbook cmdlet in your PowerShell workflows in the ISE. Because the Azure PowerShell module ships out of box in Azure Automation, it is already available for use in Azure Automation runbooks, so using this cmdlet in a workflow in the ISE and then moving that workflow into Azure Automation as a runbook will “just work.”

Here’s what this scenario looks like in the ISE and in Azure Automation. The workflow Say-HelloToJohn calls the child workflow Say-Hello. As you can see below, the workflow can work in Azure Automation as a runbook with no changes from the ISE version:

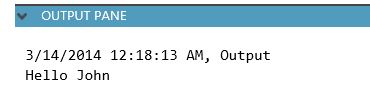
PowerShell ISE:





Azure Automation Authoring Pane:





If you want to start a child runbook **asynchronously** from a runbook you are writing in the ISE, and the child **does not exist in Azure Automation** **already,** the scenario is very similar to the scenario just discussed above. Simply follow the best practice and write the child runbook first in the ISE, then import it into Azure Automation and publish. Then use Start-AzureAutomationRunbook in the parent runbook as discussed above.

If you want to start a child runbook **synchronously** from a runbook you are writing in the ISE, and the child **does not exist in Azure Automation** **already,** simply follow the best practice and write the child runbook first in the ISE. When the child workflow is done, start working on the parent workflow. While you can’t call Azure Automation runbooks synchronously from workflows in the PowerShell ISE, you can call *workflows* synchronously from workflows in the ISE. So, to have a runbook you are writing in the ISE call another runbook synchronously in the ISE, define and compile that child runbook as a workflow in the ISE, and call it inline from the parent. Calling a runbook synchronously from a runbook uses the same syntax as calling a workflow synchronously from a workflow (since runbooks are really just workflows), so this will allow you to test calling one workflow from another synchronously in the ISE, and then import these workflows as runbooks into Azure Automation and call one synchronously from the other without having to make any changes to the runbooks’ definition.

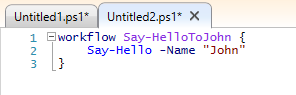
**Note**    *When you import a set of dependent parent / child runbooks into Azure Automation, and you are ready to publish, make sure the first time you publish any parent runbook that you have already published any child runbooks it invokes inline. Otherwise you will have to go back and republish the parent after you have published its children, in order for the parent runbook to execute successfully.*

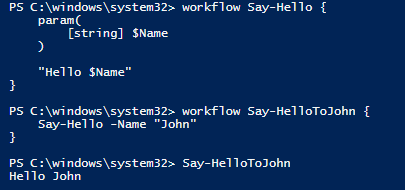
Here’s what this scenario looks like in the ISE and in Azure Automation. The workflow Say-HelloToJohn calls the workflow Say-Hello, shown previously in a screenshot above, as a child workflow, synchronously. Since calling workflows from workflows will only work if the child workflow is already defined and compiled when the parent workflow is compiled, make sure to always define and compile the child workflow before the parent.

**Note**    *If you ever have to make an update to the child workflow and wish to test this updated child workflow as part of the parent workflow, remember to recompile the child workflow, and then the parent workflow, so your latest changes to the child workflow are included when run from the parent workflow.*

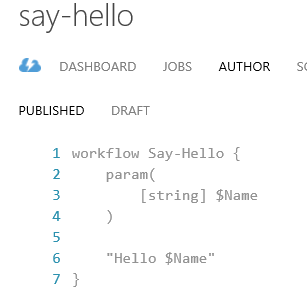
Once moved to Azure Automation, make sure the child runbook is published before attempting to run or publish the parent runbook which calls this child runbook inline. As you can see below the workflows can work in Azure Automation as runbooks with no changes from the ISE versions:

PowerShell ISE:

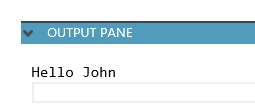




Azure Automation Authoring Pane:







If you want to start a child runbook **synchronously** from a runbook you are writing in the ISE, and the child **exists in Azure Automation** **already,** simply copy the child runbook’s definition to a PowerShell workflow in the ISE, and then follow the previous scenario. If you end up needing to make any changes to the child workflow you copied into the ISE in order to get it to work correctly with the parent workflow you are writing in the ISE, make sure to update the child runbook as well with this new code and publish it in Azure Automation before attempting to run the parent workflow as a runbook in Azure Automation.

Again, many of these steps, such as copying a runbook from Azure Automation to the PS ISE, importing a runbook from the PowerShell ISE into Azure Automation, and publishing a runbook in Azure Automation, can be done automatically using the Azure Automation Authoring Toolkit, but we’ll talk more on that later. [Source control integration](http://azure.microsoft.com/blog/2014/07/24/azure-automation-integrating-runbook-source-control-using-visual-studio-online/) could also be used for syncing runbooks between the PS ISE and Azure Automation, but this post won’t detail that.

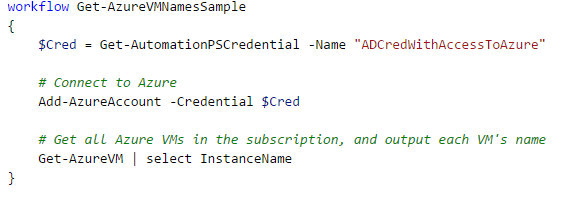
3: Working with modules

While runbooks in Azure Automation can be used to automate any task, most of our users use them to automate Azure operations. In Azure Automation, this is easy because the Azure module ships out of the box in Azure Automation. In your local PowerShell ISE, by default the Azure module is not there, since it does not ship out of the box in Windows. So the first step to talking to Azure from your local PS ISE is to install the [Azure PowerShell module](http://azure.microsoft.com/en-us/documentation/articles/install-configure-powershell/) on the computer who’s PowerShell ISE you will be authoring in. This will add the Azure PowerShell module to a folder in the computer’s [PSModulePath](https://msdn.microsoft.com/en-us/library/dd878326%28v=vs.85%29.aspx) so its cmdlets will be automatically loaded within PowerShell workflows, like in Azure Automation.

Similarly, any other modules you’ve imported into Azure Automation, that you want to call the cmdlets of within your ISE runbooks, will need to be placed in one of the folders in the [PSModulePath](https://msdn.microsoft.com/en-us/library/dd878326%28v=vs.85%29.aspx) on the computer whose PowerShell ISE you’ll be authoring in. Putting a module in one of the folders in the PSModulePath essentially acts like the “Import Module” action in Azure Automation.

4: Authenticating to Azure

As you probably know, in order for your runbooks to automate Azure operations, whether in Azure Automation or in the PowerShell ISE, they need to authenticate to Azure. Our recommended practice for authenticating is via [Azure AD credential-based auth](http://azure.microsoft.com/blog/2014/08/27/azure-automation-authenticating-to-azure-using-azure-active-directory/). This means, within a runbook, passing an Azure AD credential, stored in a PSCredential object, to the Add-AzureAccount cmdlet, to perform the authentication to Azure:

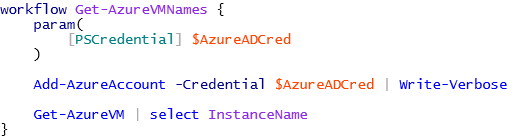


You can, of course, call Add-AzureAccount within a runbook in the ISE, but there isn’t a good way to get the username and password for this credential into a PSCredential object, to pass into Add-AzureAccount:

* You shouldn’t use [ConvertTo-SecureString](http://blogs.technet.com/b/gary/archive/2009/07/23/creating-a-ps-credential-from-a-clear-text-password-in-powershell.aspx), because that requires hardcoding the credential in plain text.
* You shouldn’t use the Azure Automation Authoring Toolkit’s Get-AutomationPSCredential “emulated” activity (discussed more below) because it requires either hardcoding the credential in plain text, or having an Azure Automation job log the credential in plain text.
* You can’t use [Get-Credential](https://technet.microsoft.com/en-us/library/hh849815.aspx) because it requires user interaction to provide the credential, and PowerShell Workflow does not support user interaction.

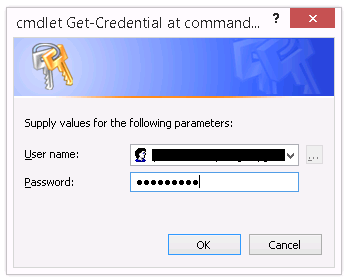
Given this credential has admin access to an Azure subscription, it’s probably not something you want to expose in plaintext! So how can you work around this, and authenticate to Azure without exposing this credential in plain text at any time?

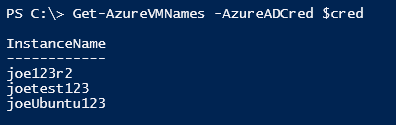
Our recommendation here is to pass the credential by having the runbook accept a parameter of type PSCredential:



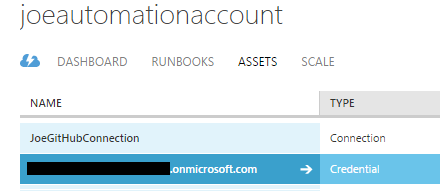
When calling the runbook from the PowerShell ISE, call Get-Credential on the command line to create a PSCredential holding the proper creds, without ever typing the credentials in plaintext. Then pass that PSCredential into the workflow using the parameter you declared:

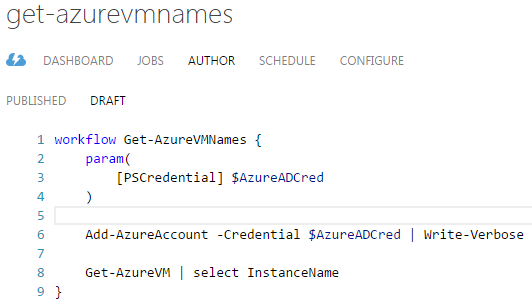


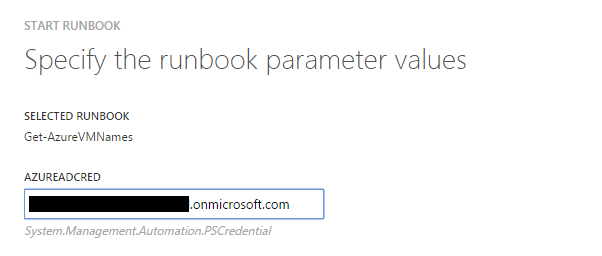


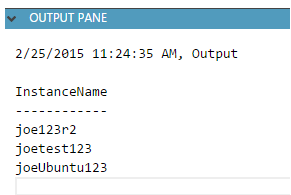


You may be thinking, “OK, that works in the PowerShell ISE, but once I move this runbook into Azure Automation, how do I run it without having to type the credentials for the PSCredential every time?” It turns out, when starting runbooks in Azure Automation that have parameters of type PSCredential, you don’t need to enter the credential’s username/password at all. Instead, Azure Automation asks you to enter the name of an Azure Automation credential asset, and behind the scenes it will fetch that credential asset and pass it as a PSCredential into the runbook:









If using the Azure Automation API/SDK/cmdlets to start a runbook, you would similarly just specify the credential asset name:

*Start-AzureAutomationRunbook –AutomationAccountName “JoeAutomationAccount” –Name “Get-AzureVMNames” –Parameters @{“AzureADCred” = “SomeCredAssetName”}*

More information on this behavior can be found [here](http://azure.microsoft.com/blog/2014/08/12/azure-automation-runbook-input-output-and-nested-runbooks/).

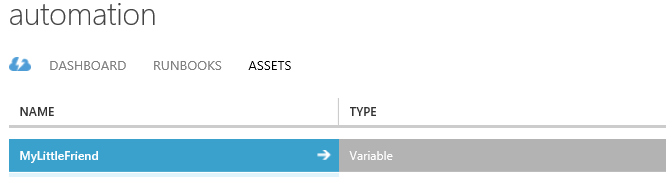
5: Using the Azure Automation-only “Automation” activities

Azure Automation ships with a set of activities useful for interacting with [Azure Automation assets](http://azure.microsoft.com/blog/2014/07/29/getting-started-with-azure-automation-automation-assets-2/) from within runbooks. However, these activities do not exist outside of the Azure Automation runbook execution environment. This means there is no way to take advantage of these activities from the PowerShell ISE. These activities are:

* Get-AutomationVariable
* Get-AutomationConnection
* Get-AutomationPSCredential
* Get-AutomationCertificate
* Set-AutomationVariable

Since Azure Automation runbooks are meant to rely heavily on Automation assets to reference important information that shouldn’t be hard coded within runbooks, having access to these activities during runbook authoring is very important. As you can see below, since these activities don’t exist in the PowerShell ISE, runbook authoring in the ISE can be a challenge if you want to follow the best practice of relying on Automation assets. The workflow shown below calls the workflow Say-Hello, shown previously in a screenshot above, as a child runbook.

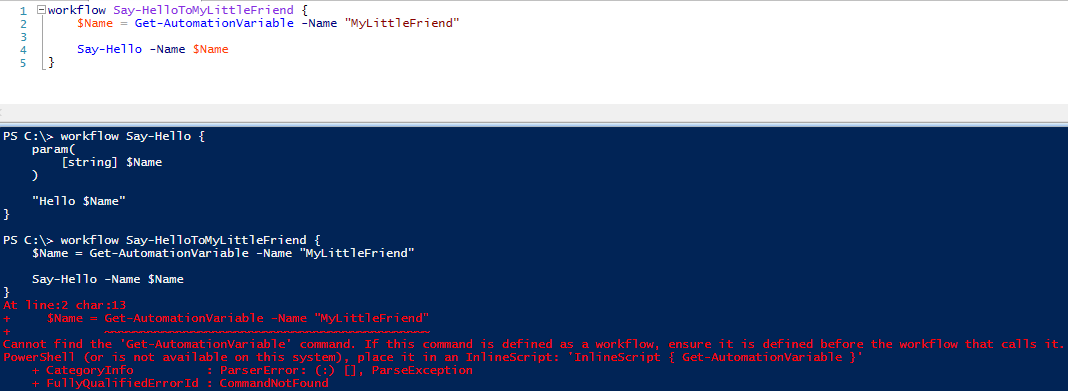
The variable our runbook relies on. It contains the value “Scott”:



The runbook we are writing, working correctly in the Azure Automation authoring experience:



The runbook we are writing, failing to compile in the PowerShell ISE, because the Get-AutomationVariable activity is not available outside of Azure Automation:



So, how can we get around this obstacle and write runbooks that depend on Azure Automation assets from the PowerShell ISE? That’s where the [Azure Automation Authoring Toolkit](http://aka.ms/azureautomationauthoringtoolkit) comes in.

With the Azure Automation Authoring Toolkit, we’ve made it very simple for you to take advantage of Automation assets in your ISE runbooks by providing “emulated” versions of the Automation activities. We’ve written a PowerShell module, “AzureAutomationAuthoringToolkit,” which, among other things, contains an ISE-friendly implementation of all of the Azure Automation-only activities:

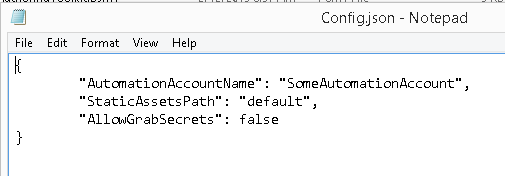
* Get-AutomationVariable
* Get-AutomationConnection
* Get-AutomationPSCredential
* Get-AutomationCertificate
* Set-AutomationVariable

In order to take advantage of the Azure Automation Authoring Toolkit, all you need to do is:

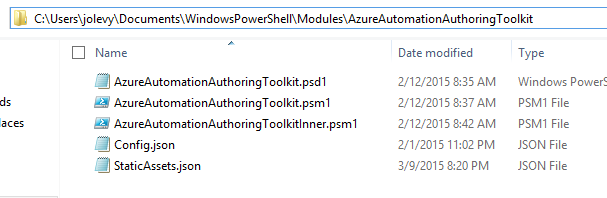
* Install the [Azure PowerShell module](http://azure.microsoft.com/en-us/documentation/articles/install-configure-powershell/) on the host you want to use the Azure Automation Authoring Toolkit from (where you’ll be authoring runbooks in the PS ISE).
* Set up a [connection from the Azure PowerShell module to Azure](http://azure.microsoft.com/en-us/documentation/articles/powershell-install-configure/#Connect) using either Import-AzurePublishSettingsFile, Set-AzureSubscription, or Add-AzureAccount.
* Download the Azure Automation Authoring Toolkit from [here](http://aka.ms/azureautomationauthoringtoolkit).
* Unzip the download. This should present you with 2 items – the Get-AutomationAsset runbook and the AzureAutomationAuthoringToolkit PowerShell module:



* Import the Get-AutomationAsset.ps1 file as a runbook into Azure Automation, and publish it. This is only required if you want to retrieve credentials, certificates, and encrypted variables directly from Azure Automation (more on that below).
* Open up the AzureAutomationAuthoringToolkit PowerShell module’s Config.json file, and configure it based on your desired settings:



* + **AutomationAccountName** represents the Automation Account for the Azure Automation Authoring Toolkit to connect to
  + **StaticAssetsPath** represents the path to the file that will be used to specify any static assets (more on that below). Defaults to “default”, which means the StaticAssets.json file in the AzureAutomationAuthoringToolkit PowerShell module will be used.
  + **AllowGrabSecrets** represents whether the Azure Automation Authoring Toolkit will attempt to grab the values of encrypted assets, not specified in the StaticAssets.json file, directly from the Automation Account the toolkit is configured to talk to. Setting this to true is only required if you want to retrieve credentials, certificates, and encrypted variables directly from Azure Automation (more on that below). Defaults to false.
  + **SecretsCacheTimeInMinutes** represents the amount of time to cache the value of any encrypted asset grabbed directly from the Automation Account the toolkit is configured to talk to. If a cached value is requested before the cache expires, the cached value will be used instead. Setting this value is only required if you want to retrieve credentials, certificates, and encrypted variables directly from Azure Automation (more on that below). If you would rather not cache secret asset values, set this field to 0, and the cache will not be used. Defaults to 10 minutes.
* Put the AzureAutomationAuthoringToolkit PowerShell module folder under C:\Users\<USERNAME>\Documents\WindowsPowerShell\Modules like so:

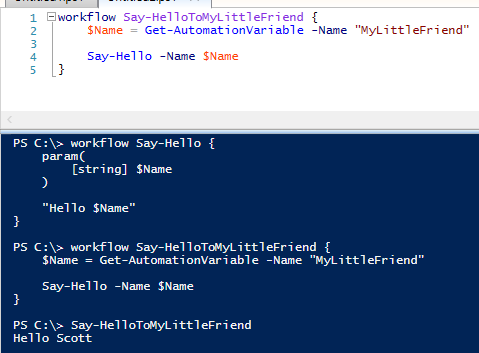


* Tell PowerShell to allow this module to be loaded even though you downloaded it from the Internet:

*Unblock-File C:\Users\<USERNAME>\Documents\WindowsPowerShell\Modules\AzureAutomationAuthoringToolkit\AzureAutomationAuthoringToolkit.psm1*

*Unblock-File C:\Users\<USERNAME>\Documents\WindowsPowerShell\Modules\AzureAutomationAuthoringToolkit\AzureAutomationAuthoringToolkitInner.psm1*

Now just open up the PowerShell ISE, and kick back, relax, and take advantage of Azure Automation-only activities in the runbooks you author. Here’s the same Say-HelloToMyLittleFriend runbook that didn’t work in the ISE before, now working thanks to the Azure Automation Authoring Toolkit!



Now, you may be wondering how the Automation Authoring Toolkit is able to provide asset values locally. The toolkit does this using a combination of manually specified asset values placed in the StaticAssets.json file, dynamically grabbed unencrypted asset values via the Azure Automation cmdlets in the Azure PowerShell module, and dynamically grabbed encrypted asset values via the Get-AutomationAsset runbook in Azure Automation.

Specifically, when a runbook in the PowerShell ISE attempts to grab an asset value via one of the Automation activities, the first thing the toolkit will try is to grab the asset’s value locally from the StaticAssets.json file (or whatever file you specify via the **StaticAssetsPath** setting in Config.json). This static assets file lets you specify the value for any Azure Automation asset, as JSON, per the format below:



As you can see, the assets are divided up by type, and the actual values are provided via key/value pairs. The only exception is certificates, where the thumbprint should be provided, and the toolkit will look for the certificate locally to provide it to the calling runbook. This requires any certificates you want to specify via the static assets file to be imported into the local box where you will be authoring. This file makes it easy to specify the values for any assets that do not actually exist in Azure Automation yet, or that do exist but that you do not want to grab directly from Azure Automation.

If the static assets file does not contain this asset, the next thing the Azure Automation Authoring Toolkit will try is to use the Azure Automation cmdlets in the Azure PowerShell module to grab the asset value from the Automation Account specified via the **AutomationAccountName** setting in Config.json. This behavior makes it very easy to dynamically grab the real value of any asset in Azure Automation – however, for security reasons, the Azure Automation cmdlets are not able to return the values of any secrets, so encrypted variables, connections, credentials, and certificates cannot have their values returned this way. This isn’t an issue for certificates because the thumbprint is returned by the cmdlets, so the toolkit uses this to look for the certificate locally (so the certificate of any certificate asset needs to be installed locally like it does with the static assets file). But for the other secret asset types, either the value of the asset must be provided via the static assets file, or the value can be dynamically grabbed via the Get-AutomationAsset runbook in Azure Automation.

If the static assets file does not contain this asset, and the asset is a secret (encrypted variable, connection, credential, or certificate not installed locally), then if the **AllowGrabSecrets** setting of Config.json is set to true, the Azure Automation Authoring Toolkit will attempt to grab the secret asset’s value from the Automation Account specified via the **AutomationAccountName** setting in Config.json by kicking off the Get-AutomationAsset runbook in that Automation Account. This runbook takes in an asset name and type, and outputs the asset in serialized form. The toolkit then reads in the runbook job’s output and deserializes it back into a value, giving you access to the values of secret Azure Automation assets in your ISE runbooks.

You may be thinking – “but running runbooks in Azure Automation is not instant. Does this mean if I run a runbook in the PS ISE that uses multiple connections / credentials / encrypted variable assets, I will have to wait for the Get-AutomationAsset runbook in Azure Automation to run for each asset? And each time I test the PS ISE runbook I will have to wait again?” Well, while you will have to wait for the Get-AutomationAsset runbook to run at least once for each “secret” asset you are leveraging in a local ISE runbook, the **SecretsCacheTimeInMinutes** setting of the toolkit’s Config.json will cache these asset values locally for a certain amount of time, so that the Get-AutomationAsset runbook is only actually run for an asset if its cached value has expired or does not exist. This means instead of Get-AutomationAsset having to run every time a local runbook requests a “secret” asset, it can be run much less frequently and the cached asset value is used instead.

**Note:** The fact that Get-AutomationAsset outputs the serialized values of assets, combined with the fact that this output is in plain text, means that when this runbook runs it is possible for someone with access to the Azure Automation account to see the values of “secret” assets, whether they be credentials, encrypted variables, or encrypted connection fields, in the output of Get-AutomationAsset’s jobs. In addition, the secrets cache mentioned above uses a plain-text user-level environment variable as the cache, and this environment variable is not automatically cleared (Reset-AzureAutomationAuthoringToolkitSecretsCache can be used to clear it), so anyone logged in as this user or an admin to the local box could see the asset values in this cache, if they know where to look. For these reasons, the **AllowGrabSecrets** setting of the toolkit is set to false by default. If you set **AllowGrabSecrets** to true so that secret asset values can be grabbed from Azure Automation and used in local runbooks, as a **best practice** the Azure Automation Authoring Toolkit should only be pointed at “test” / non-production automation accounts (via **AutomationAccountName**). This is so only “test” assets (those that only provide access to “test” systems or contain “fake” secret information) have the potential to have their values viewed in plain text.

6: Moving between local runbook authoring and Azure Automation

This one isn’t really a best practice in itself, but a way to use the Azure Automation Authoring Toolkit to make the process of exporting runbooks from Azure Automation, working on them locally, and importing / publishing them back into Azure Automation (described in section “The Process” above) easier. In addition to providing “emulated” versions of the Automation activities locally, the Azure Automation Authoring Toolkit also provides multiple cmdlets to make moving between local runbook authoring in the ISE, and use of these runbooks in Azure Automation, easier. These cmdlets are Export-AzureAutomationRunbooksToLocal, Import-LocalRunbooksToAzureAutomation, Export-AzureAutomationAssetsToLocal, and Import-LocalAssetsToAzureAutomation.

* Export-AzureAutomationRunbooksToLocal
  + This cmdlet will export all, or a named set, of runbooks from the Automation Account the Azure Automation Authoring Toolkit is configured to talk to, to a local folder. This is useful for easily grabbing runbooks in Azure Automation so they can be further developed locally.
* Import-LocalRunbooksToAzureAutomation
  + This cmdlet will import all, or a named set, of local runbooks in the specified local folder into the Automation Account the Azure Automation Authoring Toolkit is configured to talk to. This is useful for easily taking runbooks that have been further developed or created locally and putting them into Azure Automation. If the runbooks should be published once placed into Azure Automation, the cmdlet will also make sure all runbooks are published in an order that allows each runbook to successfully call eachother inline.
* Export-AzureAutomationAssetsToLocal
  + This cmdlet will export all, or a named set, of assets from the Automation Account the Azure Automation Authoring Toolkit is configured to talk to a local file that matches the schema of the static assets file (StaticAssets.json). This is useful for easily setting up a static assets file to be used by the Azure Automation Authoring Toolkit that contains the assets in the Automation Account the Azure Automation Authoring Toolkit is configured to talk to. If the –IncludeEncryptedAssetNames parameter is specified, any “secret” assets (credentials, connections, encrypted variables) will be added in the generated static assets file, but will have null as their values, so you must remember to either delete these or fill in their values manually.
* Import-LocalAssetsToAzureAutomation
  + This cmdlet will import all, or a named set, of assets defined in the static assets file the Azure Automation Authoring Toolkit is configured to use to the Automation Account the toolkit is configured to talk to. This is useful for easily taking assets that have been modified or created locally and putting them into Azure Automation. If the –NewAssetsOnly parameter is specified, only new assets in the static assets file will be imported to Azure Automation, assets that already exist in Azure Automation will not be updated.

Of course, [source control integration](http://azure.microsoft.com/blog/2014/07/24/azure-automation-integrating-runbook-source-control-using-visual-studio-online/) could also be used for syncing runbooks and assets between the PS ISE and Azure Automation, but this post won’t detail that.

**Summary**

If you’re one of those folks who loves the PowerShell ISE, you should now have the knowledge and tools you’ll need to successfully write and test your Azure Automation runbooks there – and without sacrificing any runbook functionality!

Until next time,

**Happy automating** (from the PowerShell ISE!)

Just getting started with Azure Automation? Learn about the service [here](http://aka.ms/Q2p1ap), and follow Azure Automation on [Twitter](https://twitter.com/AzureAutomation).

Looking to engage with the Azure Automation engineering team on upcoming features? Fill out our [short survey](https://microsoft.qualtrics.com/SE/?SID=SV_0VAgcjc4xF2CDvn) and we’ll be in touch.