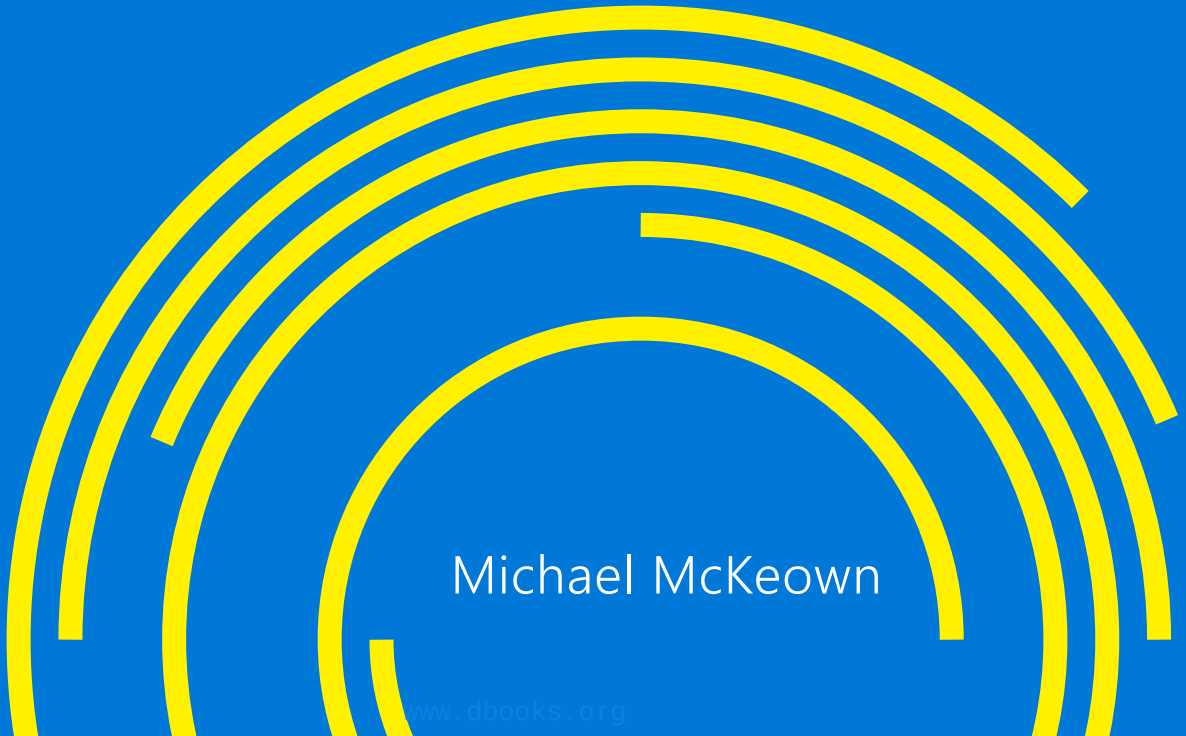
Microsoft

Azure Automation

Microsoft Azure Essentials



Michael McKeown

Visit us today at

microsoftpressstore.com

- **Hundreds of titles available** – Books, eBooks, and online resources from industry experts
- **Free U.S. shipping**
- **eBooks in multiple formats** – Read on your computer, tablet, mobile device, or e-reader
- **Print & eBook Best Value Packs**
- **eBook Deal of the Week** – Save up to 60% on featured titles
- **Newsletter and special offers** – Be the first to hear about new releases, specials, and more
- **Register your book** – Get additional benefits

Microsoft

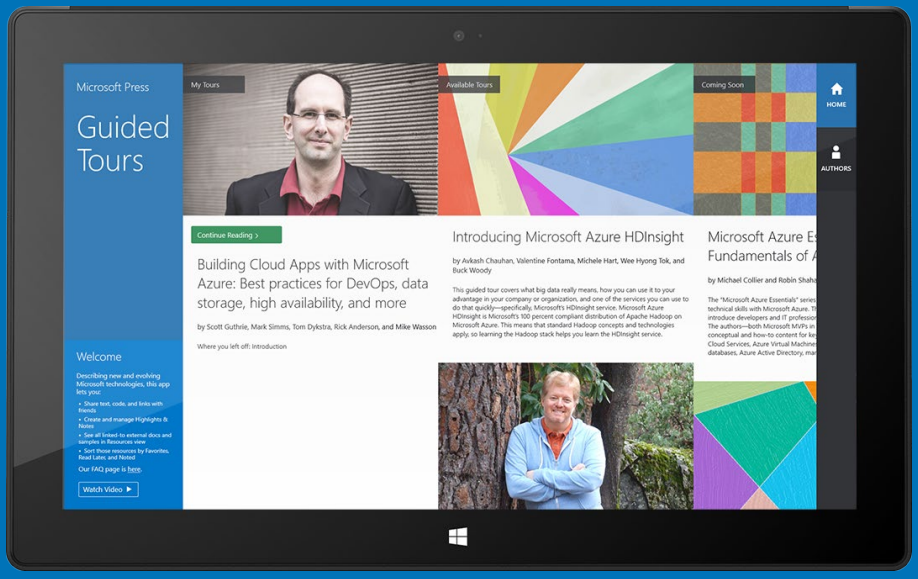
Get the latest news from Microsoft Press sent to your inbox.

- New and upcoming books
- Special offers
- Free eBooks
- How-to articles

Sign up today at MicrosoftPressStore.com/Newsletters

Microsoft

Wait, there's more...



Find more great content and resources in the **Microsoft Press Guided Tours** app.

The Microsoft Press Guided Tours app provides insightful tours by Microsoft Press authors of new and evolving Microsoft technologies.

- Share text, code, illustrations, videos, and links with peers and friends
- Create and manage highlights and notes
- View resources and download code samples
- Tag resources as favorites or to read later
- Watch explanatory videos
- Copy complete code listings and scripts

Download from Windows Store

PUBLISHED BY
Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

ISBN: 978-0-7356-9815-4

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

Acquisitions, Developmental, and Project Editors: Alison Hirsch and Devon Musgrave
Editorial Production: nSight, Inc.
Copyeditor: Teresa Horton
Cover: Twist Creative

Table of Contents

Introduction 7

Who should read this ebook 7

Assumptions 7

Organization of this ebook 7

Conventions and features in this ebook 8

Acknowledgments 9

Errata, updates, & support 9

Free ebooks from Microsoft Press 9

Free training from Microsoft Virtual Academy 9

We want to hear from you 10

Stay in touch 10

Chapter 1 Introduction to Azure Automation 11

Why automation? 11

Repeatable deployment 12

Consistent testing configurations 12

Why Azure Automation? 12

Windows PowerShell workflow 13

End-to-end automation service 13

Off-premises redundancy backed storage 14

Runbook authoring and importing 14

Scenarios 14

Azure Automation pricing 15

Enabling Azure Automation 15

Creating an Azure Automation account 16

Chapter 2 Runbook management 19

What is a runbook? 19
Runbooks support in the Azure Management Portal 19
Import a runbook 20
Import a runbook from the Script Center 20
Import or export a runbook via the Azure Management Portal 21
Create a runbook 22
Create a runbook using Quick Create 22
Create a runbook from the Gallery 23
Author a runbook 26
Runbook parameters 29
Runbook checkpoints 29
Resume or suspend a runbook 32
Chapter 3 Assets 33
Management certificates 33
Azure Active Directory and automation 35
Azure Automation assets 36
Asset scope 37
Variable assets 38
Using a variable asset 40
Integration module assets 43
Importing an integration module asset 43
Integration modules versus runbooks 43
Credential assets 45
Creating a credential asset 46
Connection assets 48
Creating a connection asset 48

Using the Connect-Azure runbook 50
Calling the Connect-Azure runbook using certificates 51
Using Azure Active Directory without the Connect-Azure runbook 53
Schedule assets 54
Creating a schedule asset 54
Using the schedule 55
Chapter 4 Runbook deployment 57
Publishing a runbook 57
Invoking a runbook 58
Invoke from code within another runbook 58
Invoke a child runbook using inline scripts 62
Invoke a child runbook using Start-AzureAutomationRunbook 63
Use Start-ChildRunbook to start an Azure Automation job 64
Invoke a runbook manually from the Azure Management Portal 67
Invoke a runbook using a schedule asset 70
Troubleshooting a runbook 73
Use the Dashboard 73
Enable logging 74
Backing up a runbook 76
Chapter 5 Azure Script Center, library, and community 78
Windows PowerShell workflows and runbooks 78
Azure workflow execution 79
Resources 81
Chapter 6 Best practices in using Azure Automation 83
Runbooks 83
Concurrent editing of runbooks 85
Azure Automation accounts 85

Checkpoints....86
Assets....87
Importing integration modules....88
Credentials and connections....88
Schedules....88
Authoring runbooks....89
Chapter 7 Scenarios....91
Scenario: Provisioning of IaaS resources....92
Provisioning resources....92
Authentication processing....93
Using the New-AzureEnvironmentResourcesFromGallery runbook....94
Creating assets for the runbook....94
Defining parameters and variables....95
Configuring authentication....96
Processing details....97
Scenario: Maintaining and updating Azure IaaS resources....101
Summary of upgrade process....101
Using the Update-AzureVM runbook....102
Supporting runbooks....105
Install-ModuleOnAzureVM runbook....106
Copy-FileFromAzureStorageToAzureVM runbook....107
Copy-ItemToAzureVM runbook....108
Some final thoughts....109
About the Author....110

Foreword

I'm thrilled to be able to share these Microsoft Azure Essentials ebooks with you. The power that Microsoft Azure gives you is thrilling but not unheard of from Microsoft. Many don't realize that Microsoft has been building and managing datacenters for over 25 years. Today, the company's cloud datacenters provide the core infrastructure and foundational technologies for its 200-plus online services, including Bing, MSN, Office 365, Xbox Live, Skype, OneDrive, and, of course, Microsoft Azure. The infrastructure is comprised of many hundreds of thousands of servers, content distribution networks, edge computing nodes, and fiber optic networks. Azure is built and managed by a team of experts working 24x7x365 to support services for millions of customers' businesses and living and working all over the globe.

Today, Azure is available in 141 countries, including China, and supports 10 languages and 19 currencies, all backed by Microsoft's $15 billion investment in global datacenter infrastructure. Azure is continuously investing in the latest infrastructure technologies, with a focus on high reliability, operational excellence, cost-effectiveness, environmental sustainability, and a trustworthy online experience for customers and partners worldwide.

Microsoft Azure brings so many services to your fingertips in a reliable, secure, and environmentally sustainable way. You can do immense things with Azure, such as create a single VM with 32TB of storage driving more than 50,000 IOPS or utilize hundreds of thousands of CPU cores to solve your most difficult computational problems.

Perhaps you need to turn workloads on and off, or perhaps your company is growing fast! Some companies have workloads with unpredictable bursting, while others know when they are about to receive an influx of traffic. You pay only for what you use, and Azure is designed to work with common cloud computing patterns.

From Windows to Linux, SQL to NoSQL, Traffic Management to Virtual Networks, Cloud Services to Web Sites and beyond, we have so much to share with you in the coming months and years.

I hope you enjoy this Microsoft Azure Essentials series from Microsoft Press. The first three ebooks cover fundamentals of Azure, Azure Automation, and Azure Machine Learning. And I hope you enjoy living and working with Microsoft Azure as much as we do.

Scott Guthrie
Executive Vice President
Cloud and Enterprise group, Microsoft Corporation

Introduction

This ebook introduces a fairly new feature of Microsoft Azure called Azure Automation. Using a highly scalable workflow execution environment, Azure Automation allows you to orchestrate frequent deployment and life cycle management tasks using runbooks based on Windows PowerShell Workflow functionality. These runbooks are stored in and backed up by Azure. By automating runbooks, you can greatly minimize the occurrence of errors when carrying out repeated tasks and process automation.

This ebook discusses the creation and authoring of the runbooks along with their deployment and troubleshooting. Microsoft has provided some sample runbooks after which you can pattern your runbooks, copy and modify, or use as-is to help your scripts be more effective and concise. This ebook explores uses of some of those sample runbooks.

Who should read this ebook

This ebook exists to help IT pros and Windows PowerShell developers understand the core concepts around Azure Automation. It's especially useful for IT pros looking for ways to automate their common Azure PaaS and IaaS application duties such as provisioning, deployment, lifecycle management, patching and updating, de-provisioning, maintenance, and monitoring.

Assumptions

You should be somewhat familiar with concepts behind Windows PowerShell programming as well as understand fundamental Azure provisioning and deployment. It helps if you have written and run some Windows PowerShell code, especially as it relates to the Azure PowerShell Management API. This ebook looks at some Azure Automation Windows PowerShell workflow scripts and breaks down what they are doing. If this is your first time with Windows PowerShell, it might be a real challenge for you.

This ebook assumes you have worked in some context with Azure in either the PaaS or IaaS spaces. Items such as Azure assets in the form of connections, credentials, variables, and schedules all will help you manage your Azure applications and deployments. For instance, you should know what is an Azure Virtual Machine (VM) or an Azure Cloud Service.

Organization of this ebook

This ebook includes seven chapters, each of which focuses on an aspect of Azure Automation, as follows:

Introduction to Azure Automation: Provides an overview of Azure Automation, looking at what it

involves, and the situations for which it is best suited. Shows how to enable Azure Automation and how to create an Azure Automation account, which is the highest-level root entity for all your automation objects under that account.

Runbook management: Covers how to manage runbooks, which are logical containers that organize and contain Windows PowerShell workflows. Also, learn about the concept of authentication and the role of management certificates or Azure Active Directory.

Assets: Describes the entities that runbooks can globally leverage across all runbooks in an Azure Automation account. Learn about variable, credential, connection, and schedule assets.

Runbook deployment: Discusses publishing a runbook after it has been authored and tested. Also provides some troubleshooting ideas.

Azure Script Center, library, and community: Learn more about Windows PowerShell Workflow functionality, the execution process, and how it relates to Azure Automation runbooks. Provides an overview of resources for reusable scripts that you can import into your runbooks and use wholly or in part.

Best practices: Looks at some key recommendations to optimize and maximize your use of Azure Automation.

Scenarios: Explores in-depth a few common Azure Automation scenarios that you can hopefully relate to your everyday work.

Conventions and features in this ebook

This ebook presents information using conventions designed to make the information readable and easy to follow:

- To create specific Azure resources, follow the numbered steps listing each action you must take to complete the exercise.
- There are currently two management portals for Azure: the Azure Management Portal at http://manage.windowsazure.com and the Azure Preview Portal at http://portal.azure.com. As of this writing, features related to Azure Automation are available only in the Azure Management Portal.
- Boxed elements with labels such as “Note” or "See Also" provide additional information.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, “Press Alt+Tab” means that you hold down the Alt key while you press Tab.
- A right angle bracket between two or more menu items (e.g., File Browse > Virtual Machines) means that you should select the first menu or menu item, then the next, and so on.

Acknowledgments

I'd like to thank the following people. Jeff Nuckolls, my manager at Aditi, who encouraged me to do this for personal growth. Charles Joy of Microsoft, who helped me get started with Azure Automation and took time to help me work through some tough issues. Joe Levy, who gave me some technical guidance to ensure I was both correct and current. And, my wife and faithful support, Tami, and my kids, Kyle, Brittany, Hap, Mikey, and Wiggy, who put up with me working all the time to get this done. Oh yeah, and so as not to offend any other family support, I might as well thank my Husky, SFD, and my two rabbits, Ting and Chesta.

Errata, updates, & support

We've made every effort to ensure the accuracy of this ebook. You can access updates to this ebook—in the form of a list of submitted errata and their related corrections—at:

http://aka.ms/AzureAuto/errata

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support for this ebook, email Microsoft Press Support at mspinput@microsoft.com.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to http://support.microsoft.com.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

Free training from Microsoft Virtual Academy

The Microsoft Azure training courses from Microsoft Virtual Academy cover key technical topics to help developers gain the knowledge they need to be a success. Learn Microsoft Azure from the true experts. Microsoft Azure training includes courses focused on learning Azure Virtual Machines and virtual

networks. In addition, gain insight into platform as a service (PaaS) implementation for IT Pros, including using PowerShell for automation and management, using Active Directory, migrating from on-premises to cloud infrastructure, and important licensing information.

http://www.microsoftvirtualacademy.com/product-training/microsoft-azure

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this ebook at:

http://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: http://twitter.com/MicrosoftPress

Chapter 1

Introduction to Azure Automation

From an Infrastructure as a Service (IaaS) standpoint, Microsoft Azure Automation is one of the most exciting technologies Microsoft has released. After working with customers in the IaaS space over the past few years, I've seen the need for this type of centralized and high-performance automation functionality.

Azure Automation is a managed service to script and automate application life cycle areas such as deployment, provisioning, and life cycle management. As of this writing, the specific Azure technologies areas supported by Azure Automation include the following:

- Microsoft Azure SQL Database, which is a scaled-down version of full Microsoft SQL Server and provided as a relational database service for platform as a service (PaaS) Azure applications. This service abstracts out the file-system management of a SQL Server service-based solution.
- Microsoft Azure Storage, which encompasses tables, blobs, and files. Table storage service is a nonrelational NOSQL environment to store structured nonrelational data that is optimized for very quick access.
- Microsoft Azure Websites, a simple way to create and deploy websites in Azure by managing the infrastructure, patching, and scalability for you.
- Microsoft Azure Virtual Machines, which allows you to create an IaaS environment that can be attached to an Azure Virtual Network. You manage the non-operating-system installations and all updates from the default platform that is provided for you. Autoscaling and high availability are possible as needed when load increases.
- Microsoft Azure Cloud Services, which is virtual machines (VMs) behind the scenes, but Microsoft manages all the patching to the data and application level on your behalf. Autoscaling and high availability are possible as well.

Why automation?

Although it's not a panacea for all configuration and testing scenarios, automation can truly be an incredible timesaver and increase the consistency of complex dependent deployment and testing scenarios. An enterprise best practice is to look at processes that are frequently repeated and then automate them. Automation minimizes the chance of errors tremendously because the same script is always running every time.

Repeatable deployment

Automation provides the support for repeatable and reproducible results every time a deployment occurs. Deployment can be done rapidly and in a consistent manner each time it's executed, yielding the same results and configurations in less time. The key term here is "same results." Deployment can be one of the most error-prone operations, and it's typically done many times over the lifetime of an application. For example, deployment occurs before a product is released but also to re-create a test environment post-release during regression testing. An application in a test environment is typically deployed repeatedly. Automating the process makes perfect sense to ensure it happens consistently and quickly each time a deployment occurs.

Consistent testing configurations

Variations in testing are a common requirement and can be a significant nuisance at times. For instance, a test configuration might require a scenario with multiple replicated database servers. To set these up manually over a number of runs is a very time-consuming process. As testing progresses and becomes more complex, being able to tweak a base configuration script according to the stage in testing is a significant timesaver. Automating test cases via scripting allows quick configuration and setup of an environment for a test team. Often, there is more than one test team within an organization working in similar environments. Via automation, multiple test teams in an organization can share and modify workflows to fit the requirements of their applications. For example, for testing within an organization, sharing a base automation script across multiple units improves the time taken to plan, design, and deploy multiple test scenarios. In addition, shared automation scripts allow you to more accurately compare results across different applications with teams that might be using the original, or slightly modified, shared testing automation script. This is because when running the same scripts the output is of the same type and can be equally compared with a relative accuracy across different organizational units.

Integrating automation into testing minimizes the time for setup and execution of different and complex test scenarios. For example, consider testing a complex multi-server configuration with SQL Server or Microsoft SharePoint using a replicated Active Directory. As a key part of the testing configuration, automation allows simple setup and teardown of that environment many times in a test cycle. In a test scenario, automation allows you to focus on the processing flow and the value of the data instead of the configuration setup and management.

Why Azure Automation?

Azure Automation is targeted at the repetitive enterprise-level tasks, from simple to complex, that you perform regularly. Any error-prone operation that takes a long time to complete and is going to be done two or more times in its lifetime is a good candidate for Azure Automation. Its overall focus is to provide management of the previously mentioned Azure services.

Let's look at some reasons Azure Automation might interest you from a business and technical standpoint.

Windows PowerShell workflow

You can leverage your current expertise and investment in Windows PowerShell. Azure Automation is based on Windows PowerShell, but it's implemented via Windows PowerShell workflows. A workflow is a group of individual steps that performs a defined task. The workflow follows the model of orchestration set forth by System Center Orchestrator. It gives a flexible orchestration process for workflows and improves reliability across tools, systems, and departments in your organization with a scripting element.

You can integrate Azure Automation into existing systems and existing Windows PowerShell workflow modules to enable integration into other systems via automated repeatability. Azure Automation's engine is the same one used by Service Management Automation (SMA) and is built on the PowerShell Workflow engine. You can take Windows PowerShell workflows you have today and, with a few modifications to adjust to the Windows PowerShell workflow model, run them in the Azure Automation portal. IT operations staff don't need to completely learn a new scripting language.

The Windows PowerShell workflow model increases the reliability of the workflows with their checkpoint model. If for some reason a workflow is interrupted or fails due to a transient error, when it resumes, it does not start at the very beginning of the workflow. Rather, the workflow starts again at the last successful checkpoint in the workflow. This model also provides other improvements in connection pooling and throttling, workflow throttling, and parallel execution of tasks.

End-to-end automation service

With Azure Automation, you can automate end-to-end processes. For instance, if you have a set of Dev/Test VMs, virtual networks, or storage that is no longer needed when testing is done, or you have cloud services that are sitting idle for long periods of time, you can attach metrics and notifications to these processes. Then, you can notify appropriate personnel or release resources when the resources are no longer needed. You can also set up a schedule to automate shutting down resources during certain hours. For a production environment, you might want to manage updates or backups in a way that reduces downtime. Provisioning and updates are easy to manage via automation when you deploy Azure Cloud Services or Azure Virtual Machines and configure the rest of the supporting resources, or enable monitoring for the newly deployed services.

For Azure IT operations personnel, it's a lot of work to spin up an environment and manage it manually from end to end. If deployment is going to be done two or more times, it makes sense to script deployment using Azure Automation. By automating as much work as possible, IT operations staff are free to do other work while the workflows are working in the background in a consistent and repeatable manner.

Off-premises redundancy backed storage

It's often useful to have workflows available to an administrator anywhere in the world. Azure Automation gets workflows outside of on-premises computers and into a safe and highly available central repository where they're available as long as you have an Internet connection. Because they're stored in Azure locally redundant storage, three copies of the workflows within the same datacenter are backed up automatically. Azure Automation gives you an end-to-end solution so you can manage everything (deployment, maintenance, monitoring, and deprovisioning) about your workflows in a centrally and globally accessible location.

Runbook authoring and importing

Think of a runbook as a physical entity to house Windows PowerShell scripts to run within Azure. Azure Automation provides an environment via a built-in browser control to allow you to author and modify runbooks right in the Azure Management Portal (manage.windowsazure.com). You can create a runbook, import a runbook and run it as is, or you can import a runbook and modify it to fit your needs.

Scenarios

Scenarios in which you could use Azure Automation in your cloud environment include the following:

- **Disaster recovery** Deploy quickly new instances of Azure resources within an alternative Azure datacenter after a disaster occurs. Resources might include Azure VMs, virtual networks, or cloud services, along with database servers. This approach would be part of a less expensive "cold" disaster recovery strategy where you don't have a very high recovery time objective (RTO) and don't need to keep an active version of your deployment up and running.
- **High availability** Manage service-level agreements (SLAs) related to high availability to ensure that you have the proper level of availability and personnel are notified so that they can take the appropriate steps when resources fail within a datacenter.
- **Provisioning** Perform initial and subsequent provisioning of a complete deployment, for example, a virtual network, where you assign VMs to it, create cloud services, and join the services to the same virtual network. Anything that you can provision with the Azure Management Portal can be done via Azure Automation.
- **Monitoring** Establish ways to monitor various attributes of your deployment and take appropriate actions when monitored values reach certain threshold limits.
- **Deploying patches** Patch remediation is especially important in the IaaS world because you're responsible for managing the platform and deciding when and how to update the VMs. Azure Automation allows you to develop a runbook to manage the updates at scheduled times to manage patch remediation.

- **Managing VMs** Azure Automation can help manage the life cycle of your VMs. For instance, you might want to provision VMs, or shut down VMs at a specific time each day. You might want an additional way to scale down unused VMs and not rely on the criteria used by autoscaling criteria (CPU or queue length). After a VM is shut down, you might want to delete its accompanying virtual hard drive (VHD) files that back them up, or store them off to Azure Blob storage for later use if needed.
- **Running backups** Azure Automation is great for running regular backups of nondatabase systems, such as backing up Blob storage at certain intervals. Using the credential and connection assets of Azure Automation, you can do backups to external systems or servers.

Azure Automation pricing

Azure Automation provides both a Free and a Standard offering. As of this writing, Azure Automation accounts are located in the Eastern U.S., Southeast Asia, and West Europe regions.

The amount of time your jobs run (CPU time) in the system differs between the offerings. For the Free offering, you have up to 500 minutes of CPU time. For Standard, you have up to 10,000 minutes of CPU time available at $20 per month.

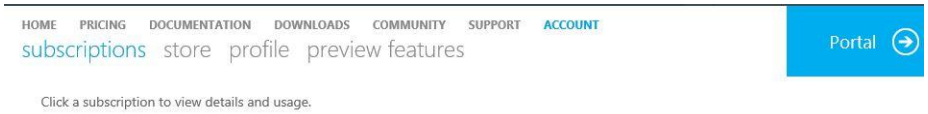
This is a great price for all the features that help you better manage your Azure Automation demands by storing, authoring, editing, running, testing, and deploying your automation workflows in the Azure Management Portal conveniently in one place. For more information, see Automation Pricing.

Enabling Azure Automation

To use the Azure Automation preview feature, you must enable the Automation feature in the Azure portal. To do that, you need an Azure subscription. The easiest way to get one is to sign up for a free trial subscription at free one-month trial. With that subscription you get up to $200 in credits for Azure services that you can use during the trial period. All you need to get a trial subscription is a Microsoft account, a phone number, and a credit card. After that trial subscription, if you decide you want to keep using Azure, you can upgrade and start paying for the services.

To enable Azure Automation, do the following:

1. Go to the Azure Management Portal at *manage.windowsazure.com*, and enter your credentials to log in to your subscription.
2. Click Subscriptions to see a list of your subscriptions.
3. Click Manage Your Account, and then click Preview Features to see all the available Azure features. Click Try It Now for Azure Automation.



4. Select the Azure subscription for which you want to enable Azure Automation. After you complete the registration process, you will see the Azure Automation icon displayed in the left side of the portal.

5. Click Learn More in the Azure Automation section to find information to help you get started. You can find the latest documentation on Azure Automation, run a simple tutorial, jump to the Runbook Authoring Guide, and access the Azure forums to discuss and get answers from others about Azure Automation.

The Learn More page includes a link to the Automation Library. From there, you can access step-by-step directions about how to perform various Azure Automation tasks. For instance, you can learn how to create, schedule, and execute a runbook. The Learn More page provides useful sample runbooks from Microsoft that show how to simplify common tasks and demonstrate useful concepts. In addition, you will find information about how to back up Azure Automation objects before deleting an Azure Automation).

Creating an Azure Automation account

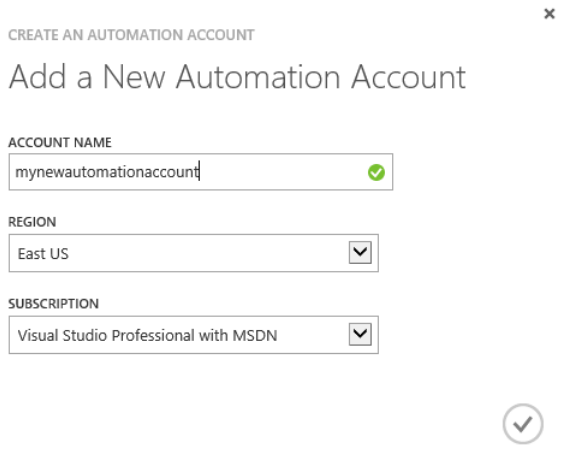
Now that you have an Azure subscription with Azure Automation enabled, the first item of business is creating an Azure Automation account. An Azure Automation account is different from your Microsoft account or Azure subscription. Your Azure subscription contains all your Azure resources, such as Cloud Services, Service Bus, HDInsight, Mobile Services, and so on. An Azure Automation account holds all Azure Automation resources for that account. An Azure Automation resource (such as a runbook or assets) within one account cannot be implicitly shared across other Azure Automation accounts. However, you can view one or more Azure Automation accounts as a logical unit of isolation within an Azure subscription.

You can use an Automation account to organize the automation runbooks specific to a person or a group. Think of an Automation account as a top-level file folder in which you store your runbooks in specific regions. You can have multiple automation accounts per subscription with a maximum of 30 Azure Automation accounts per subscription in different regions, if needed.

For example, an IT operations person might create an Azure Automation account for different groups, such as Marketing, Finance, HR, Development/Test, and Research. These five individual accounts can then hold automation runbooks that are specific to each group's provisioning and lifetime management of resources. Programmatically, the resources in one Azure Automation account don't have scope in another Azure Automation account.

To create an Azure Automation account, do the following:

1. Sign in to the Azure Management Portal at *manage.windowsazure.com*.
2. Click Automation in the left pane to go to the Automation page, and then click Create An Automation Account. The Add A New Automation Account dialog box appears. (Note: If you have only one subscription, the Subscription option is not shown.)



3. In the Account Name text box, enter the name you want to use for the account. In the Region drop-down list box, select the region you want to use for the account. Choose the Azure Subscription that you want the Automation account to apply to, and then click the check mark.
4. The Azure Automation account is displayed on the Automation page as shown in Figure 1-1.

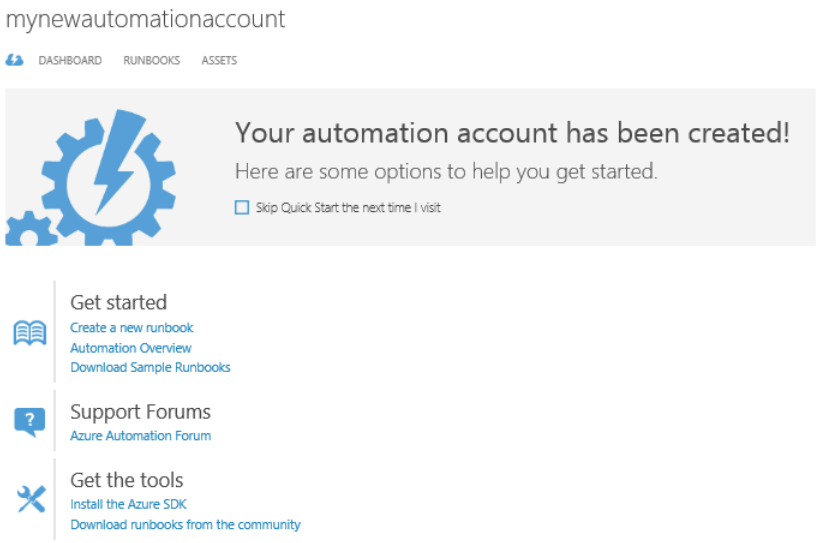


FIGURE 1-1 Newly created Azure Automation account page.

The Azure Automation Account page provides you with the following information:

- **Dashboard tab** Shows diagnostic, job, and usage information for the Automation jobs that have run. It indicates the different status of the jobs (queued, failed, stopped, suspended, completed, and running) with a granularity of one hour to 30 days. You can see the number of runbooks you have, assets (variables, connections, credentials, schedules), and more summary information.
- **Runbooks tab** Provides a list of runbooks and their current view. You can filter the job status by specific dates and times. Other runbooks can be imported here, or you can export one of the runbooks from this Azure Automation account to be used in another account.
- **Assets tab** Allows management of assets global to the mynewautomationacccount runbook. You can modify and create variables, connections, credentials, and schedules. You can also import additional modules that contain Windows PowerShell workflows to use in your runbooks.
- **Scale tab** Allows you to choose the Free or Standard Automation plan. The plan you choose applies to all Automation accounts in the subscription. The Free plan allows 500 minutes of job runtime per month and is not billed. If you need unlimited minutes of job use, choose the billed Standard plan.

Chapter 2

Runbook management

When you distill Azure Automation down to its simplest form, it's the execution of Windows PowerShell Workflow scripts that accomplish tasks related to provisioning, deployment, maintenance, monitoring, and management of your Azure and third-party resources. These scripts are contained in deployment and execution units called *runbooks*. Understanding the development, application, and management of runbooks is critical to being able to effectively use Azure Automation.

What is a runbook?

Let's start by giving a definition of an Azure Automation runbook. If you're familiar with the enterprise software development life cycle (SDLC), the definition of a runbook encompasses a set of processes and procedures that you execute repetitively to support various enterprise tasks. These tasks can include deployment and lifecycle management of resources, deployment, patching, upgrades, backups, error log management, database volume management, user management, and security management.

A runbook allows you to build processes that can be repeated using a Windows PowerShell Workflow script. This technology and methodology (and the script) is now being moved into Azure Automation runbooks, which map to automation of operational tasks. A runbook is one of the core components for Azure Automation that can be used to automate and orchestrate business processes. It is a container for a Windows PowerShell workflow script.

See Also *For more information about Windows PowerShell Workflow and Azure Automation, see Chapter 4, "Runbook Deployment."*

The Windows PowerShell Workflow code used in runbooks typically leverages Azure Automation Assets, which are common and reusable operations and items that can be shared globally across all runbooks. These items include schedules for which the runbooks can run, variables, connections to databases and resources, and authentication entities like certificates and credentials.

Runbooks support in the Azure Management Portal

The Azure Management Portal is your one-stop shop for creating, importing, and managing runbooks. When creating a runbook, you can start it as a draft or you might have a library of runbooks inside your organization that you import from a common location. You can also import runbooks from the Microsoft Script Center. *Creating* a runbook refers to your creation of an empty runbook, for example, in the Azure Management Portal. *Authoring* a runbook refers to the editing and building of the code inside the runbook.

In addition to establishing the runbook via the Azure Management Portal, you can also do all of your runbook management there (including debugging). For example, you can insert Windows PowerShell Workflow scripts into the runbook, and edit and test the scripts. Then, you can debug the script, make sure it works, and see the results, all in the same portal. After you test the script, you can publish the runbooks, invoke them, and manage or view the execution of the job in an integrated manner in the Azure Management Portal.

Import a runbook

Importing a runbook is a very powerful way to quickly add functionality to your script library. When you import a runbook you typically are working with runbook code that has already been written, tested, and made ready to go. You might need to assign your specific values to the variables when you run it, or modify the code slightly after import to meet your specific needs. You can import from a file share on your internal company site where you store your company Windows PowerShell scripts. You can also import runbooks from community sites or the Microsoft Script Center. Many common scripts are available from the Script Center that you can import and use as is or modify for your deployment.

There are many ways to import runbook functionality. Note that when you import a plain Windows PowerShell script, it will be converted to a workflow.

- **Internal site** You can import from a file share on the internal company site where you store your company PowerShell scripts.
- **Script Center** You can import from community sites or the Microsoft Script Center. To import a runbook, you can go to the TechNet Script Center and filter by technologies you might want to consider. Many common scripts are available that you can import directly or copy and modify a bit for your deployment. For more guidance, see "Import a runbook from the Script Center" later in this chapter.
- **Runbook Gallery—Portal** You can import a runbook from the Azure Management Portal Runbook Gallery that contains similar functionality to what you can download from the Script Center for Azure. The best recommendation is to use the Azure Portal Runbook Gallery first, and then go to the online Runbook Gallery if you can't find what you need
- **Runbook Gallery—Online** You can also download and import from the online Runbook Gallery, which continues to expand with many scenarios and sample utility runbooks. Get more information about the Runbook Gallery at Introducing the Azure Automation Runbook Gallery.

Import a runbook from the Script Center

An imported runbook comes in with a draft status, and you have to explicitly decide when to publish it. After you publish it, other runbooks within your subscription can invoke it.

Important Before you import any runbook, make sure that you have done testing in a separate environment to ensure compatibility with the Windows PowerShell Workflow model.

To import a runbook from the Microsoft Script Center, do the following:

1. Go to the Repository of the Script Center.
2. In Categories, select Windows Azure. (As of this writing, the label isn't updated to Microsoft Azure.)
3. Under Operating System, make the selection appropriate to your environment.
4. For Scripting Language, select the PowerShell check box.
5. For Contributors, select the check box for the contribution source you want to use. Generally, scripts originate from either Microsoft or the community.
6. From the results, select a script of interest to you and then download and save it as a file to a location of your choice.
7. Sign in to the Azure Management Portal at *manage.windowsazure.com*.
8. Click Automation, and then click the automation account for which you want to import the runbook.
9. On that automation account's page, click Runbooks and then click Import.
10. On the Import Runbook page, browse to select the runbook file (for example, the file you downloaded in step 6), and then click the check mark to import the runbook. You can import any Windows PowerShell script file with a .ps1 extension. This could be a file you have written previously or a file you download from the Microsoft Script Center. Clicking Import allows you to browse for any PS1 file that is 1 MB size maximum.
11. Find the imported runbook on the Runbooks page of the Azure Automation account. The value in the Authoring column is New. Click the name of the runbook to go to the Learn About This Runbook page.

Import or export a runbook via the Azure Management Portal

To import a runbook, go to the Azure Automation icon on the left side of the portal, which shows any Azure Automation accounts you have for that subscription. Clicking one of those subscriptions will take you to the main account screen. Simply click Runbooks to display all the runbooks contained in that Azure Automation account. At the bottom of the screen are two related buttons, Import and Export.

You can import any Windows PowerShell script file with a .ps1 extension. This could be a script file you have written previously or one you download from the Script Center. Click Import to browse for any PS1 file that is 1 MB size maximum. When you import a PS1 script file, it will be converted to a

workflow during the import operation.

Correspondingly, you can export a runbook by clicking Export. You will be prompted to make sure you want to export it, and you can to save it to any location you want.

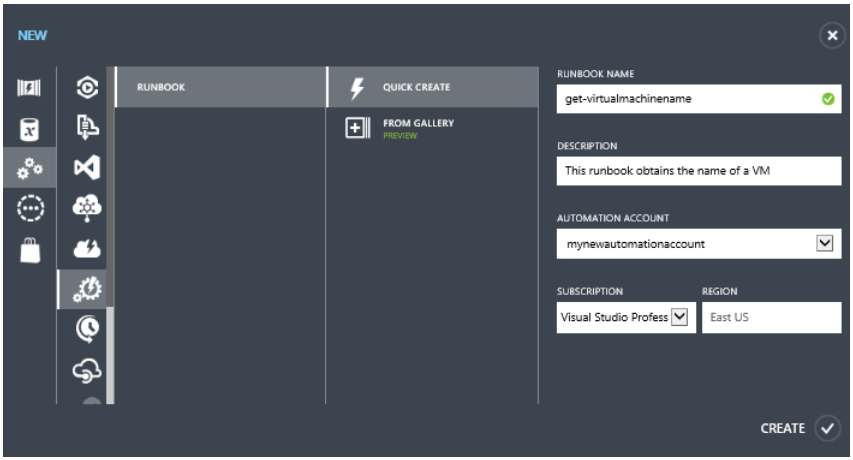
Create a runbook

If you don't want to import a runbook, you can create a runbook in the Azure Management Portal. You have two options: Create a runbook by using Quick Create or create a runbook from the Gallery.

Create a runbook using Quick Create

To create a runbook using Quick Create, do the following:

1. Sign in to the Azure Management Portal at *manage.windowsazure.com*.
2. Click Automation, and then click New.
3. On the New page, App Services and Automation are already selected. Click Runbook, and then select Quick Create.
4. In Quick Create, provide the following information:
    - **Runbook Name** This name can be whatever you want. However, in keeping with the Windows PowerShell format of verb-object, as a best practice it makes sense to name the runbooks accordingly.
    - **Description** If you have many runbooks in your Automation account, the names might become confusing after a while. It's a good idea to enter an informative sentence in the Description field to give you an additional hint about the purpose of the runbook.
    - **Automation Account** Create a new Automation account or select from your existing Automation accounts.
    - **Subscription** Select the subscription for which you want to create the runbook.
    - **Region** The region field autopopulates with the region that corresponds to the subscription selected.
5. Click the Create check mark to create the empty runbook.



When you create a runbook using Quick Create, you get an empty runbook with a workflow template. To accomplish the task, enter or paste Windows PowerShell commands between the curly brackets (Figure 2-1).

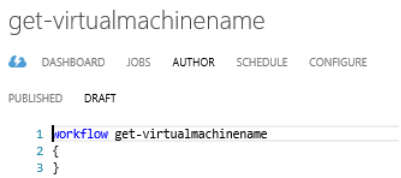
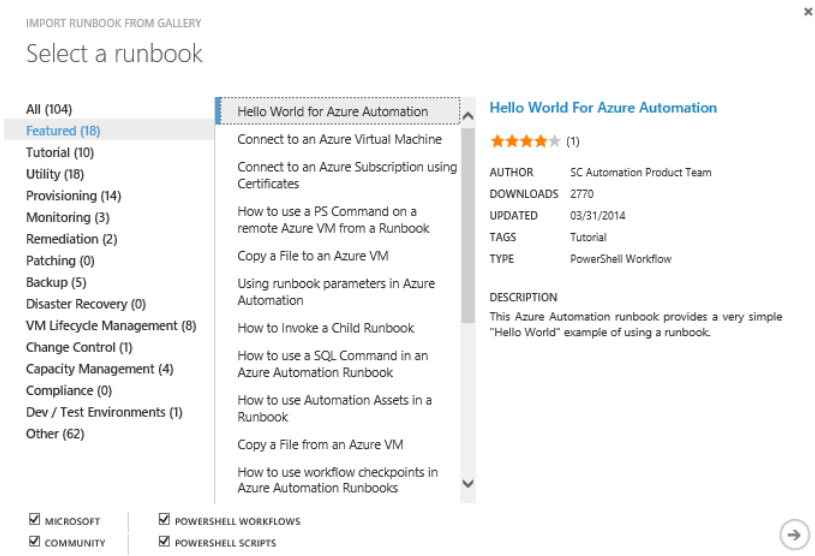


FIGURE 2-1 Workflow template provided by Runbook Creation Wizard.

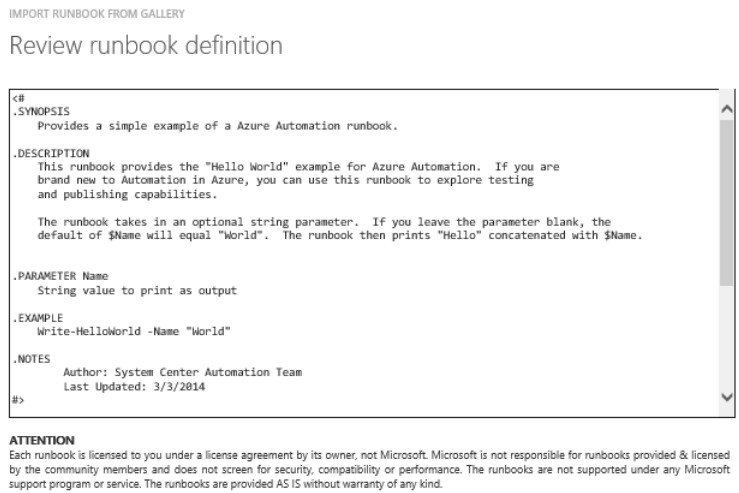
Create a runbook from the Gallery

To create a runbook from the Gallery, do the following:

1. Sign in to the Azure Management Portal at *manage.windowsazure.com*.
2. Click Automation, and then click New.
3. On the New page, App Services and Automation are already selected. Click Runbook, and then click From Gallery to display the Select A Runbook page.
4. On the Select A Runbook page, you can filter to select runbooks authored by Microsoft or the Community and by PowerShell Workflows or PowerShell Scripts. After the filter selections are made, select a category to see the predefined runbooks you can choose for that category.



5. Select the runbook that interests you, and then click the arrow in the lower right of the page. On the Review Runbook Definition page, you can read the information provided about the runbook and then decide if it meets your needs.



6. If the runbook isn't what you want, click the left arrow to go back to select a different runbook. If you do like the runbook, click the right arrow to go to the Enter Runbook Details page. There, you can modify the default name of the runbook. You also select the appropriate Automation account (or create a new account) and the subscription. The region field autopopulates with the region that corresponds to the subscription selected. Click the check mark to create the runbook in your Automation account.

You can now edit the runbook to meet your specific needs.

As shown in Figure 2-2, you can view all your runbooks on the Runbooks tab, filter the time frame, and then view the information in various columns.

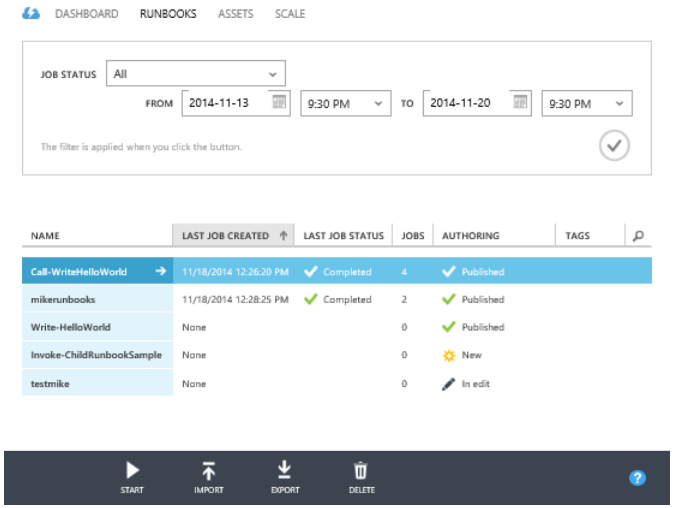


FIGURE 2-2 Existing runbooks for this Azure Automation account.

- **Last Job Created** Shows the last time a job was created. This could contain "None" if a job has not been run within the filtered time frame.
- **Last Job Status** Indicates different job states of Starting, Running, Completed, Queued, Suspended, Completed, or Failed.
- **Jobs** Shows **the** number of published runs of this runbook; does not include draft test runs.
- **Authoring** Indicates the authoring status of the runbook. New means a newly created runbook that has not been published. Published, not surprisingly, means a published runbook. In Edit means a published runbook that is currently also being edited.
- **Tags** Helps you to quickly and easily identify the organize runbooks so that you can easily find them using the list filter. These tags are free-form strings so you can enter whatever values make sense.

When it's in a draft state, a runbook cannot be linked to a schedule asset. It also cannot be invoked by another runbook. Calling runbooks in a nested pattern is a powerful paradigm that you can use to combine existing functionality of runbooks to make up a solution. Any time you import or create a runbook, it is in in a draft state until you explicitly promote it to a published state. You can test the runbook while authoring in draft mode, and then promote it to the published state when it's ready to be used in production. After it's in a published state, the runbook is eligible to be invoked by a schedule asset and called by other runbooks.

Note If you run a runbook in draft mode, it runs just as if it were in a published state with respect to the Azure resources it touches. That is, it will provision, modify, and delete real resources from the script. If your runbook provisions or allocates, or releases and deletes any Azure resources, the resource operations are the same running the runbook in draft mode as running it in a published state.

The in edit state creates an additional draft version of a runbook that can only be run in test mode. The published version also still exists; thus, if you call this runbook from another runbook, even if it's in the in edit state, the previous published version will be run. Also, if you have an already published version of a runbook and then publish a draft, any runbook that is currently in the execution state will continue to use the original version of the runbook under which the job was started. This applies even if the job is in a suspended state. The published version of the runbook will always be called outside of the test experience. The published runbook will run from the schedule, from any other runbooks that call it, or from the command line if you use Start-AzureAutomationRunbook.

In some cases, an issue that could cause a problem is that, after you run a new version of a published runbook (say, V2), any already executing previous versions of that runbook will continue to run with the old version (say, V1). All new instances of that runbook will use the newly published V2 version. Therefore, if a new job is started after the new draft runbook has been published, this new version will be run. It will not affect any of the older versions of the jobs that are running.

Be aware of this issue as you deploy runbook updates to make sure you are running the latest version. In some cases, that difference could be very important. Having multiple versions of the runbook concurrently running might or might not be what you want. With respect to multiple versions, there is no source control per se for versioning different versions of your runbooks. In the History tab of a runbook, however, you can get access to the source code for each of the previous runs of a job.

Author a runbook

To author a runbook, you can develop new Windows PowerShell runbooks on your own and enter your own Windows PowerShell commands. You can call common global runbooks that you have in your asset library. You can then access those runbooks as linked libraries. For instance, you could have a runbook that manages all your credentials and connections that you call from each runbook at the start. A best practice when authoring runbooks is to write granular and single tasks so you can then reuse and insert them later (after they are published) in other runbooks.

All the authoring can be done in the browser in the Azure Management Portal. Log in to the Azure Management Portal, click Automation, and on the Automation page, click the Azure Automation account of interest. On the Automation account's page, click Runbooks, and then click the name of the runbook. On the runbook's page, click Author. At the bottom of the screen, click Edit to allow you to enter or modify its content right there in the Azure Management Portal. This puts the runbook automatically into draft mode for the version you are editing. Note that the previous published version still exists as a separate entity while the editing is taking place.

On the Author tab within Edit mode, you can take the following actions.

- **Manage** From here, you can select the following options:
  - **Import Module** After a module is imported, you can call activities (cmdlets) from that module in your runbook.
  - **Add Setting** This allows you to create any type of asset, or add a type of asset or setting to the runbook code. You insert a setting (or asset) with set or get operations on the assets. These assets have global scope to the entire account for all runbooks in that Automation account.
- **Insert** From here, you can select the following options:
  - **Activity** In this context, an *activity* is a cmdlet. An integration module is a package that contains a Windows PowerShell module; you can import it into Azure Automation. You choose the integration module, and then select the activity, for example, Add-AzureAccount. After selecting the activity, click the arrow to go to the activity's Parameters page where you can make selections from the Parameter Set drop-down list box for required and optional parameters. Select the check mark to close the Insert Activity dialog box and return to the runbook where the template for the activity has been inserted.
  - **Runbook** Select this option to insert an entire runbook from the list of published runbooks in your Azure account. Call the newly inserted runbook just like you would an Azure cmdlet, passing in parameters and getting values back.
  - **Setting** Choose a setting action in which you can get or set a variable, get a connection, get a certificate, and get a Windows PowerShell credential.
- You can create an asset that is a global entity. If the asset is created by one runbook, it can be called by another asset in the same Azure Automation account.

Figure 2-3 shows an example of a runbook the can be edited in Author mode. The Published option is unavailable while the Draft option is active to show that the runbook is in an editing mode.

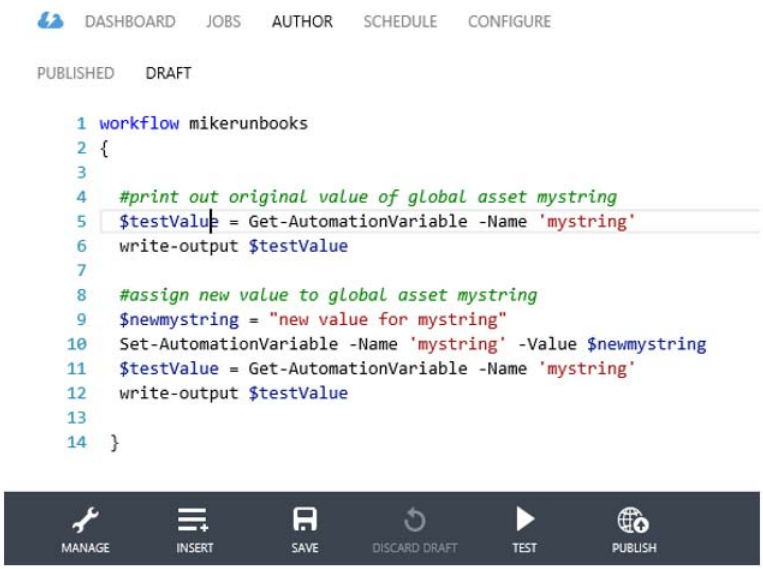


FIGURE 2-3 A runbook in Author mode where code can be written and changed and other operations can be performed from the toolbar.

- **Save** After you have finished entering or inserting the script into your runbook, click Save. If you decide you don't want to keep the modifications, you can leave the page via the browser controls, or by clicking on another runbook or another tab.
- **Discard Draft** This option only applies if the Authoring column of the runbook shows a status of In Edit. As mentioned previously, you can have both a published and a draft version of a runbook. If you want to get rid of the draft version of the runbook, click Discard Draft.
- **Test** Select this option to run the draft workbook, which will modify any Azure resources as if the runbook was running in published mode.
- **Publish** Select this option after you're satisfied that your runbook works as it should. Publish promotes the runbook to a published status. When placed into published status, a runbook is in read-only mode and cannot be edited unless it's transitioned back to draft status.

A common point of confusion about authoring of runbooks concerns simultaneous editing by more than one user from different instances of the portal, for example, a co-administrator scenario. In the Azure Management Portal, you can have more than one administrator, so at times two or more administrators could author a runbook at the same time. However, be aware that Azure does not lock a runbook for editing by a single user. In the Azure Management Portal, a runbook that is being edited is shown as In Edit status. No information is provided about how many people might be editing the runbook.

As a general guideline, if a runbook is in in edit status, no one but the initial person editing the

runbook should edit it until it leaves this state. It's best to wait until the runbook is moved out of in edit status and the draft status has moved into a published state before you try to make changes to it. You can also contact your co-administrator who is editing the runbook and get a copy of the runbook. Then, you can add your changes to their latest edited version.

Runbook parameters

When you author a runbook, you can define parameters that will be passed into the runbook when it's called. This runbook can be called by another runbook in a nested fashion, or it can be invoked by an Azure Automation scheduler asset. You can make the input parameter required by setting Mandatory=$true, or make it optional by setting that value to $false. In the example shown in Figure 2-4, we have set the value to $false (that is, optional), and given the Name parameter a default value of World. This approach allows the logic in the write-output string to work correctly if it's called without a supplied parameter.

```
26 workflow Write-HelloWorld {
27     param (
28
29         # Optional parameter of type string.
30         # If you do not enter anything, the default value of Name
31         # will be World
32         [parameter(Mandatory=$false)]
33         [String]$Name = "World"
34     )
35
36         Write-Output "Hello from new my script - $Name"
37
38 }
```

FIGURE 2-4 Parameterized runbook with optional parameter (Mandatory=$false)

Runbook checkpoints

The scripts used in Azure Automation are built on the Windows PowerShell Workflow model, which provides a powerful feature for checkpoints within the runbooks. By adding a checkpoint to a runbook, you increase its reliability to function despite transient errors, unexpected exceptions, service delays and outages, network downtime, and other issues that are commonly found in a distributed system such as Microsoft Azure for long-running and widely distributed resources. Using checkpoints allows you to confidently automate processes that span multiple networks and systems.

A checkpoint provides a persistence mechanism you can implement at various strategic points in the execution of the Windows PowerShell Workflow. If a problem occurs and the processing of the workflow is interrupted, it can be resumed again near the point of interruption. A checkpoint also ensures that an action will not occur more than once and have a negative duplicate effect. This is the concept of a workflow being idempotent; you can run the workflow more than once, but the result will be the same as if you only ran it once.

Checkpoints are used to persist the state of a running runbook to the Azure Automation database.

Think of a checkpoint as a point-in-time picture that includes any presently generated output, any other implicit, serialized state information, and any existing values of any variables when the checkpoint view was taken. A checkpoint exists in the database until another checkpoint is taken, in which case the first checkpoint is overwritten, or until the runbook completes.

Overhead is associated with the placement of a checkpoint within a runbook. Each time a checkpoint is invoked, a serialization of data persists to storage. If you have a large Windows PowerShell workflow and add a number of checkpoints to it, workflow performance can suffer noticeably. So, although you could place a checkpoint before and after each line in a script file, be smart about your use of checkpoints so performance isn't negatively affected.

Although there are no firm rules on where to put checkpoints, you should plan and strategize their placement within a workflow. If the time it takes to rerun a section of an interrupted workflow is greater than the time it takes to persist the checkpoint, that's probably not a wise use of a checkpoint. Rather, it makes more sense to place a checkpoint after a good chunk of work is done by the workflow itself. This could be defined as making a call to a resource that might or might not be available or ready, calling a routine that takes a very long time to complete its work, or an operation that coordinates multiple distributed resources that are geographically distributed or are highly contended for by a number of processes.

Where you place checkpoints is specific to the workflow and its duties and performance constraints. You don't want to persist a checkpoint when it's not really necessary. Look at activities that might be prone to failure. You also want to avoid having to take the time and resources to do expensive work. Therefore, set checkpoints in the runbook at critical points, and ensure that any runbook restarts do not redo any work that has already completed. Also, you want to encompass any idempotent activities to make sure they don't run more than once when the workflow resumes.

For example, your runbook might create an instance of a Microsoft Azure HDInsight Hadoop cluster with perhaps a hundred or so VMs to handle a big data issue with your script. You could set a checkpoint both before and after the commands to create the cluster. If the runbook fails during cluster creation, when the runbook is restarted, it will repeat the cluster creation work. If the creation succeeds but the runbook later fails, the HDInsight cluster will not be created again when the runbook is resumed.

Azure Automation limits the amount of time a runbook can execute to 30 minutes. Azure will unload a runbook that takes longer than that, assuming that something has gone wrong or the runbook is monopolizing the system. The runbook will eventually be reloaded, and you will want to resume it from where it left off. To ensure that the runbook will eventually complete, you should add checkpoints at intervals under the 30-minute limit.

By using the Checkpoint-Workflow activity within a Windows PowerShell workflow, you tell the system to immediately persist a checkpoint. If an error occurs and the workflow is suspended, the workflow will resume from the point of the latest checkpoint when the job is resumed. Checkpoint-Workflow is a simple call that does not take any parameters and can be placed before or

after any workflow command. However, you can't use the Checkpoint-Workflow activity within an inline block of code.

Let's take a look at some sample workflow code that takes a checkpoint. From within the Azure Gallery, I created a new Azure Automation runbook using the New-AzureEnvironmentResourcesFromGallery runbook that is part of the Microsoft Azure Automation gallery. This is a great example of a runbook that uses a lot of checkpoints. It was written by my good friend Charles Joy, who has a great Azure Automation "Building Cloud" blog at http://aka.ms/BuildingClouds. This script provisions a lot of Azure resources such as an Azure affinity group, adds Azure VMs to that affinity group, and creates a cloud service, among other things. Many of these operations need to occur only once and they need to be successful. For brevity, the entire script isn't included here, just the checkpoint-related sections.

In the following code are calls to Checkpoint-Workflow. After the checkpoint is called, the Connect-Azure Runbook is called to reconnect to Azure using the Automation Connection Asset.

```
# Create/Verify Azure Cloud Service
    if ($AzureAffinityGroup.OperationStatus -eq "Succeeded" -or $AzureAffinityGroup.Name -eq
$AGName) {
        $AzureCloudService = Get-AzureService -ServiceName $CloudServiceName -ErrorAction
SilentlyContinue
        if(!$AzureCloudService) {
            $AzureCloudService = New-AzureService -AffinityGroup $AGName -ServiceName
$CloudServiceName -Description $CloudServiceDesc -Label $CloudServiceLabel
            $VerboseMessage = "{0} for {1} {2} (OperationId: {3})" -f
$AzureCloudService.OperationDescription,$CloudServiceName,$AzureCloudService.OperationStatus,$Az
ureCloudService.OperationId
        } else { $VerboseMessage = "Azure Cloud Serivce {0}: Verified" -f
$AzureCloudService.ServiceName }

        Write-Verbose $VerboseMessage
    } else {
        $ErrorMessage = "Azure Affinity Group Creation Failed OR Could Not Be Verified for: $AGName"
        Write-Error $ErrorMessage -Category ResourceUnavailable
        throw $ErrorMessage
     }

    # Checkpoint after Azure Cloud Service Creation
    Checkpoint-Workflow
    # Call the Connect-Azure Runbook after Checkpoint to reestablish the connection to Azure using
the Automation Connection Asset
    Connect-Azure -AzureConnectionName $AzureConnectionName
    Select-AzureSubscription -SubscriptionName $AzureConnectionName
    # Create/Verify Azure Storage Account
    if ($AzureCloudService.OperationStatus -eq "Succeeded" -or $AzureCloudService.ServiceName -eq
$CloudServiceName) {
        $AzureStorageAccount = Get-AzureStorageAccount -StorageAccountName $StorageAccountName
-ErrorAction SilentlyContinue
        if(!$AzureStorageAccount) {
            $AzureStorageAccount = New-AzureStorageAccount -AffinityGroup $AGName
-StorageAccountName $StorageAccountName -Description $StorageAccountDesc -Label
```

```
$StorageAccountLabel
            $VerboseMessage = "{0} for {1} {2} (OperationId: {3})" -f
$AzureStorageAccount.OperationDescription,$StorageAccountName,$AzureStorageAccount.OperationStat
us,$AzureStorageAccount.OperationId
        } else { $VerboseMessage = "Azure Storage Account {0}: Verified" -f
$AzureStorageAccount.StorageAccountName }
        Write-Verbose $VerboseMessage
    } else {
        $ErrorMessage = "Azure Cloud Service Creation Failed OR Could Not Be Verified for:
$CloudServiceName"
        Write-Error $ErrorMessage -Category ResourceUnavailable
        throw $ErrorMessage
}

    # Checkpoint after Azure Storage Account Creation
    Checkpoint-Workflow
    # Call the Connect-Azure Runbook after Checkpoint to reestablish the connection to Azure using
the Automation Connection Asset
    Connect-Azure -AzureConnectionName $AzureConnectionName
    Select-AzureSubscription -SubscriptionName $AzureConnectionName
```

If you want to have Azure Automation runbooks that withstand being suspended, insert checkpoints carefully. Checkpoints help create runbooks that are able to run for long periods of time to completion and can withstand unexpected failures and maintain reliability.

Resume or suspend a runbook

Resuming or suspending a runbook is closely related to checkpoints. You can manually suspend a workflow, which is typically done if some manual processing needs to be accomplished prior to running another set of activities. You can do this using the Azure Management Portal. A runbook can also be suspended by calling the activity Suspend-AzureAutomationJob, or it can call the Suspend-Workflow activity. This activity will set a checkpoint to cause the workflow to immediately suspend. At the next checkpoint, the job will be subsequently suspended. A possible scenario for this might be where you want to insert a Suspend-Workflow activity if there is a manual step that needs to be taken before a runbook can complete.

Suspension can also occur due to certain conditions. If a runbook unexpectedly crashes, the worker role on which it is running it can be suspended and will resume again from its last checkpoint. As mentioned previously, if an Azure Automation job runs for more than 30 minutes, it will be automatically suspended until given a chance to run again, resuming at the last checkpoint. A job can also unexpectedly raise an exception that causes it to be placed into a suspended state.

Once suspended, jobs can be resumed by calling the Resume-AzureAutomation activity from a Windows PowerShell script. In the Azure Management Portal, you can also manually resume a job. If a worker thread running the runbook crashes, it will find and restart any jobs that need to be completed soon. The resumption on a worker thread will most likely happen on a different worker thread than it was running previously, so don’t make any assumptions about storing any local state on that worker thread.

Chapter 3

Assets

Microsoft Azure Automation assets are global resources used by runbooks to assist in common tasks and value-specific operations. Assets can be imported into modules. Types of assets include connections, credentials, schedules, variables, and integration modules. Global connections and credentials help authenticate between the Windows PowerShell workflows and Azure when the Azure Automation scripts are run against a specific Azure subscription. For instance, Microsoft published the Connect-Azure runbook, which can be used globally within an Azure Automation account to encapsulate the connection and login functionality needed to connect to Azure. Schedule assets can be linked to runbooks, allowing them to run at a specific date and time. Variable assets are used to provide runtime values for runbooks to work on specific subscriptions, as well as to control the logic within the Windows PowerShell code.

Azure Automation is incorporated into Azure Active Directory (Azure AD), which allows simpler management of identity and access for users and groups to the Azure Automation accounts and runbooks. Authentication can now be done with an account within Azure AD instead of having to manage and use management certificates. Using Azure AD greatly simplifies the process of authentication over using management certificates. The account in Azure AD can also be reused and leveraged in other Azure services that support the use of Azure AD.

Management certificates

To run Windows PowerShell Workflow scripts from Azure Automation, you first have to authenticate during the connection using Windows PowerShell credentials or a certificate. You must connect in an authenticated manner to Azure to be able to run any commands against resources within a subscription. Authentication must be set up between Azure Automation and the Azure resources in an Azure subscription that you intend to manipulate via script. You can upload a management certificate to handle this authentication within an Azure subscription.

Azure uses X.509 v3 certificates for authentication in many places. These certificates can be self-signed (usually done for development or testing) or signed by a trusted signature authority (usually done for production). Typically, you upload a .cer file as a management certificate. Certificates used by Azure can contain a private or a public key. A .cer management certificate file does not contain the private key embedded within it, as does a .pfx service certificate (a .pfx file is used to secure client calls to cloud services). Certificates have a thumbprint that provides a means to identify them in an unambiguous way to Azure. For a .cer file, the client connecting the service needs to be trusted and have the private key.

You can share certificates across Azure subscriptions with different subscription owners. This helps you to limit the actual number of certificates you have to create in an enterprise subscription. The limit is 100 certificates per subscription.

A management certificate is not an automation asset per se, although it is global to the subscription in its scope. You upload the management certificate just like any other management certificate in Azure, such as certificates used for Azure Recovery Services, via the Management Certificates tab under Settings.

However, for Azure Automation, the management certificate is also uploaded as an Azure Automation Credential asset if you choose to authenticate using the Certificate Credential asset. This is a key point: To work correctly for Azure Automation, a management certificate has to exist both in the Settings for the subscription and be created as a Certificate automation asset. Why the certificate needs to exist concurrently in two different forms at once at first might seem very confusing.

The Certificate Creation Tool (Makecert.exe) that ships with the Windows SDK provides information about how to create a self-signed management certificate. You can also create one using Internet Information Services (IIS). Alternatively, you can obtain a signed certificate from a verified certificate authority. However, authenticating with a certificate is no longer recommended for Azure Automation.

See Also *For more information about Makecert.exe, see Makecert.exe (Certificate Creation Tool). For more information about using IIS to create a self-signed management certificate, see Create a Self-Signed Server Certificate in IIS 7.*

After you have the management certificate file (.cer) that contains the public key, you must upload it to Azure. Sign into the Azure Management Portal, click Settings, and then click Management Certificates. Click Upload, and then in the Upload A Management Certificate dialog box, browse to the location of your .cer file and select it. As shown in Figure 3-1, select the subscription to which you want to apply the certificate file, and then click the check mark to upload it to the Azure Management Portal.

FIGURE 3-1 Dialog box to upload a management certificate to the Azure Management Portal.

After the upload completes, the certificate is displayed in the list of management certificates, as shown in Figure 3-2. The thumbprint is the public key component of the certificate. It's used with the private key component and verified against any of the loaded certificates for the subscription when Azure Automation is making requests to Azure.

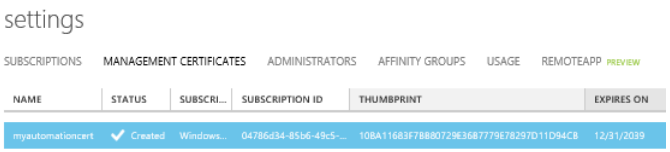


FIGURE 3-2 Settings section of the Azure Management Portal showing uploaded management certificates.

After you have loaded a management certificate into Azure, you're ready to create a certificate.

Azure Active Directory and automation

Authenticating using management certificates is the original and primary way to secure your calls from your Azure Automation scripts into the Azure environment, but there are a lot of steps to create and upload the certificates to Azure. Managing them can also require a lot of organizational effort.

There is now a new and recommended option that provides a more integrated and simpler authentication mechanism for Azure Automation runbooks. Using Azure AD, you can use credential-based authentication for your Azure Automation runbooks. Azure Automation allows a robust and rich, integrated, identity-based authentication mechanism, supporting key industry-wide identity access mechanisms such as single sign-on (SSO) and Multifactor Authentication (MFA). Azure Automation easily integrates and synchronizes with your on-premises enterprise Active Directory installation. Azure Automation also uses role-based access control (RBAC) mechanisms available in the Azure Preview Portal. Additionally, you can leverage RBAC in your Azure Automation runbook authentication strategy. This permits you to simplify and improve control regarding who in your organization is allowed to perform specific operations or access specific resources.

Azure Automation is becoming increasingly integrated into the various Azure services as an all-inclusive identity solution. With Azure Automation, your organizational groups and user accounts are used to simplify secure access to different parts of Azure. When you log into your Azure subscription or use the Azure REST Management application programming interface (API), you authenticate using Azure Automation. Azure Automation, along with services such as Microsoft Office 365, Microsoft Azure SQL Database, Microsoft Azure Mobile Services, and Microsoft Azure Cloud Services, trust Azure Automation with identity access management.

To enable Azure Automation for a new user, do the following:

1. Create the user in Azure AD. For more information about creating a user in Azure AD, see Create or edit users.

2. Add the user as co-administrator to your Azure subscription. Log in to the Azure Management Portal at *manage.windowsazure.com*, click Settings, click Administrators, and then click Add.

3. Log in to the Azure Management Portal as the Azure Automation user you created in step 1 and change the password when prompted.

(This procedure isn't necessary if you want to use an existing Azure user account.) After the user is created, you will want to create an Azure Automation credential asset with the login credentials of that user. As a best practice, it often makes sense to create a user account just to use for running your Azure Automation scripts.

You can access the Azure Automation credential asset from within your Azure Automation runbook. The runbook code gets the credentials from Azure Automation, using the Azure Automation credential asset, and then uses the credentials to authenticate when it connects to Azure.

In the following example, Kim Akers is the credential asset used to authenticate with Azure AD. The Windows PowerShell workflow code makes a call to the Get-AutomationPSCredential cmdlet to authenticate the script:

```
Workflow Get-AzureVMNamesSample
{
      # Grab the credential to use to authenticate to Azure.
      # TODO: Fill in the –Name parameter with the name of the Automation PSCredential asset
      # that has access to your Azure subscription
      $Cred – Get-AutomationPSCredential –Name “KimAkers.onmicrosoft.com”

      # Connect to Azure
      Add-AzureAccount –Credential $Cred

  InlineScript {

      # Select the Azure subscription you want to work against
      # TODO: Fill in the –SubscriptionName parameter with the name of your Azure subscription
Select-AzureSubscription –SubscriptionName “Windows Azure MSDN – Visual Studio Ultimate”

      # Get all Azure VMs in the subscription, and output each VM’s name
Get AzureVM | select InstanceName

   }
}
```

Although using management certificates to authenticate Azure Automation runbooks is still supported, as a best practice, use Azure AD for all your Azure Automation authentication mechanisms whenever possible.

Azure Automation assets

Assets are to Azure Automation as running water is to a modern home. Sure, you can exist without

piped running water by going to the stream or lake near your home (if you have one), manually filling buckets of water, and lugging them home over and over again. But the spillage and time lost in this process makes it not nearly as effective as turning on the faucet to access clear and safe water out of the tap. After you have water, you use it for household tasks such as washing dishes after dinner, running the clothes washer after football practice, bathing the children in the tub, and making lemonade drink mix for snack time.

Assets serve a very similar purpose in Azure Automation as the modern day public water system. Assets are reusable shared global resources that support global and common connections, credentials, variables, and schedules. These can be shared across runbooks in the same Azure Automation account, or between multiple jobs from the same runbook. They can also manage a value from the Azure Management Portal or the Windows PowerShell command line that can be shared across runbooks. Assets promote centralized management of constant values. In the Automation area of the Azure Management Portal, assets are also referred to as settings. You can create variables that can be input by the administrator of the scripts at runtime or set via code. Assets allow a simple standard mechanism for sharing of global entities between jobs, such as variables, schedules, credentials, and connections. By using assets to encapsulate connections and credentials, the login security information is much safer than being hard-coded in workflow code. Schedule assets provide a global scheduling capability.

A good example of using assets is the Connection asset. It allows you to group the connection data necessary to connect an external system into a single object so that it can be easily accessed by runbooks. It provides a template describing how a connection for a certain system should look. This allows users to use this template when defining the connection to this system. Any changes to the connection data can be made in a single place without having to replicate the change in multiple locations (variable assets, runbooks, and so on).

Assets are useful for keeping your configuration values consistent across all runbooks. Using assets simplifies runbook maintenance by storing and maintaining configuration values in a central location. You will most likely want to use assets across multiple runbooks, so allowing updates in one place ensures the changes are reflected everywhere they are used.

Asset scope

The scope of assets is global within an Azure Automation account and shared across all runbooks in that account. For an example, see Figure 3-3. If we have a variable in Asset 1, a credential in Asset 2, and a schedule in Asset 3, with runbooks A and B in the same Azure Automation account AA, either runbook can use Assets 1, 2, and 3. When accessed in code, both runbooks get the same value for all the assets in their respective scripts. If the value is changed in runbook A, the change will be reflected in runbook B the next time it is accessed. However, runbooks in another Automation account (say BB) but in the same subscription will not have scope for any of the assets in Automation account AA.

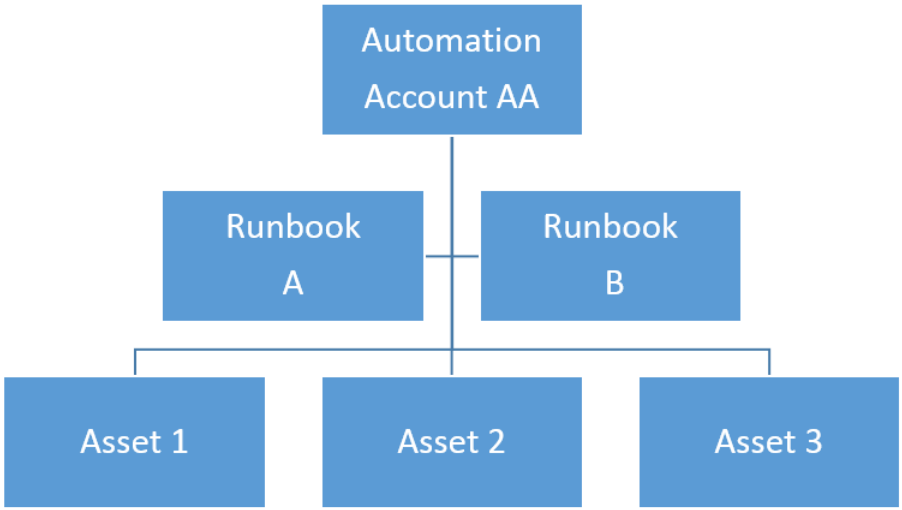


FIGURE 3-3 Runbook scope of assets within an Azure Automation account.

You can view all the assets you have for a particular Azure Automation account. Log in to the Azure Management Portal, click Automation, select the Azure Automation account, and then click Assets. Figure 3-4 shows each of the different types of assets: certificate, connection, credential, module, schedule, and variable.

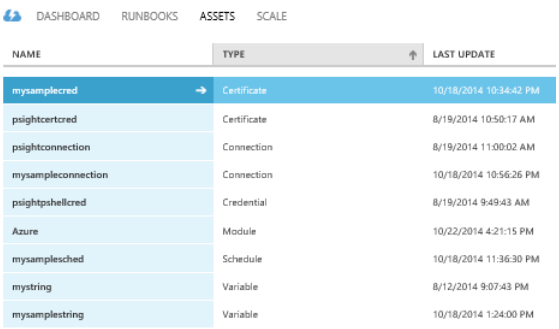


FIGURE 3-4 Automation assets for a specific Azure Automation account.

Variable assets

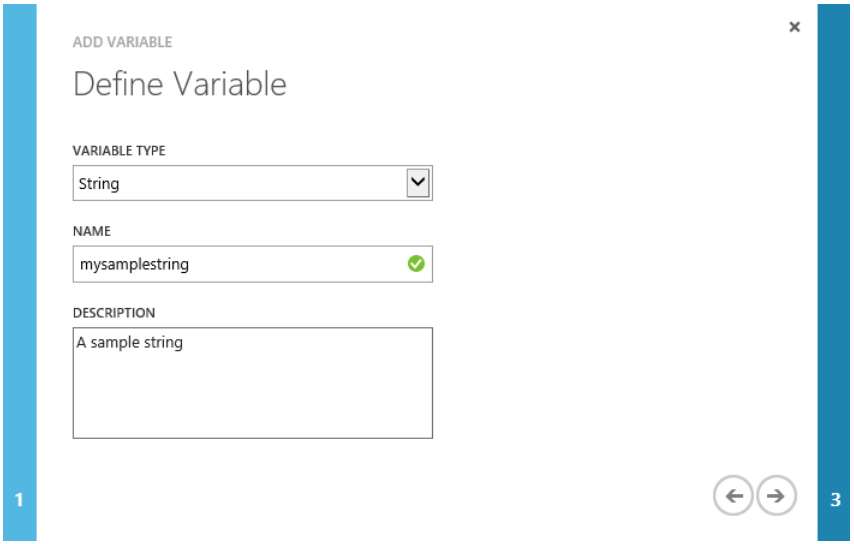
Within Azure Automation, variable assets play an important part in the Windows PowerShell Workflow scripts in the runbooks. A variable is nothing more than a name that represents a value. We can use

variables to reflect changing or current values rather than entering hard-coded data directly into the script code. When the script is run, the variables are replaced with real values. This makes variables quite flexible in that they can hold and reflect data that could be different each time the runbook is run.

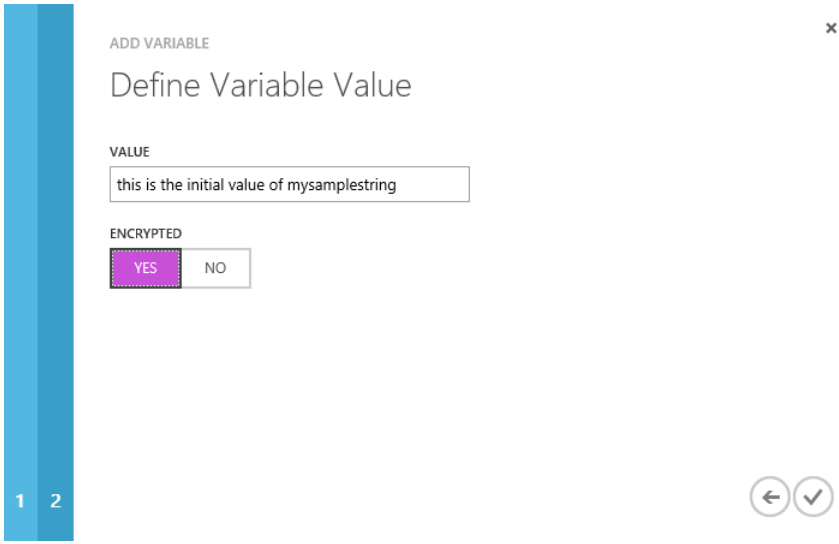
A variable is an asset you define that has global scope (as do all types of assets) across all runbooks in that Azure Automation account. There are four types of variables—string, integer, Boolean, datetime—and a special class named Not Defined that is basically a null value. For all types but Not Defined, you can define an initial value.

To create a variable asset, do the following:

1. Log in to the Azure Management Portal, click Automation, select the Azure Automation account, click Assets, and then click Add Settings.
2. In the Add Settings dialog box, options are available to add a connection, credential, variable, or schedule. Click Add Variable to open the Define Variable dialog box.



3. Click Variable Type and, then select String. In the Name text box, enter a name for the variable. Enter a description (this is optional). Click the right arrow to go to the Define Variable Value dialog box.



4. In the Value text box, enter the initial value for the string. This is optional, as you can leave the variable uninitialized at the start if you choose and later assign it a value at runtime or through script code. After you enter the value, you can choose to encrypt the variable. Select No if you want to see the value of the variable in the Azure Management Portal. Select Yes if you do not want to see the value of the variable in the portal. When a value is encrypted, it's displayed with circle symbols in the portal instead of its actual characters. However, it does not encrypt the data in storage. Only the appearance in the UI is encrypted. Click the check mark to create the variable.

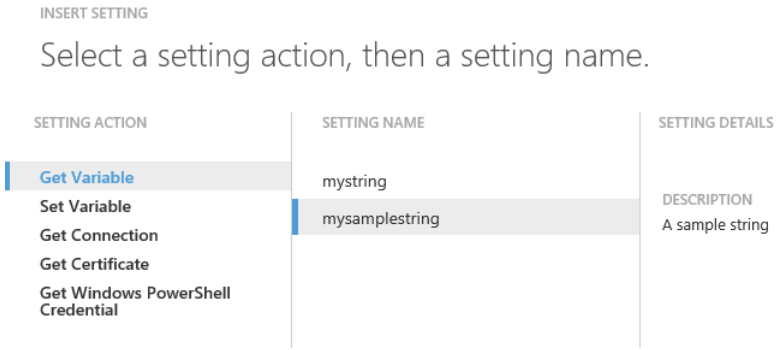
Using a variable asset

To use variable assets in a runbook, you must assign (Set) them a value in code or access (Get) their current value to do something with that value. For example, use the Set-AutomationVariable activity to set the value of a variable asset. Correspondingly use the Get-AutomationVariable activity to get the value of a variable asset. Both of these activities are part of the core Azure module, as are most of the Azure PowerShell activities used in this book.

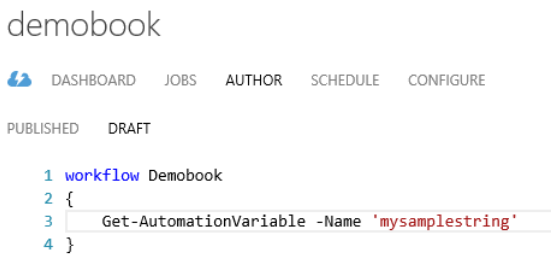
Note For example purposes, this book uses a runbook named Demobook.

1. To set a variable value, log in to the Azure Management Portal, click Automation, click the Automation Account, click Runbooks, and then click the runbook of interest. Click the Author tab to edit that runbook.
2. Move the cursor to the location where you want to make the insertion in the Demobook runbook, and then click Insert > Setting.
3. In the Insert Setting dialog box, select a setting action (Get or Set), and then select a setting

name. This example is about setting a variable value, so select Get Variable. You can use an existing variable and set a value for it, or if the variable does not exist, create one. Choose a setting name for an existing variable. Setting Details shows the current value for that variable.



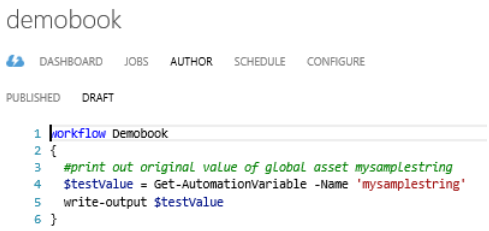
4. Click the check mark to insert the Get-AutomationVariable activity into the runbook code at the location of the cursor. By default, if you don't move the cursor, the activity is inserted in the first column in the first row. If you insert a setting at that location, it will remove the name of your runbook.



To insert the Set-AutomationVariable activity, use the same process except choose that activity from the Setting Action column in the Insert Setting dialog box.

You can use the Get-AutomationVariable and Set-AutomationVariable activities together to understand the concept of global scope for assets. The following process uses the mysamplestring variable asset and the Demobook runbook shown previously. In addition, a second runbook example named Demobook2 shows the global scope across runbooks of a variable asset.

1. Create a temporary variable $testValue, and then assign it the value of mysamplestring. Make a call to write-output to display the original value of $testvalue. Click Test to run and show this output.

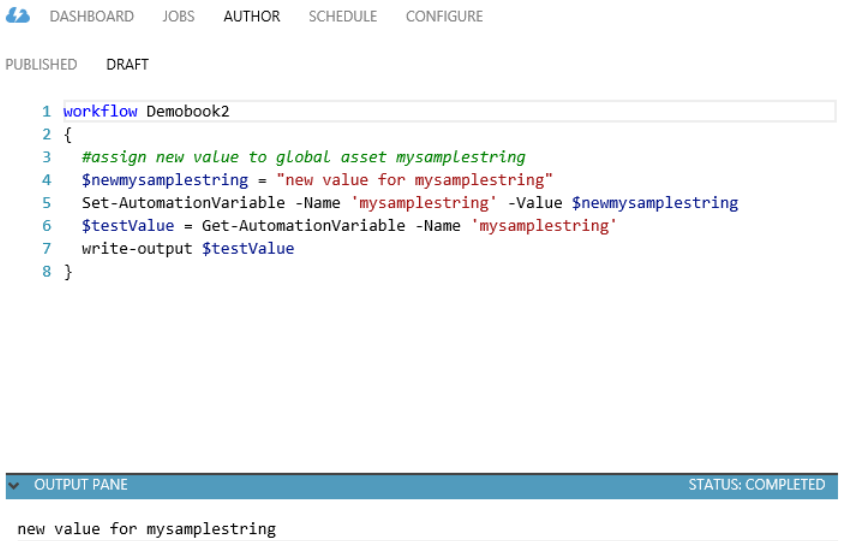


2. Create a new runbook called Demobook2.

3. Assign a new value to mysamplestring of "new value for mysamplestring". Click Insert >Setting. In the Insert Setting dialog box, under Setting Action, select Set Variable. In Setting Name select the setting name, and then click the check mark. This action results in the following being written to the Demobook runbook at the current cursor location:

```
Set-AutomationVariable -Name 'mysamplestring' -Value <System.Object>
```

4. Replace the <System.Object> with $newmysamplestring. This sets the value of mysamplestring to the value contained in $newmysamplestring. Call Get-AutomationVariable to obtain the value of mysamplestring into the $testvalue variable, which has just changed. Call write-output to display the value of $testvalue. Click Test to run the code and see the output of both the original global value of mysamplestring of "mysamplestring" and the updated global value of "new value for mysamplestring".



This example demonstrates that all runbooks in an Azure Automation account share the same global value for mysamplestring. If one runbook changes its value, the change is reflected across all runbooks in that automation account. Also note that if you have a variable asset in another of your automation accounts by the same name—mysamplestring, in this case—it will be a completely

different value and in a totally different storage location than the mysamplestring variable in your other runbook.

This principle applies not just to variable assets, but to other assets you can insert into code, including connections, certificates, and Windows PowerShell credentials. Schedule assets are a bit different from the other types of assets in that you don't call them in scripts. However, their capability is still global to all runbooks in an Azure Automation account.

Integration module assets

Integration modules are published Windows PowerShell libraries that can be imported into Azure Automation as a module asset and used when authoring runbooks. They can be up to 30 MB in size and must be in zipped format. By default, when you create an Automation Account, the Azure PowerShell module is imported. This module asset supplies all the Azure PowerShell cmdlets (also referred to as activities) that you can use in your runbooks. You can see the Azure module by itself initially when an Azure Automation account is created. Additionally, you can import additional Windows PowerShell Workflow modules as assets.

Importing an integration module asset

To import a module asset, do the following:

1. Download the module asset as a .zip file and then save it locally.
2. In the Azure Management Portal, click Automation, select the Azure Automation account, and then click Assets.
3. Click Import Module to browse and select the module to be imported, and then click the check mark to begin the import process. The display shows it is unzipping the activities in that module. After the module has completed the import process, it is displayed at the Azure Automation Account level under Assets as a Module asset type.

The most common issue encountered during importing a module is that the zipped module package must contain a single folder within the .zip file that has the same name as the .zip file. Within this folder, there needs to be at least one file with the same name as the folder, and using the extension .psd1, .psm1, or .dll. Also, the Integration Module package is a compressed file with the same name as the module and a .zip extension. It contains a single folder also with the name of the module. The Windows PowerShell module and any supporting files, including a manifest file (.psd1) if the module has one, must be contained in this folder.

Integration modules versus runbooks

A common misunderstanding in Azure Automation is the concept of a module versus a runbook, as well as the terms *import* and *insert*.

An Azure Automation *runbook* is an execution unit that contains Windows PowerShell Workflow

scripts. Scripts are a program, a group, or many Windows PowerShell commands in one file. Runbooks contain scripts. A runbook that is less than 1 MB in size and not currently part of an Azure Automation account can be imported in PS1 format into that Azure Automation account. When a runbook is invoked, it is sent to a virtual runtime environment to run, which occurs transparently behind the scenes. Runbooks are invoked by a schedule, called by other runbooks (when the runbook has been published) when they are inserted during authoring, or run manually by themselves.

A *module* is a group of activities (cmdlets) that you can *insert* into a runbook after the module has been imported. You can *import* a module into an Azure Automation account via controls in the Assets area of the Azure Management Portal runbook UI. All runbooks in an account can then call any activities of that runbook. (Remember, an activity is a cmdlet.) The module must be in zipped format, up to a 30 MB limit. By default, Azure activities are imported as assets for use in all your runbooks. In the Azure Management Portal, when you look at Assets under any new Azure Automation account, the runbook that contains the Azure activities is named Azure.

When in Author mode for a runbook, you can choose Insert and then select Activity to display the dialog box shown in Figure 3-5. In the Insert Activity dialog box, you can choose the integration module and then select an activity to see its description.

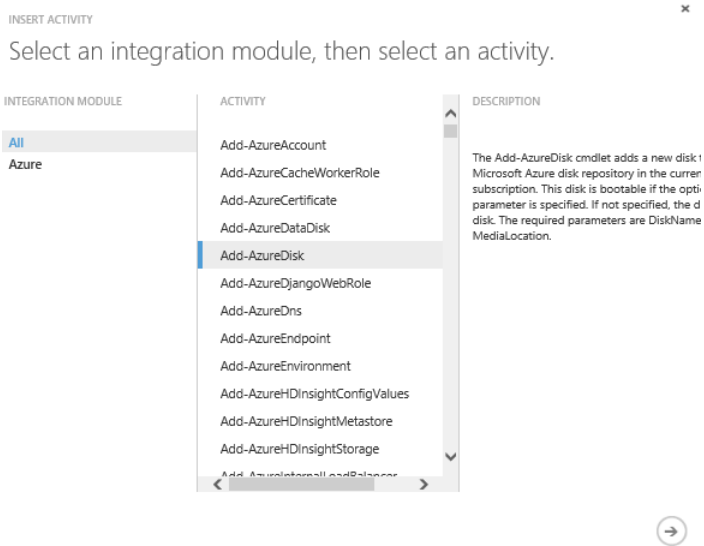


FIGURE 3-5 Adding the AddAzure-Disk activity to the code in the current runbook.

Figure 3-6 shows the list of parameters for the Add-AzureDisk activity. The DiskName and MediaLocation parameters are required when you make the call to Add-AzureDisk. The Label and OS parameters are not required.

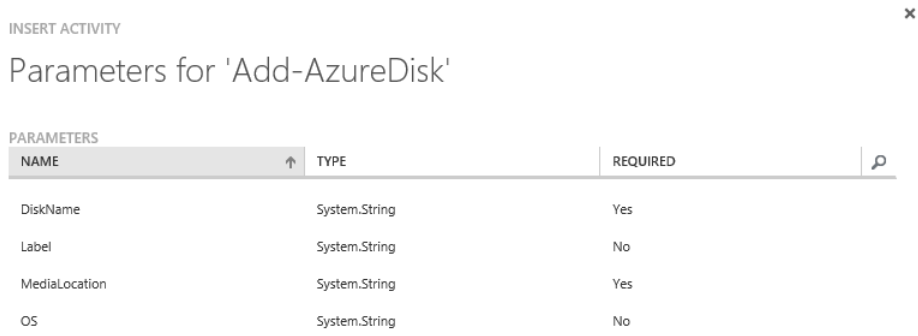


FIGURE 3-6 The parameters for the Add-AzureDisk activity.

When you click the check mark to close this dialog box, a template for Add-AzureDisk is added to the cursor location for the runbook. In the following code example, note the line continuation character ` at the end of each line as it is inserted. Azure uses this method to insert each activity into a script. If you prefer, you can remove these characters and put the command all on one line.

```
Add-AzureDisk `

-DiskName <System.String> `

    -MediaLocation <System.String> `

    [-Label <System.String>] `

    [-OS <System.String>]
```

Due to the lack of brackets [ ] around them, the first two parameters, DiskName and MediaLocation, are required when using this activity. The other two parameters in [ ] square brackets, Label and OS, are optional. You would replace the <System.String> entities with actual string values or temporary variables. For instance, the call within your runbook at runtime might look something like the following example:

```
Add-AzureDisk -DiskName "MyDiskName" -MediaLocation
"http://mystorageaccount.blob.core.azure.com/vhds/winserver-system.vhd" -Label "BootDisk" -OS
"Windows"
```

Credential assets

The credential asset is used to gain secure access to external systems, networks, databases, services, and so on. You can view it as a "Run As" security principal that gives an identity to the call into that external system. This asset is used most commonly in IaaS deployment situations such as authenticating when accessing a SharePoint or a SQL Server IaaS VM. Credential properties are stored in Azure Automation assets and are referenced inside of script workflows using either the Get-AutomationCertificate or the Get-AutomationPSCredential activity.

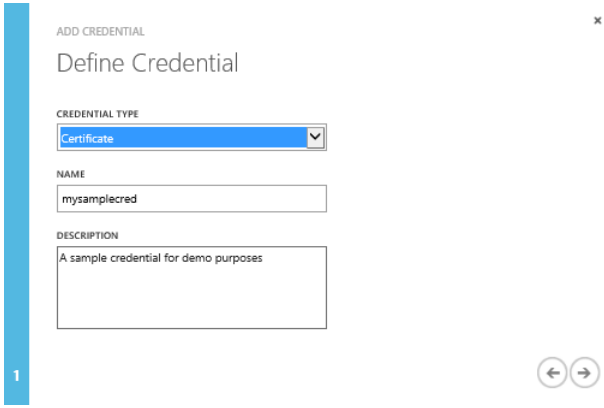
When using credential assets, you can authenticate using either a certificate credential or a

Windows PowerShell credential. Certificate credentials are based on a management certificate. It's a best practice to use Azure AD for the certificate. The connection asset uses the management certificate to authenticate to that subscription. A Windows PowerShell credential uses the script when it connects to the VM and needs to provide a username and a password. Typically this identity is used to log into an Azure VM.

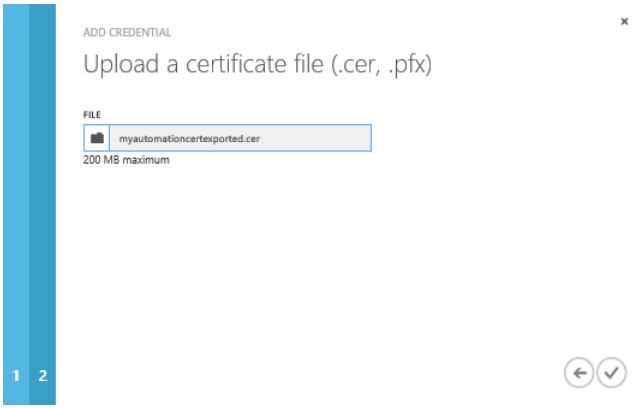
Creating a credential asset

To create a credential asset, do the following:

1. In the Azure Management Portal, click Automation, select the Azure Automation account, and then click Assets. Click Add Setting.
2. In the Add Setting dialog box, click Add Credential.
3. In the Add Credential dialog box, select the Credential Type of the setting you want to add. In the following screenshot, the Certificate credential type is selected. The other credential type option is Windows PowerShell Credential, where the user will need to provide the userid and password credentials. In addition, provide a name and brief description in the Name and Description text boxes. After you have provided the information, click the right arrow.



4. On the Upload A Certificate File page, browse for a certificate file, which can be a .cer or .pfx file.



After you load the certificate and create the credential, you can go to the Assets tab, find your new credential, open it, and see the certificate details, as shown in Figure 3-7. Note the value of the thumbprint.

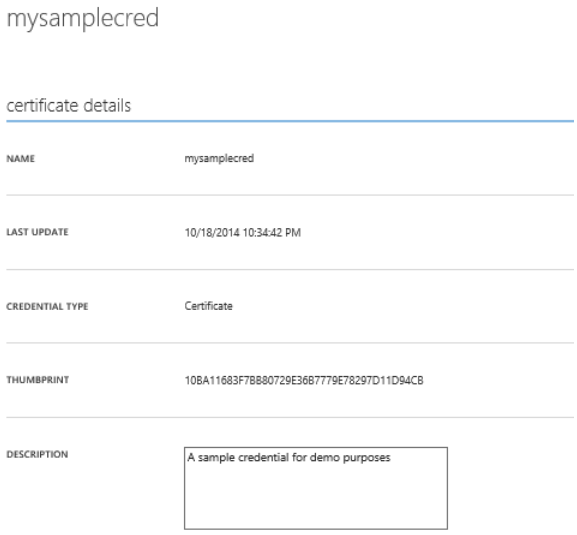


FIGURE 3-7 Details of a certificate that has been installed in Azure to support the credential asset.

If you go into the management certificate section of the Azure Management Portal and find the certificate you just loaded up for this credential, you will see that same thumbprint value. When Azure Automation tries to authenticate, it will use this credential to access the thumbprint and pass it to Azure. Azure will attempt to match the thumbprint of the credential to that of the corresponding certificate to authenticate the call.

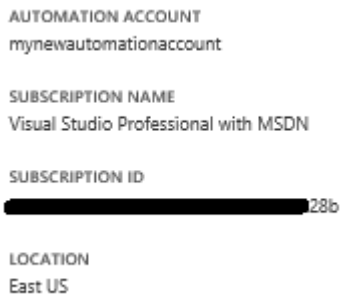
Connection assets

Connections are assets that contain information to connect to external networks or systems. For these external connections, a method must present all the data necessary for connecting to external systems. This could mean ports, protocols, usernames, and passwords. Potential complications are that different systems require different types of data to be passed from the runbooks. Assets allow a runbook to connect in a consistent manner using a subscription ID and certificates that are already in that account. To connect to Azure, connection assets use credential assets as the part of the connection that contains authentication information, along with the subscription ID.

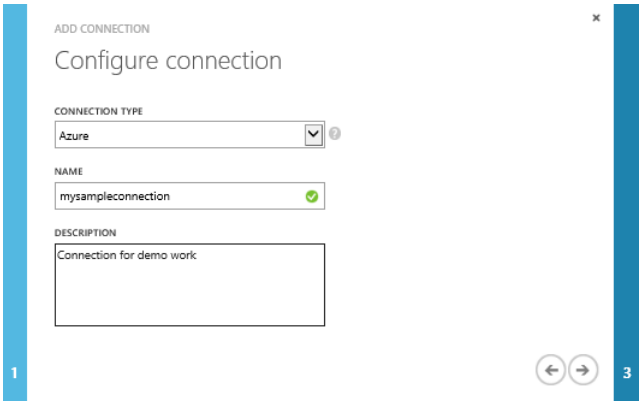
Creating a connection asset

To create a connection asset, do the following:

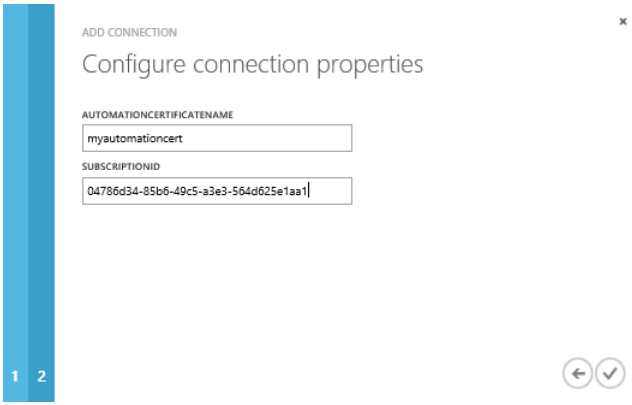
1. Open Notepad so that it's available to store some temporary data used to create the connection asset.
2. In the Azure Management Portal, click Automation, select the Azure Automation account, and then click Dashboard. In the Quick Glance section, scroll to find the subscription ID. Copy its value to Notepad.



3. In the Azure Management Portal, click Settings, click Management Certificates, and find the name of the automation certificate in the Name column. Copy the name to Notepad.
4. Click Automation, click the Azure Automation account, click Assets, and then click Add Setting. Select Add Connection and then in the Configure Connection dialog box, select the type of connection you want to add. The only connection type that is available at this time is Azure.



5. Click the right arrow to configure the connection properties. Here you need to add the name of the Azure Automation certificate and the subscription ID that you previously copied into Notepad. Click the check mark to complete the creation.



When you're done with this process, open the connection you just created to view the connection details. On the Connection Details page (see Figure 3-8), you find the name of the connection, the day and time it was last updated, its type, a description, the subscription ID, and the Automation certificate name.

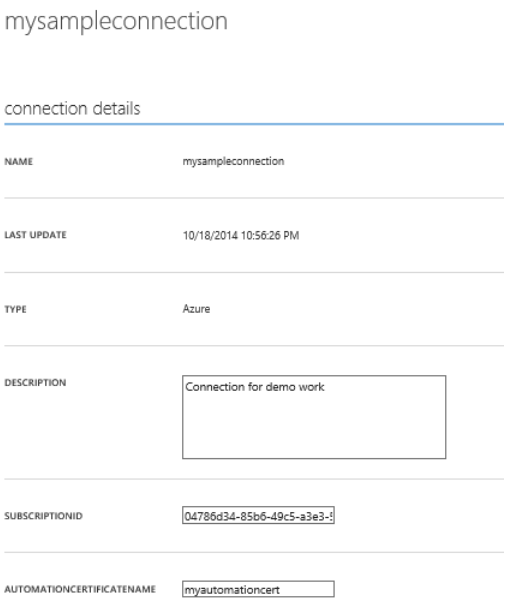


FIGURE 3-8 Properties of an established connection.

Using the Connect-Azure runbook

Microsoft has created a collection of sample runbooks and published them for free public download and use in the MSDN Library at Sample runbooks for Azure Automation. The Connect-Azure runbook is one of the most commonly used runbooks. This runbook is used to connect to an Azure subscription. You will probably want to import it into most of your runbooks that connect to Azure to do operations. A script that is running while hosted in an Azure-managed VM process must connect to your Azure subscription to access the resources you are trying to manipulate with the script.

You can import the Connect-Azure runbook into your Azure Automation account, and then publish it so other runbooks can call it. After it's imported, the Connect-Azure runbook can be a key part of your connectivity, leveraging assets for Azure Automation. This is because it uses the Azure connection and credential assets, which you need for any Azure Automation connection. It inserts the Azure Management certificate from the local machine store to set up a connection to the Azure subscription.

Before you use this runbook, you must make sure that you have an Automation certificate asset in Azure that holds the management certificate. You must also have a connection asset containing the name of the certificate and the subscription ID.

Before we talk in more detail about this runbook, recall that we mentioned earlier the ability to authenticate now with Azure AD and not have to use management certificates. At the end of this section I discuss a bit about how to do that. However, the concepts shown in the Connect-Azure runbook are still applicable and are great examples of how to use the credential and connection assets

together to authenticate the runbook to Azure. It also is a very good example of how to call an imported runbook from another runbook.

Calling the Connect-Azure runbook using certificates

The Connect-Azure runbook takes a connection name as a parameter. This parameter can be passed on the command line if you were invoking the connect-azure script from the command line. However, as it is an imported module, you will most likely call it from another runbook, passing in the AzureConnectionName parameter, which is an Azure Automation connection asset. The connection asset is a common connection object that you can define as an Automation asset in the Azure Management Portal to be used globally by many runbooks. When you create a connection asset, you specify the subscription ID. In addition, a certificate asset contains the management certificate that is correlated to that connection asset.

In the following code example, the Param block shows the mandatory (since =$true) string parameter that accepts the name of the connection. Several Get PowerShell commands take strings and output automation objects. In this case, the Connect-Azure runbook gets back an Automation connection object. The call to Get-AutomationConnection is wrapped in an exception block that will throw an exception and end the processing of the script if it can't find the named connection. After it gets the connection object, the script accesses the AutomationCertificateName property, again throwing an exception if it is unable to access that value. If the script can access the property, an AutomationCertificate object is returned. After the script has both the Automation connection object and the certificate object, the script then calls the Set-AzureSubscription cmdlet, passing in the connection name, the subscription ID (acquired from the connection object), and the certificate object.

```
workflow Connect-Azure
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]
        $AzureConnectionName
      )


    # Get the Azure connection asset that is stored in the Automation service based on the name that
was passed into the runbook
    $AzureConn = Get-AutomationConnection -Name $AzureConnectionName
    if ($AzureConn -eq $null)
    {
        throw "Could not retrieve '$AzureConnectionName' connection asset. Check that you created
this first in the Automation service."
    }
    # Get the Azure management certificate that is used to connect to this subscription
    $Certificate = Get-AutomationCertificate -Name $AzureConn.AutomationCertificateName
    if ($Certificate -eq $null)
    {
        throw "Could not retrieve '$AzureConn.AutomationCertificateName' certificate asset. Check
that you created this first in the Automation service."
```

```
    }
    # Set the Azure subscription configuration
    Set-AzureSubscription -SubscriptionName $AzureConnectionName -SubscriptionId
$AzureConn.SubscriptionID -Certificate $Certificate
}
```

You would call this runbook at the start of almost any Windows PowerShell Workflow script that is connecting to Azure to be able to work on those resources. By importing it, publishing it, creating global asset objects, and then calling it from another runbook using those assets, you can leverage common code over multiple runbooks and make it much easier to perform common tasks. Following is the code to call the Connect-Azure runbook from another runbook called Connect-AzureVM. Call Connect-AzureVM to set up a connection to an Azure VM. To do this, you first have to import and publish the Connect-Azure runbook.

```
workflow Connect-AzureVM
{
[OutputType([System.Uri])]
    Param
    (
        [parameter(Mandatory=$true)]
        [String]
        $AzureConnectionName,
        [parameter(Mandatory=$true)]
        [String]
        $ServiceName,
        [parameter(Mandatory=$true)]
        [String]
        $VMName
    )
    # Call the Connect-Azure runbook to set up the connection to Azure using the Automation connection
asset
     Connect-Azure -AzureConnectionName $AzureConnectionName

    InlineScript {
        # Select the Azure subscription we will be working against
        Select-AzureSubscription -SubscriptionName $Using:AzureConnectionName
        # Get the Azure certificate for remoting into this VM
        $winRMCert = (Get-AzureVM -ServiceName $Using:ServiceName -Name $Using:VMName | select
-ExpandProperty vm).DefaultWinRMCertificateThumbprint
        $AzureX509cert = Get-AzureCertificate -ServiceName $Using:ServiceName -Thumbprint
$winRMCert -ThumbprintAlgorithm sha1
        # Add the VM certificate into the LocalMachine
        if ((Test-Path Cert:\LocalMachine\Root\$winRMCert) -eq $false)
        {
            Write-Progress "VM certificate is not in local machine certificate store - adding it"
            $certByteArray = [System.Convert]::fromBase64String($AzureX509cert.Data)
            $CertToImport = New-Object
System.Security.Cryptography.X509Certificates.X509Certificate2 -ArgumentList (,$certByteArray)
            $store = New-Object System.Security.Cryptography.X509Certificates.X509Store "Root",
"LocalMachine"
            $store.Open([System.Security.Cryptography.X509Certificates.OpenFlags]::ReadWrite)
            $store.Add($CertToImport)
            $store.Close()
```

```
        }
    # Return the WinRM Uri so that it can be used to connect to this VM
    Get-AzureWinRMUri -ServiceName $Using:ServiceName -Name $Using:VMName
     }
}
```

The runbook calling order here is the Connect-AzureVM runbook calling the Connect-Azure runbook. If using inline script, the Connect-Azure runbook must be published first, and then the Connect-AzureVM runbook published after. The reason order matters is due to a feature in Azure Automation when using inline script, which is what both of these runbooks do. Any runbook called inside inline script must be published before its parent, the calling runbook. Inline Windows PowerShell script is used to run commands or expressions in a workflow that are valid in Windows PowerShell, but not valid in workflows.

To manage this feature, run the commands in an inlineScript activity. You also can use an inlineScript activity to run Windows PowerShell scripts (.ps1 files) in a workflow. The inlineScript activity runs commands in a standard, nonworkflow Windows PowerShell session and then returns the output to the workflow. It is valid only in workflows. The commands in an inlineScript script block run in a single session and can share data, such as the values of variables. By default, the InlineScript session runs out-of-process; that is, it runs in its own process, not in the workflow process, but you can change this default by changing the value of the OutOfProcessActivity property of the session configuration.

If the publishing doesn't occur in this order, an error message states that it can't find the called runbook. Even though the runbook exists and is published, if it's not published before its parent runbook, it will not be called. This problem can be hard to find.

Using Azure Active Directory without the Connect-Azure runbook

As mentioned in the "Azure Active Directory and automation" section at the start of this chapter, recent updates to Azure Active Directory, Windows PowerShell, and Azure Automation have given the option to authenticate without using certificates in favor of authenticating using Azure AD. Using Azure AD for authentication is the more strategic method than using certificates and is the clear future direction with respect to authentication, not just in Azure Automation, but in almost all Azure services that require that service.

In this case, the calling module (such as Connect-AzureVM) no longer needs to call the Connect-Azure runbook to authenticate. Instead, make calls to Get-AutomationPSCredential and pass in the name of the Azure AD Automation account that the script is running under and that needs to authenticate. This action returns a credential object that is immediately passed in the call to Add-AzureAccount. Here is another code sample of this authentication process to show you how it is done as a recommended best practice.

```
workflow MySampleWorkflow
{
```

```
    param
    (
     #include your list of parameters
    )

    # Get the credential to use for Authentication to Azure.
    $Cred = Get-AutomationPSCredential -Name 'Azure AD Automation Account'

   # Connect to Azure
    $AzureAccount = Add-AzureAccount -Credential $Cred

    # Begin processing of workflow

}
```

Schedule assets

When you want to execute your runbooks automatically at either a specific date and time or on a recurring basis, you can use a schedule asset. No manual intervention is necessary to start schedule assets. Azure will allocate resources, load, and then execute the runbook when the schedule triggers. When the script completes, Azure will manage the release of execution resources.

Although schedules are assets, they differ slightly from assets such as connections, certificates, and variable assets. The difference is that you never insert or call a schedule from script code. Rather, you will link a runbook to a schedule. A schedule asset determines when runbooks that are linked to it can run. A schedule asset triggers runbook execution when the schedule is activated. You select a published runbook, and on its Schedule tab, you can choose to link to a new schedule. Draft runbooks cannot be linked to a schedule.

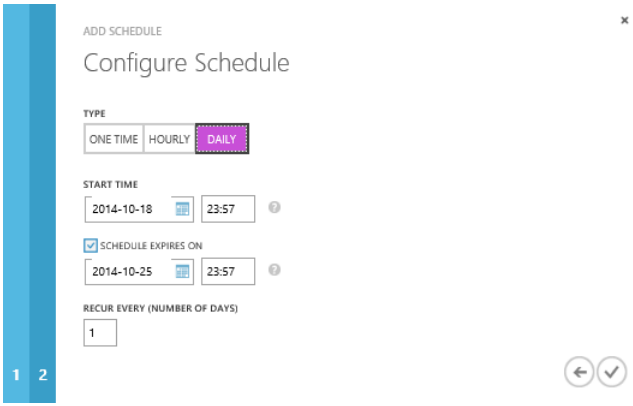
Schedule assets are the alternative for manual invocation of runbooks or being called by code from another runbook. Scheduling is just a deeper level of automation beyond just having a script: It's like automating the automation!

Creating a schedule asset

To create a schedule asset, do the following:

1. In the Azure Management Portal, click Automation, select the Azure Automation account, and then click Assets. Click Add Setting.
2. In the Add Setting dialog box, click Add Schedule. On the Configure Schedule page, enter a name that is unique to that Azure Automation account, enter an optional description, and then click the right arrow.
3. In the Configure Schedule dialog box, you can choose to run the schedule one time, hourly, or daily. Depending on the option you select, the remainder of the dialog box entry fields will change slightly. The options are as follows:

- **One Time** Choose a start date and time.
- **Hourly** Choose a start date and time. As an option, you can select the Set Schedule Expiration Time check box and then enter a date and time to ensure the schedule expires at that date and time. Enter a value in the Recur Every (Number Of Hours) field.
- **Daily** The same settings are available here as in Hourly, but you choose the number of days to recur instead of number of hours.

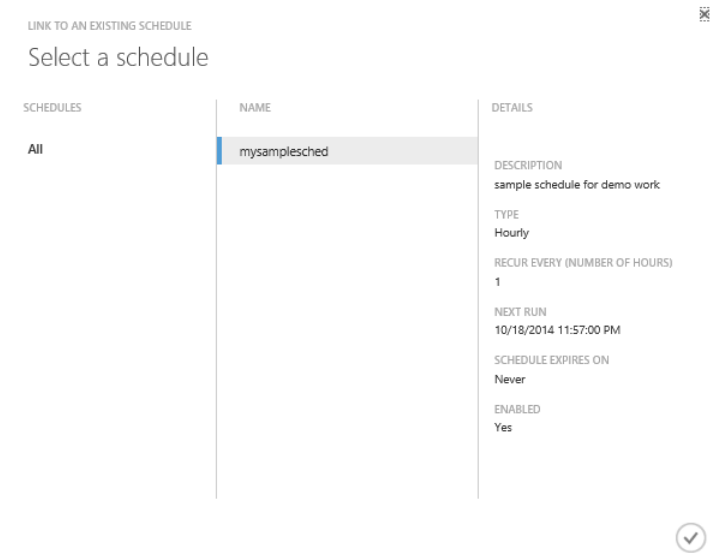


Configuring the granularity of scheduling options here is not like setting up a meeting in Outlook. You can only run a schedule at a maximum once per hour and you set an expiration time for it to end. For example, if you set the schedule Start Time as 8 pm on September 1 for a recurrence of every two days, and you set the Schedule Expires On date to September 10, the schedule would run on September 1, 3, 5, 7, and 9, but it will not run on September 11 because the expiration date is September 10.

Using the schedule

After you create the schedule, you can link it to a runbook. In the Azure Management Portal, display the list of runbooks for the Azure Automation account, and verify that the runbook that you want to link to a schedule is published. If the runbook isn't published, publish it before you try to link a schedule to it.

In the Azure Management Portal, select the runbook, click Schedule, and then choose to either link to a new schedule or an existing schedule. If you select a new schedule, you can use the Configure Schedule dialog box to create a new schedule. If you select an existing schedule, use the Link To An Existing Schedule dialog box to choose a schedule and display its details.



If the runbook you are linking to the schedule has no parameters that it needs specified, click the check mark to complete the link process. If the runbook does require parameters, however, you will have to specify the runbook parameter values. As shown in Figure 3-9, in the Specify The Runbook Parameter Values dialog box, enter values to pass to the runbook automatically when the schedule invokes. If you manually invoke a runbook that has parameters, you enter the values when the runbook is run. However, the Connect-AzureVM runbook requires you to enter values for all three fields when linking it to a schedule instance.

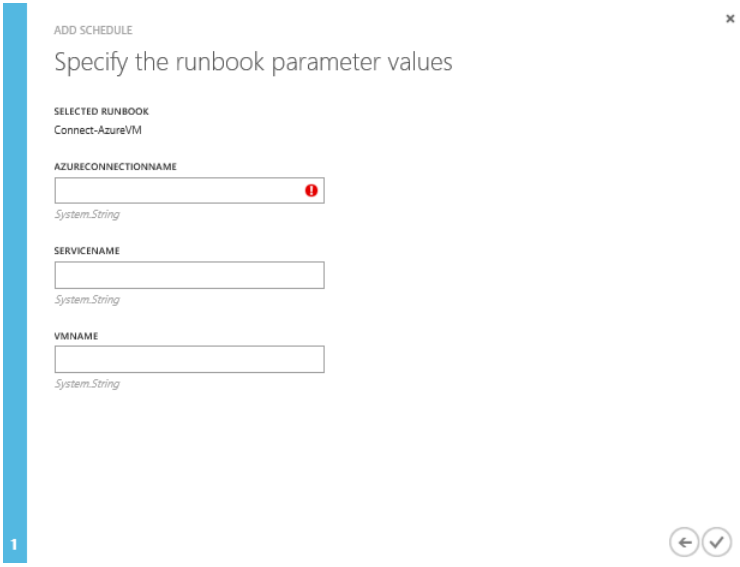


FIGURE 3-9 Enter values for parameters when specifying the runbook parameter values.

Chapter 4

Runbook deployment

Runbook deployment refers to a runbook that has already been through the authoring process and now needs to be published and run. After it's published, a runbook can be invoked in several ways. It can be invoked from code, manually, or using a schedule. Child runbooks can be invoked using the inline method versus being invoked with the activity Start-AzureAutomationRunbook. Backing up runbooks, configurations, and assets for an Azure Automation account is an important part of runbook management.

Finally, sometimes things don't go the way you expect in a deployment scenario. Error messages, job status, and other information in the Azure Management Portal dashboard can be useful in troubleshooting a runbook's execution. Reviewing log files can also help you troubleshoot problems with runbook deployment.

Publishing a runbook

Before you promote a runbook to a published state, test it to ensure it's ready to use. Realize that a test run of a runbook still runs all the code and does what it needs to do in reality. Put another way, the test run is real! There isn't an Undo switch to hit after the runbook test completes. You can use test assets or parameters that contain test environment information and point to the test environments. When the runbook is ready to be published, you can switch them into a runtime environment.

Promote a runbook from draft status into a publish status after it's tested and you're sure it works correctly. If later you want to go back and modify it, you can toggle it back into Draft mode, edit it, and then publish it again from the Author pane in the Azure Management Portal. While you are editing a runbook, the published version will be the version called by schedules or cmdlets. This allows you to edit while the published version runs.

A runbook needs to be published to be started. You can't start a runbook in Draft mode. Also, if you want to link it to a schedule or call it from another runbook, it must be published. All draft runbooks can run only in Test mode. For the sake of testing, a draft runbook does not need to be published to call other runbooks in Test mode. Once published, it can be called from other runbooks.

To publish a saved runbook, in the Azure Management Portal, select it from the list of runbooks and then click Publish. As shown in Figure 4-1, along with Publish, you can also manage the runbook by importing a module or adding a setting; insert an activity, runbook, or setting; save the runbook; discard the draft and only keep the published version; and test the runbook. A runbook that is still in Draft mode can only be run by clicking Test. If you have not saved the runbook before you run it in Test mode, the Azure Management Portal prompts you to save the runbook.

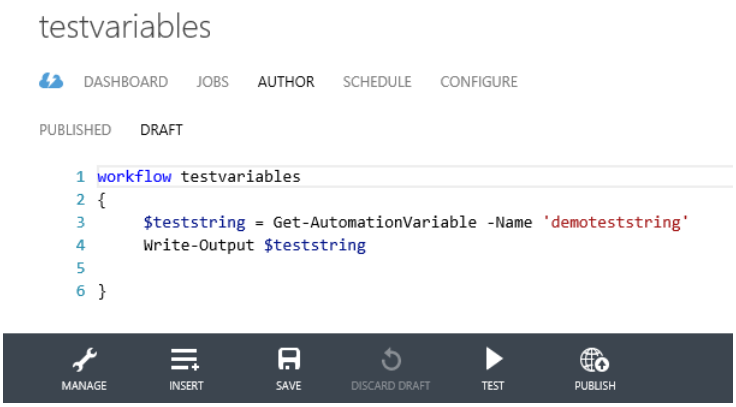


FIGURE 4-1 Toolbar and Publish button used to publish a draft runbook.

Invoking a runbook

Within the runbook lies the key to almost all of the operations that need to take place to automate your Azure deployments. You can invoke a runbook in the following ways:

- **Invoke from code within another runbook** You can be called from the parent runbook when that parent runbook is invoked. Calls can be made directly, using InlineScript, or using Start-AzureAutomationRunbook.
- **Invoke manually** You can run it manually when you aren't sure of the exact time and date it will need to be run, or based on dynamic events or occurrences that don't happen regularly.
- **Invoke via schedule** You can schedule it to run at a certain date and time, which could be one time or daily.

Invoke from code within another runbook

The scope of called and calling runbooks inline is limited to those that have been imported into an Automation account. A runbook cannot invoke, or be invoked, inline by runbooks outside of its Automation account. However, you can call Automation runbooks from another account asynchronously using the Start-AzureAutomationRunbook cmdlet. To make this call, pass in the AutomationAccountName parameter, along with the name of the runbook to be invoked.

```
Start-AzureAutomationRunbook -AutomationAccountName "MyAutoAccount" -Name "MyRunBook"
```

For a runbook to be called by another runbook, it must be published first. It cannot be called from another runbook if it's still in New (draft) status. The category of New will be displayed in the Azure Management Portal for a runbook in draft status.

However, a new runbook that has not been published can invoke a published runbook. The calling runbook does not need to be published to make the call. When testing the logic of the calling runbook is likely the only time you would have an unpublished runbook call a published runbook. In a real-world deployment, it's highly unlikely you would ever have a test runbook call a published runbook.

The syntax for calling a runbook is the same as calling any other Windows PowerShell workflow activity. The runbook might or might not return a value to the calling runbook.

Here is a very simple example of a parent runbook, Call-WriteHelloWorld, and the child runbook that it calls, Write-HelloWorld.

```
workflow Call-WriteHelloWorld

{

 Write-HelloWorld -Name "myvalue"

}
```

Within Write-HelloWorld, the logic will either append the Name parameter to the output string or use the default value for the Name parameter that is defined within the workflow itself.

```
 <#

.SYNOPSIS

    Provides a simple example of an Azure Automation runbook.

.DESCRIPTION

    This runbook provides the "Hello World" example for Azure Automation.  If you are

    brand new to Automation in Azure, you can use this runbook to explore testing

    and publishing capabilities.

    The runbook takes in an optional string parameter.  If you leave the parameter blank, the

    default of $Name will equal "World".  The runbook then prints "Hello" concatenated with $Name.


.PARAMETER Name

    String value to print as output


.EXAMPLE

    Write-HelloWorld -Name "World"


.NOTES
```

```
   Author: System Center Automation Team

   Last Updated: 3/3/2014

#>

workflow Write-HelloWorld {

    param (


        # Optional parameter of type string.

        # If you do not enter anything, the default value of Name

        # will be World

        [parameter(Mandatory=$false)]

        [String]$Name = "World"

    )

        Write-Output "Hello from new my script - $Name"

}
```

You can manually start the parent runbook Call-WriteHelloWorld from the Azure Management Portal by clicking Start. After you start the job, you can switch to the Runbooks tab and see the job starting, as shown in Figure 4-2.

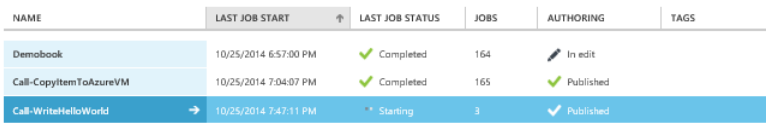


FIGURE 4-2 Call-WriteHelloWorld runbook status of Starting after being started from the Azure Management Portal.

You can go to the Dashboard for Call-WriteHelloWorld and see the results of the jobs. In Figure 4-3, you can see that three jobs completed successfully.

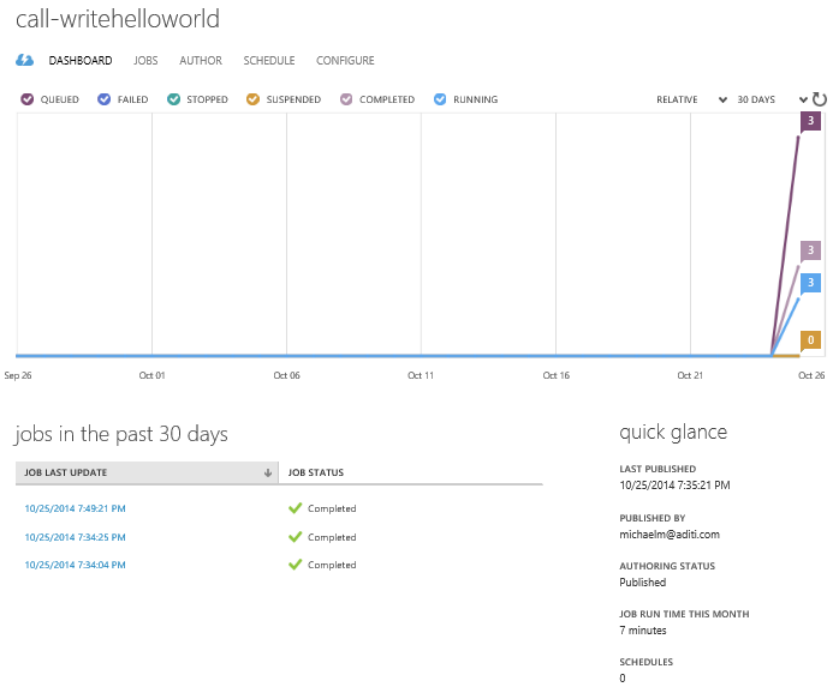


FIGURE 4-3 Dashboard display for Call-WriteHelloWorld runbook.

The Call-WriteHelloWorld parent runbook calls its child runbook Write-HelloWorld with a parameter of myvalue. If a parameter wasn't in the call, HelloWorld would use its default value of World. If you click on one of the jobs, the summary of that job run and the output from the operation is shown in Figure 4-4.

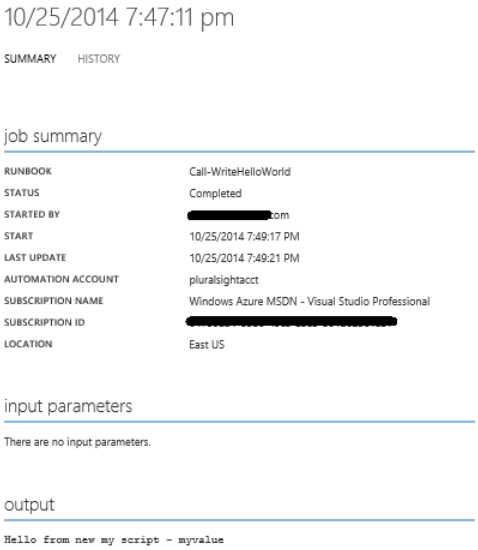


FIGURE 4-4 Job summary for a specific Call-WriteHelloWorld runbook job.

One of the best practices when creating code of any kind is modularization: creating discrete, reusable code units. For Azure Automation, this means putting self-contained tasks within runbooks, and then calling those runbooks from within other runbooks that need the functionality. Thus, it could be common practice for a parent runbook to call one or more child runbooks as part of the process being executed.

Note The terms *nested* and *child* refer to any child runbook that is called from a parent runbook.

There are two ways to call child runbooks in Azure Automation. You can invoke the runbook inline, or you can start a runbook with the Start-AzureAutomationRunbook cmdlet.

Invoke a child runbook using inline scripts

Runbooks that are invoked inline run in the same job as the parent runbook. This means that the parent workflow will wait for a child runbook to finish (synchronous) before continuing with the next part of the process. This also means that all exceptions thrown by the child runbook and all stream output produced by the child runbook are associated with the parent job; therefore, all tracking of child runbook execution and output will be through the job associated with the parent runbook.

When you start a runbook that has child runbooks invoked inline, the child runbooks (and all descendants) are compiled into the parent runbook before execution starts. During this compilation phase, the parent runbook is parsed for the names of child runbooks. This parsing happens recursively through all descendant runbooks. When the complete list of runbooks is obtained, the scripts for these runbooks are retrieved from storage and assembled into a single file that is passed to the Windows

PowerShell workflow engine. For this reason, at the time a runbook job is submitted, the parent and descendant runbooks must already be published. Otherwise, an exception will occur during compilation.

The order of publishing also matters: You must publish the child runbooks first and then publish the parent. Similarly, when testing the parent runbook, you must first publish the child runbooks and then you can test the parent. Another consequence of this requirement is that you cannot use a variable to pass the name of a child runbook invoked inline: You must always explicitly name the child runbook within the parent runbook.

This is a simple synchronous model of execution where the parent calls the child runbook that waits for the call to complete and return. Because called inline scripts run in the same job environment as the parent, there are fewer jobs running in the system. For instance, if an exception occurs from the child runbook, it's associated with the same parent job, which makes debugging logically easier. This simple flow of the parent making a synchronous call into the child makes it much simpler to directly return output result data back to the calling parent. A restriction is that both the parent and child runbooks have to be in the same Azure Automation account.

Invoke a child runbook using Start-AzureAutomationRunbook

Another option you can use to invoke a child runbook is to asynchronously start it from another runbook by using the Start-AzureAutomationRunbook activity. When the child runbook is called from another runbook, a new automation job is created for the called runbook. This multijob approach is helpful when the calling runbook needs to create an additional thread of concurrent work so that it can do its own tasks without waiting for the called job to complete. However, both runbooks run in different jobs, meaning the exceptions and output are stored in different locations. This increases the complexity of debugging because the exceptions need to be reviewed separately.

To access the results of the called child runbook, the calling parent runbook can use other Azure Automation cmdlets. The Get-AzureAutomationJob cmdlet can be called on the child runbook to obtain a job, using its globally unique identifier (GUID) to identify the job. It passes in the type of the objects being piped to the activity, and the output type is the type of the object that the cmdlet produces. Using cmdlet piping, the output of Get-AzureAutomationJob is passed to the Get-AzureAutomationJobOutput cmdlet to obtain the output of the child's work for the GUID that identified the job. Calling a job asynchronously, as well as obtaining the output from an asynchronously called job, can add more complexity to your code.

As with using the inline script option, there are trade-offs to using the Start-AzureAutomationRunbook approach.

- The calling parent runbook calls the child runbook asynchronously. Because the call doesn't block and wait for the call to the child runbook to return, simultaneously concurrent and productive execution can occur at the same time between the caller and the called threads of execution. This parallel work can be further divided among two or more Azure Automation

accounts within an Azure subscription. The name of the runbook to be invoked is passed in as a parameter to the called child runbook.

- The parameters are limited in types due to the fact that the JSON object serialization takes place through a Web service call.

Use Start-ChildRunbook to start an Azure Automation job

An example runbook provided by Microsoft, Start-ChildRunbook, shows how to call a child runbook asynchronously and then obtain the job output.

The Start-ChildRunbook runbook offers different options for starting the job. Depending on the parameters passed in, Start-ChildRunbook can be called and return the ID of the job if the call is to be made asynchronously. If the call is to be made synchronously, it can wait for the job to complete. Other key parameters passed in include the polling interval to check if the child runbook job is complete as well as a maximum poll time that will throw a timeout exception if the child runbook job doesn't return.

> **Note** The Start-ChildRunbook runbook is a large file, so it's not shown here in its entirety. You can download the runbook to review and use from the Microsoft Script Center at Start Azure Automation Child Runbook.

First, the runbook must authenticate to Azure so the script can get permissions to execute. It then selects the subscription that the script will run against.

```
# Connect to Azure so that this runbook can call the Azure cmdlets
    Add-AzureAccount -Credential $AzureOrgIdCredential

# Select the Azure subscription we will be working against
    Select-AzureSubscription -SubscriptionName $AzureSubscriptionName
```

You can use InlineScript to convert parameters to those that have scope. Then, invoke the Start-AzureAutomationRunbook.

```
InlineScript
    {
        # Convert the parameters in the Workflow scope into parameters in InlineScript scope
        $AutomationAccountName = $using:AutomationAccountName
        $ChildRunbookInputParams = $using:ChildRunbookInputParams
        $ChildRunbookName = $using:ChildRunbookName
        $JobPollingIntervalInSeconds = $using:JobPollingIntervalInSeconds


        $JobPollingTimeoutInSeconds = $using:JobPollingTimeoutInSeconds
        $ReturnJobOutput = $using:ReturnJobOutput
        $WaitForJobCompletion = $using:WaitForJobCompletion
```

```
    if ($ChildRunbookInputParams -eq $null) { $ChildRunbookInputParams = @{} }
    # Start the child runbook and get the job returned
     $job = Start-AzureAutomationRunbook `
                 -Name $ChildRunbookName `
                 -Parameters $ChildRunbookInputParams `
                 -AutomationAccountName $AutomationAccountName `
                 -ErrorAction "Stop"
```

The job status needs to be watched and polled regularly to look for either a completion status or an exception due to a timeout threshold being exceeded.

```
if ($WaitForJobCompletion -eq $false)
        {
            # Don't wait for the job to finish, just return the job id
            Write-Output $job.Id
        }
        elseif ($WaitForJobCompletion -eq $true)
        {
            # Monitor the job until it finishes or the timeout limit has been reached
            $maxDate = (Get-Date).AddSeconds($JobPollingTimeoutInSeconds)


            $doLoop = $true


            while($doLoop) {


                Start-Sleep -s $JobPollingIntervalInSeconds


                $job = Get-AzureAutomationJob `
                    -Id $job.Id `
                    -AutomationAccountName $AutomationAccountName


                $status = $job.Status


                $noTimeout = ($maxDate -ge (Get-Date))


                if ($noTimeout -eq $false) {



                    throw ("The job for runbook $ChildRunbookName did not complete within
the timeout limit of $JobPollingTimeoutInSeconds seconds, so polling for job completion was
halted. The job will continue running, but no job output will be returned.")
                }
```

```
                    $doLoop = (($status -ne "Completed") -and ($status -ne "Failed") `
                              -and ($status -ne "Suspended") -and ($status -ne "Stopped") `
                              -and $noTimeout)
                }
```

After the job has finished, if it completed successfully, the runbook uses the Get-AzureAutomationJobOutput cmdlet to obtain the job's output string. If it did not complete successfully, the runbook raises an exception.

```
    if ($job.Status -eq "Completed")
                {
                    if ($ReturnJobOutput)
                    {
                        # Get the output from job
                        $jobout = Get-AzureAutomationJobOutput `
                            -Id $job.Id `
                            -Stream Output `
                            -AutomationAccountName $AutomationAccountName
                        # Return the output string
                        Write-Output $jobout.Text
                    }
                    else
                    {
                        # Return the job id
                        Write-Output $job.Id
                    }
                }
                else
                {
                    # The job did not complete successfully, so throw an exception
                    $msg = "The child runbook job did not complete successfully."
                    $msg += "  Job Status: $job.Status.  Runbook: $ChildRunbookName.  Job Id:
$job.Id."
                    $msg += "  Job Exception: $job.Exception"
                    throw ($msg)
                }
            }
```

Here is an example of how Start-ChildRunbook could be called. Most of the parameters are self-explanatory, but here are a few key points:

- The input parameters are passed as a key/value hashtable. In the following example, three input parameters are passed in the table.
- The credential asset must be established previously and is passed in as $mycredential.
- WaitForJobCompletion is set to $true, meaning the caller will wait synchronously for the called

child job to complete.

- Start-ChildRunbook will poll every 10 seconds and, after 3 minutes (180 seconds), a timeout exception occurs.

```
Start-ChildRunbook `
        -ChildRunbookName "MyCalledRunbook" `
        -ChildRunbookInputParams @{'myval1'='111';'mval2'=22;'myval3'="myvaluethree"} `
        -AzureSubscriptionName "MyAzureSubscriptionName" `
         -AzureOrgIdCredential $mycredential `
        -AutomationAccountName "myautomationaccountname" `
        -WaitForJobCompletion $true `
        -ReturnJobOutput $true `
        -JobPollingIntervalInSeconds 10 `
        -JobPollingTimeoutInSeconds 180
```

As a job proceeds through its processing, it passes through many states. Table 4-1 shows the job states that are valid as of this writing.

Table 4-1 States of a runbook processing job

Job Status	Status Description
Suspending	A job was requested to be suspended by the user, but first must complete its next checkpoint.
Suspended	A job can be suspended by the user, a runbook cmdlet call, or the system. Only if an exception occurs can a job be suspended by the system.
Stopped	A user stopped the job before it was allowed to complete.
Stopping	A job is currently in the process of being stopped.
Running	A job is running currently.
Running, waiting for resources	A job has reached the "Fair Share" limit. This means Azure unloads a job after it has been running for 30 minutes and restarts it from its last checkpoint.
Starting	A worker server has been assigned the job and the job is starting.
Resuming	A job was suspended but is being restarted.
Queued	Currently there are not enough resources free to start the job.
Failed	An error occurred and the job did not complete successfully.
Failed, waiting for resources	A job failed to reach what is called the "Fair Share" limit three times, and was started from the start of runbook or same checkpoint each time.
Completed	A job successfully finished.

Invoke a runbook manually from the Azure Management Portal

Another way to invoke a runbook is to do so manually from the Azure Management Portal. Use this approach when you need to run a job immediately and don't want to set up a schedule or call it externally. You might want to do this for a runbook that doesn't need to be called on a regular basis but only occasionally as needed. For instance, you might be in a test scenario that might run for different periods of time. You want to automate the deletion of that entire environment when you're done using it. You can do this by manually invoking the runbook to do that immediately after the testing is done.

You can imagine a few different manual invocation scenarios where your application might occasionally need to scale up when a certain external event occurs. You could have a runbook that

monitors your cloud service for heavy usage and then calls another runbook that handles the scaling up or down. You might also have a situation where the person on call might have a set of runbooks that would remediate a problem and the logic to determine the correct remediation steps can't be coded. This is something you can't schedule or plan for. When you see that the need to scale must occur due to external events, you want to be able to get servers up and running to handle the situation.

To manually invoke a published runbook, log in to the Azure Management Portal, select the Automation icon, choose an Automation account, click the Runbooks tab, and see what is shown in Figure 4-5. Choose a particular runbook and click Start. Before the runbook starts, you have one opportunity to prevent the runbook from running. If you click Yes, the runbook will start running.

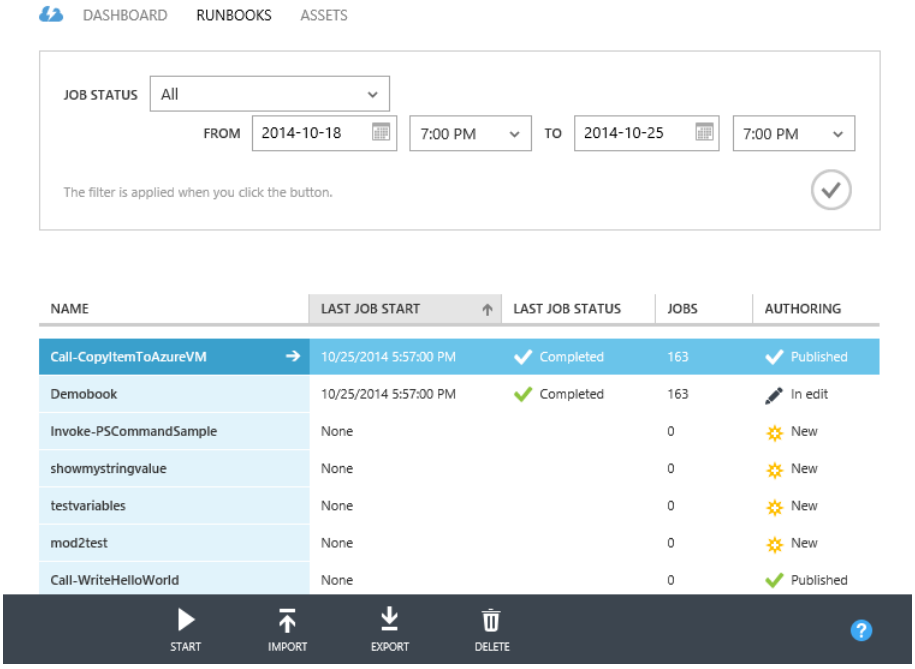


FIGURE 4-5 Toolbar and Publish button used to publish a draft runbook.

The Copy-ItemToAzureVM runbook is another common runbook that you can invoke manually. When it is invoked, it will require seven mandatory parameters passed to it because the Mandatory attribute is set to $true. Here is the parameter declaration for the Copy-ItemToAzureVM activity.

```
workflow Copy-ItemToAzureVM {
    param
(       [parameter(Mandatory=$true)]
        [String] $AzureConnectionName,
        [parameter(Mandatory=$true)]
        [String] $ServiceName,
```

```
        [parameter(Mandatory=$true)]
        [String] $VMName,
        [parameter(Mandatory=$true)]
        [String] $VMCredentialName,
        [parameter(Mandatory=$true)]
        [String] $LocalPath,
        [parameter(Mandatory=$true)]
        [String] $RemotePath
    )
```

As an example, a call to the Copy-ItemToAzureVM activity looks like the following:

```
Copy-ItemToAzureVM
-AzureSubscriptionName "Visual Studio Ultimate with MSDN"
-ServiceName "myService"
-VMName "myVM"
-VMCredentialName "myVMCred"
-LocalPath ".\myFile.txt"
-RemotePath "C:\Users\username\myFileCopy.txt"
-AzureOrgIdCredential $cred
```

When you run the Copy-ItemToAzureVM runbook manually via the Azure Management Portal, you can enter values for these parameters. Figure 4-6 shows the Specify The Runbook Parameter Values dialog box in the Azure Management Portal. Values are shown for the first three of the seven parameters.

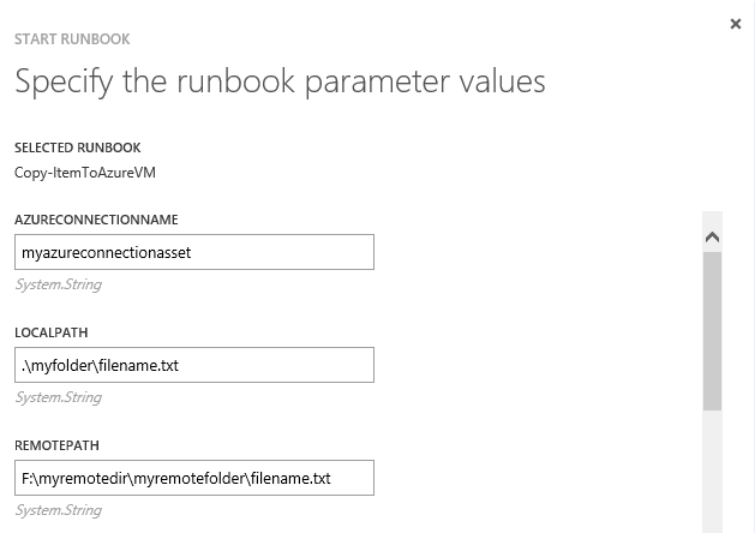


FIGURE 4-6 Parameter entry for the Copy-ItemToAzureVM runbook.

After you provide values for all the parameters, click the check mark to start the runbook using

those specified parameters. Any output meant to be displayed to the screen will be shown in the Azure Management Portal.

> **Note** You can download the Copy-ItemToAzureVM runbook from the Microsoft Scripting Center at Copy a File to an Azure VM.

Invoke a runbook using a schedule asset

You might want to run operations regularly or at normal intervals. You can create a schedule that continuously runs and then link a runbook to that schedule.

You can establish a schedule for the runbook to run once on a specific date and time in the future. The scheduled start time must be at least 5 minutes past the current time for the system to run it. Therefore as a best practice, if you need to run a runbook in a time span less than 5 minutes away, you should just run it manually from the Test option in the Author panel for a job.

The time zone of the schedule on the Azure Management Portal is based on the computer time zone setting where the Azure Management Portal is being used to author the script. The time shown is by default shown as +30 minutes from the time zone in the Author's computer.

Alternatively, a runbook can be scheduled to run daily with a set number of days apart (as of this writing, the maximum is 99 days between runs). To run the schedule on specific days, you have to manually add logic to check the day of the week in the runbook itself. For an example of how to do this, see Check if current time is in work hours in the Script Center.

To create a schedule, on the main screen for your Azure Automation account, click the Assets tab and then click Add Setting. This gives you the options to add a connection, a credential, a variable, and a schedule. Click Add Schedule. In the first Configure Schedule dialog box, enter the schedule name. Azure will check within that Azure Automation account to ensure that the name is available. You can also enter an optional description, and then click the right arrow to go to the final Configure Schedule page. You can also create a new schedule or link to an existing one via Runbooks > Schedules.

When you're configuring a schedule, choose a type of One Time, Hourly, or Daily, as shown in Figure 4-7. One Time has only a date and start time from which to choose. Hourly and Daily are recurring in nature. They have a start time, an optional schedule expiration time, and a recurrence of every number of hours or days, respectively.

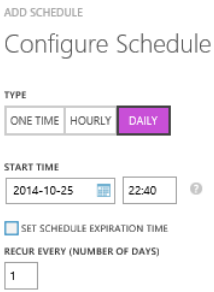


FIGURE 4-7 Configuring a schedule to link to runbooks.

After the schedule is created, you link to one or more published runbooks. Draft runbooks can't be called by other runbooks and they also can't be scheduled. Select a published runbook to go to its main page. Next to the Author tab at the top of the page is the Schedule tab, where you have the options of linking to a new schedule or an existing one, as shown in Figure 4-8.

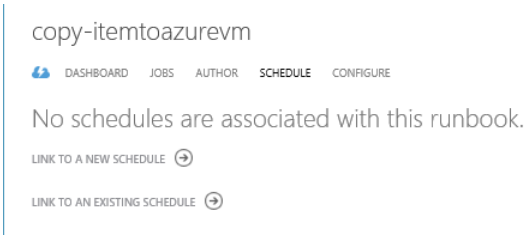


FIGURE 4-8 Configuring a runbook to link to a new or existing schedule.

If you choose to link to a new schedule, the Configure Schedule dialog box is displayed for your use, as shown in Figure 4-7. If you want to link to an existing schedule, the Select A Schedule page allows you to choose a schedule, as shown in Figure 4-9.

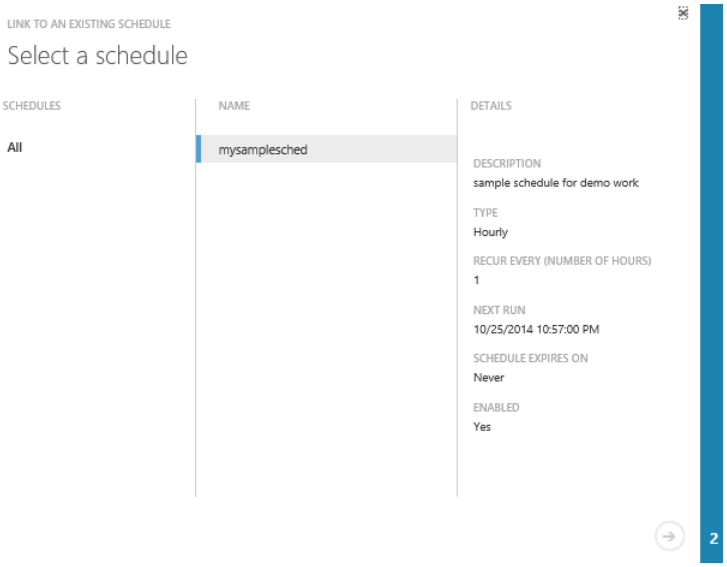


FIGURE 4-9 A runbook linking to an existing schedule.

Choose the schedule (in Figure 4-9, there is only one choice), and then click the right arrow to proceed. The Specify The Runbook Parameter Values page (Figure 4-10) allows you to enter the parameters ahead of time for this runbook so that when Azure starts the schedule at the desired time(s) there is no need for user interaction to obtain the parameter values.

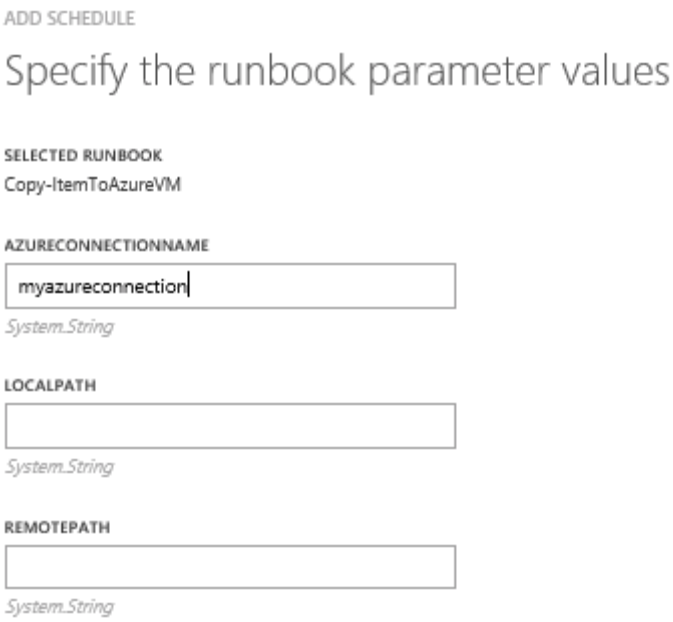


FIGURE 4-10 A runbook linking to an existing schedule.

When the schedule is triggered, the jobs associated with it run. The job of the schedule ends after the job completes normally.

Troubleshooting a runbook

Suppose that the invocation of a runbook does not complete as planned for a job invoked by a schedule, or by any other means. How can you troubleshoot or debug the process to figure out why the runbook is failing?

Use the Dashboard

One of the best ways to start your troubleshooting is with the information on the Dashboard, which is what you see when you first open the Azure Management Portal under Azure Automation for a specific Azure Automation account. When you open that Dashboard, you will see a chart, and sections for Usage Overview, Jobs In The Past 30 Days, and Quick Glance, as shown in Figure 4-11.

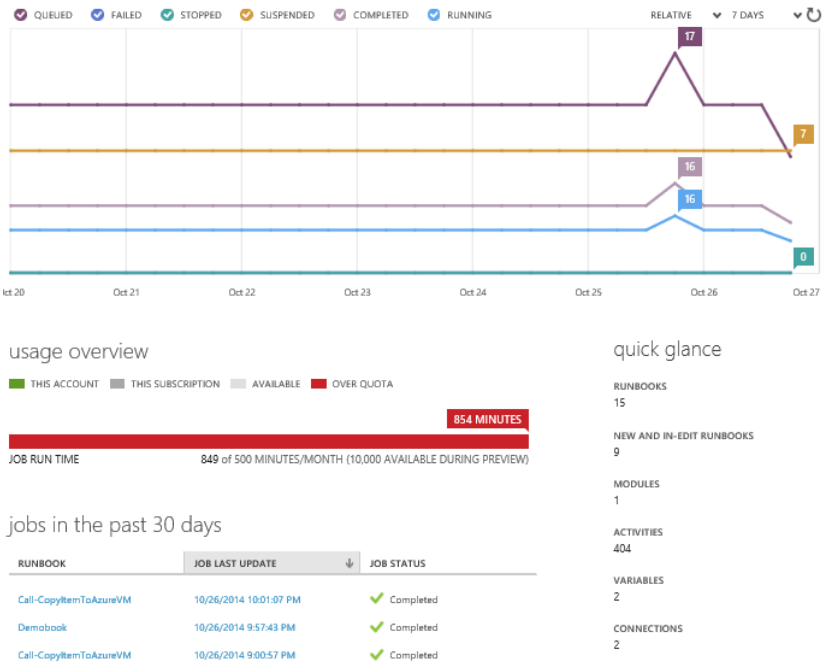


FIGURE 4-11 Dashboard for a runbook showing the details of the runbook's lifetime and characteristics.

- **Chart** This section shows the status of each runbook job during the time period you choose (from the last hour up to the last 30 days). In this chart, you can quickly see how many jobs are

running right now, how many have completed, and most important, you can see any jobs that require your attention—those that are suspended or failed. The icons on the top of the chart represent each possible job state. Click these icons to toggle particular status lines, which allows you to focus on job states of particular interest.

- **Usage Overview** This section identifies your current usage against your quotas, for job execution time, number of runbooks, and chapter size. You can use this information to measure how much more usage of the system you have available before you'll have to start paying for Azure Automation (postpreview), or if you're already paying, how much more you'll be able to use the service before you have to move to an upgraded plan.
- **Jobs In The Past 30 Days** This section contains an entry for each job that was started in the last 30 days, and shows the name of the runbook (workflow), the time the job was last updated, and the current status of the job. Thus, if there are any recent runbook jobs that require your attention, you can use this table to quickly identify the exact jobs and then drill in to troubleshoot.
- **Quick Glance** This section contains useful static information about your Automation account, such as the number of runbooks, chapters, and assets. It also indicates how many runbooks are currently in an authoring state.

Enable logging

Another good way to debug the scripts is to run them with various levels of logging enabled. Additionally, from within the scripts in a workflow, you can write your own debug messages to the logs, and then remove them when you are ready to publish them. There is no way, such as a compiler switch, to explicitly enable messages for debug mode and disable them for release mode in this environment. However, you can enable the Verbose and Progress logging levels, and leave the Configuration off. If problems occur, you can then enable this logging configuration and run the runbooks again.

Tip Scripts typically fail due to external values changing unexpectedly.

On the Configure tab, you can choose different levels of error logging for each Runbook, as shown in Figure 4-12. If you enable Verbose logging, errors are logged at all levels. You can also enable Progress logging, which logs significant ordered events in the execution lifetime of the runbook.

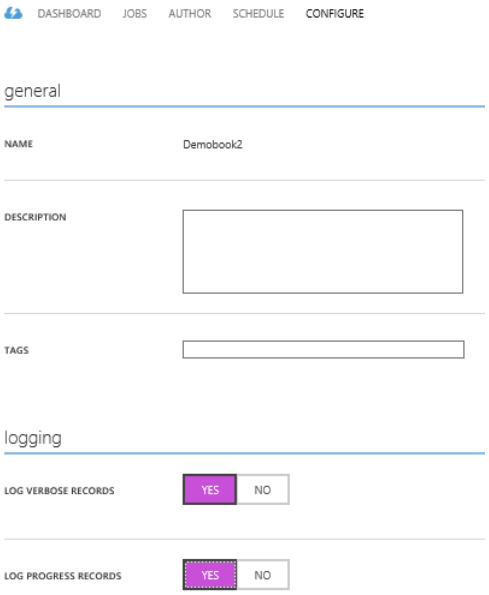


FIGURE 4-12 The Configure tab for a runbook.

You can see the results of the job on the History tab only after a job completes. If you enable logging, you will have a more verbose record of execution to track.

You identify a job by its timestamp for that runbook. Click the runbook, and then click the Jobs tab. Click a job, and then click the History tab to see the History view for that job and the logging information, as well as the data output for that specified job (shown in Figure 4-13). All error log output is always displayed regardless of what you set for logging options. All error logging is done automatically for you by Windows PowerShell.

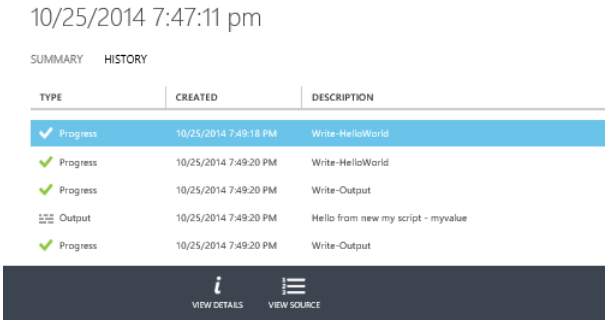


FIGURE 4-13 History with logging of a job's execution.

Click View Source on the History tab to see the source code that was run for that version of the job. Note if you change the source code and submit the job again, this job will always keep the copy of the

source code that was run for its invocation. You can use this as a way to keep track of the versions and changes you make to the script because that information is captured on every run. Azure Automation does store old published versions of your runbooks so there is a way to bring back and run a previous published version. By using View Source when you're troubleshooting, you can see the source code used for a job that ran successfully previously and then compare the changes to a later job that failed.

Click View Details to display troubleshooting information about the execution of the activity.

Backing up a runbook

When an Azure Automation account is deleted, all its runbooks are deleted as well and cannot be retrieved later if needed. Therefore, as a best practice, it's important to back up a runbook (via the export process) before you delete the Azure Automation account associated with that runbook. When backing up an Automation account before deleting it (and all its runbooks), export any runbooks you want to use later in other Automation accounts.

Exporting is useful even if we are not deleting an automation account but just want to share a generic runbook with another Automation account in general. Be sure when exporting that any references to global assets or calls to runbooks that exist only in the soon-to-be-deleted automation account are considered when moving (not copying) runbooks from one Azure Automation account to the other. For example, if you access a credential asset in the old runbook, when moving it to the new Azure Automation account, make sure you have either created a credential asset identical to the name of the one in the old account in the new account, or that you have removed the code from the runbook after it's imported into the new Automation account.

To export a runbook in the Azure Management Portal, go to the Azure Automation account and then select a runbook. Click Export (Figure 4-14), choose Yes to indicate that you want to export this runbook, and then save the runbook as a PS1 file at the location of your choice.

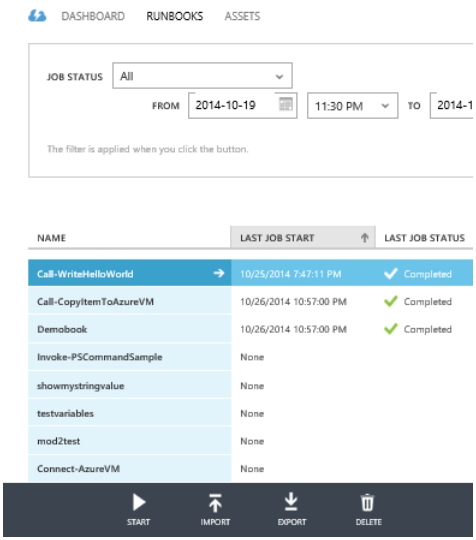


FIGURE 4-14 The Runbooks tab with options to import and export a runbook.

When you create the new Automation account, click Import and then import that runbook into the new Automation account. To be imported successfully, a runbook must be 1 MB or less.

Chapter 5

Azure Script Center, library, and community

This chapter explores why you will use PowerShell workflows and how they execute. Workflows are the core part of Azure Automation and is how any work gets done. The script code within the workflows runs within Azure Automation to make changes, provision resources, and begin key processes.

Windows PowerShell workflows are similar to Windows PowerShell scripts with a few additional features, but also some differences. The good news is that you don't have to develop Windows PowerShell workflow scripts from scratch. You can leverage and reuse workflows available from many resources, including the Azure Runbook Gallery, Microsoft Script Center, Microsoft MSDN and TechNet libraries, and the Azure community forums.

Windows PowerShell workflows and runbooks

Runbooks used in Azure Automation are Windows PowerShell workflows that contain Windows PowerShell code but are a bit different than pure Windows PowerShell scripts. Windows PowerShell Workflow functionality harnesses the Windows Workflow Foundation technology and was introduced in Windows PowerShell 3.0. Think of Windows PowerShell workflows as Windows PowerShell with a few additional runtime features. Differences between Windows PowerShell scripts and Windows PowerShell workflows include the following:

- Windows PowerShell workflows are compiled into Extensible Application Markup Language (XAML), which makes them a bit different from normal Windows PowerShell scripts, which are not compiled into XAML.
- Windows PowerShell Workflow uses deserialized formats for objects to implement its checkpoints. Normal Windows PowerShell scripts do not have checkpoints. For more information about checkpoints, see Chapter 2, "Runbook Management."
- Windows PowerShell cmdlets are implemented as activities in Windows PowerShell workflows. This means that only activities can execute in Windows PowerShell workflows. Powershell has implemented most of the common PowerShell cmdlets as activities. Other PowerShell cmdlets that are not converted to activities are run in as InlineScript behind the scenes.
- Some parts of Windows PowerShell scripts don't translate directly into Windows PowerShell Workflow.

When moving a Windows PowerShell script, be aware of some syntactic differences between Windows PowerShell workflows and scripts, including the following:

- Windows PowerShell workflows begin with the "workflow" keyword.
- Parameterization of the calls is different. You can't invoke methods of objects produced on a workflow because they are converted to XAML and serialized as XML-formatted objects (property bags).
- Control statements, reserved words, and some statements are not handled the same by Windows PowerShell.

A feature that helps move Windows PowerShell scripts to workflows is the Azure Automation Script Converter. Prior to its release, if you imported a Windows PowerShell script into Azure Automation that contained anything other than a single PowerShell workflow, the import would fail. With the converter, if you import a PowerShell script with no PowerShell workflows in it, not only will the import succeed, but Azure will try to convert the PowerShell script to a PowerShell workflow. This transition gives the workflow the best chance to run as a runbook with minimal manual modifications.

When you import a runbook, the Script Converter is used automatically and attempts the conversion from PowerShell script to workflow. However, there are times this conversion will fail. Also, ensure that the runbooks are tested thoroughly beforehand to maximize your chance for conversion success.

See Also *For more information about the script converter, see Introducing the Azure Automation Script Converter.*

You can make direct calls into other systems from PowerShell to do non-Azure end-to-end system processing. Don't limit your scripts just to pure Azure operations.

You can also use InlineScript in your code, as discussed in Chapter 4,"Runbook Deployment." An InlineScript activity allocates a separate, nonworkflow session to run a block of commands. When done it returns the output to the workflow. Commands in a workflow are sent to Windows Workflow Foundation for processing, whereas commands in an InlineScript block are processed by Windows.

Azure workflow execution

Understanding how the workflows execute isn't necessary to be able to use Azure Automation, but, it's handy to know what is going on architecturally when it's time to figure out what happened when problems occur.

When a runbook is invoked (manually, via a Start-AzureAutomationRunbook call from another runbook, or when a schedule is triggered), Azure Automation locates and pulls the runbook from the database. Then, it loads the Windows PowerShell workflow into the Windows PowerShell workflow engine that executes the runbooks.

The workflows run in a virtual sandboxed environment, but they do not execute inside the Azure Management Portal, which gets the initial request to start the runbook. The initial request could occur via manual invocation of the runbook, invoking the runbook via a schedule asset to which it is linked, or being called from another runbook. The runbook is passed as the input to an entirely different sandboxed execution worker server.

Think about this worker server as a special kind of virtual execution environment—a shared VM environment that to the workflow appears as if it's running in its own isolated virtual environment. This virtual execution environment is handled completely for you—provisioning, allocation, assignment, dynamically scaling as needed, deallocation of associated resources, and deprovisioning. This is a place in which the workflow is unaware of what it might be connecting to or that it is even part of Azure.

Azure Automation manages the scaling of the execution, which might include automatically scaling the resources as needed. After the process is complete, Azure Automation handles the deallocation of the resources used during the processing of the workflow. This is transparent to the workflow and to the users of the workflow. The complete virtual environment and process is handled by Azure Automation. Figure 5-1 shows a high-level overview of runbook execution.

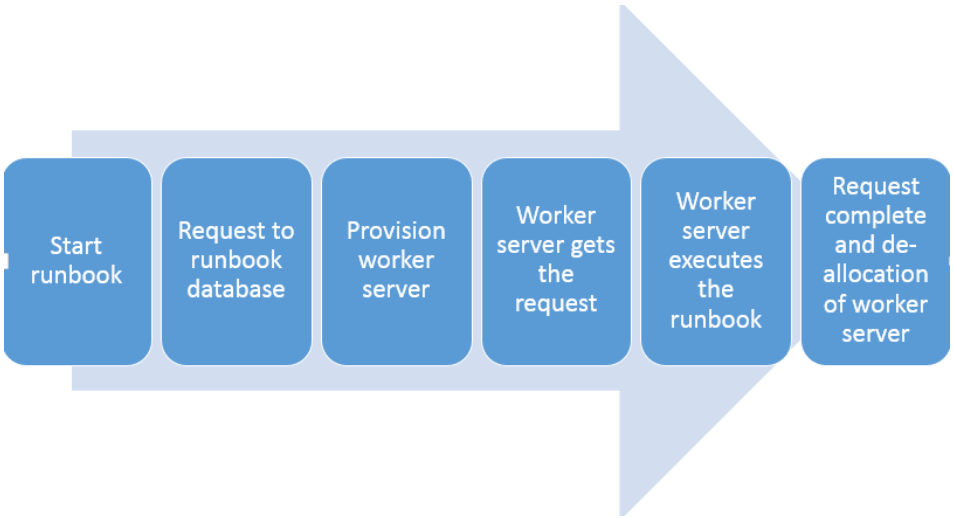


FIGURE 5-1 Runbook execution flow among invocation, runbook database, and worker server.

When communicating with Azure, the workflows need a mechanism to authenticate (just as you do when you execute Azure PowerShell scripts). The recommended process to authenticate is using Azure Active Directory, which was discussed in Chapter 3, "Assets."

During execution, you get all the failover and fault tolerance of Azure VM fault domains as part of the automation service when these sandboxes are used to run the workflows. If checkpoints are inserted in the correct junctures, runbooks are fail-safe. If a runbook crashes, when the process is restored, it picks up where it left off. At restart the workflow doesn't need to start over and redo the entire workflow. A failure in the runbook will not cause the runbook to get resumed, only suspended. If

the runbook is running and then the host running it fails, when a replacement host is allocated for that host, it will be automatically picked up from its last checkpoint. This is possible because of the checkpoint model of PowerShell Workflow.

Resources

You don't have to write all workflows from scratch. Fortunately, you can use workflows from the Azure Runbook Gallery, the Microsoft Script Center, Microsoft MSDN and TechNet libraries, and the Azure community forums.

The Azure Runbook Gallery is the simplest way to get the latest version of the most commonly used runbooks. The Runbook Gallery is displayed when you create a new runbook in the portal, select From Gallery, and then choose from the 100 or so available runbooks. Many execution categories are displayed along with the number of runbooks for that category. As of this writing, categories include monitoring, remediation, disaster recovery, VM life cycle management, change control, capacity management, compliance, and development and test environments.

In the Tutorial category, tutorials can assist you in understanding Azure Automation concepts. For instance, you can learn how to use runbook parameters, how to use Azure Active Directory to handle the authentication of your runbooks, how to use a SQL Server command in a runbook, and the different ways to invoke a child runbook. You can filter the display of runbooks to display those created by Microsoft, the Azure Automation community, Windows PowerShell Workflow, or Windows PowerShell scripts that are not workflows, but will be converted during import.

See Also *For more related information, see the "Scripting resources" section in Chapter 2.*

If you want more sample runbooks to help you learn, download them from Sample runbooks for Azure Automation. The samples contain recommended techniques that were developed with best practices in mind. This site is a great place to start because these runbooks will run as is in Azure Automation. You don't have to make any modifications.

The Azure Automation forum is a great place to not only get answers to some of your Azure Automation questions, but also to access some Windows PowerShell workflows.

The Azure Script Center provides free Windows PowerShell workflows to automate various tasks in Azure. These Windows PowerShell workflows must be transferred into runbooks to run in Azure Automation. These Azure Automation runbooks are also available in the Azure Runbook Gallery found in the Azure Management Portal. The runbooks are converted during the import process by the Azure Automation Script Converter. The Script Center also includes a link to the Azure Cmdlet Reference documentation and to the Azure GitHub site where you can contribute to the code.

In the Microsoft Script Center Repository, check out the Windows Azure category in Script resources for IT professionals.

The Official Scripting Guys Forum has great tips and information about Windows PowerShell and workflow scripting.

Azure Automation Cmdlets provides a list of every Azure Automation cmdlet and links to more information about each cmdlet. These cmdlets allow you to create and manage runbooks, schedules, and automation jobs. You can view all these cmdlets in the Azure PowerShell module, which you can download and install. After you download the PowerShell cmdlets, you can filter the output to see the Azure Automation PowerShell workflows, as shown in Figure 5-2. Use the Get-Command –Module Azure –Name "Automation" command.

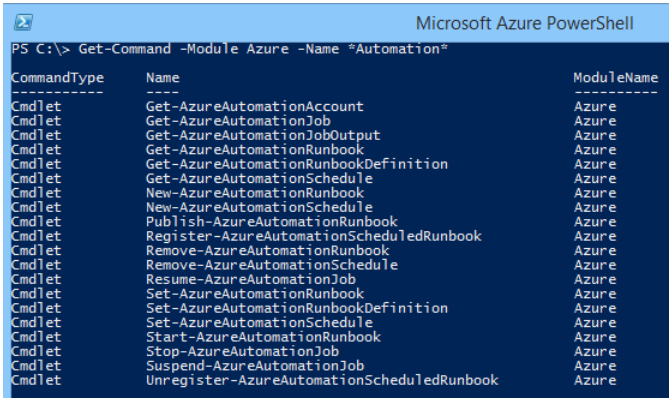


FIGURE 5-2 Windows PowerShell cmdlets for Azure Automation.

See Also *For more information about Azure PowerShell, see How to install and configure Azure PowerShell.*

Chapter 6

Best practices in using Azure Automation

Here are some recommendations and best practices that might help you in your use of Azure Automation. By implementing some of these lessons learned, you improve the chances of success for your Azure Automation project. This chapter also will give you insight into key points to help you optimize your time and application of Azure Automation. We will look at information about the development, management, and deployment of runbooks; Azure Automation account management; the use of checkpoints within runbooks; concurrent editing of runbooks; assets, importing modules, credentials and connections, schedules; and authoring runbooks outside of Azure Automation.

Runbooks

- Create your own utility runbooks for commonly used scenarios that you can reuse in many solutions. Write small and modular runbooks that accomplish discrete tasks using parameterized calls. This increases the reusability of the runbooks and allows you to build larger solutions from smaller components.
- Create runbooks that run for short periods of time with small-grained tasks.
- If you must write complex and long-running runbooks (making them harder to reuse), the #region and #endregion tags help to logically and visually make them more organized and easier to read.
- At a higher level, using tags more clearly shows the purpose for runbooks in the Azure Management Portal. A tag is an open format field that permits you to enter any values you want to allow you to better identify the purpose of and organize the runbooks. You can also list and display all runbooks that have the identical tag value. Tags should be in the form “tag1, tag2”, or “tagKey1:tagValueA, tagKey2:tagValueB”.
- Be sure to make use of commonly used runbooks that are available for download from Microsoft and the community through the Runbook Gallery.
- When designing a runbook, define parameters that can be passed into the runbook when it is called. This runbook can be invoked by an Azure Automation schedule asset or called by another runbook in a nested fashion.

- When using input parameters, you should always do the following:
    - Explicitly define if using the Mandatory attribute is mandatory . Without the Mandatory attribute, the parameter is optional.
    - Include the parameter Type and Name for your parameters.
- When assigning a default value for a parameter, Windows PowerShell views a parameter as optional regardless of whether you specify the Mandatory attribute.
- When you're assigning a name to a parameter, consider the following:
    - Use the CamelCase syntax for parameter names (e.g., MyVirtualMachine).
    - Avoid using a hyphen (-) when assigning a name for a parameter, which indicates that the parameter requires special processing that is not needed. Instead, use the underscore (_) in place of the hyphen, along with numbers and letters.
- When you're naming a runbook, consider the following:
    - In keeping with the Windows PowerShell format of verb-object, it makes sense to name the runbooks accordingly. If you have a lot of runbooks in your Automation account, after a while the names might be confusing. It's also a good idea to enter a descriptive sentence in the optional Description field to give you an additional hint about the runbook's purpose.
    - Use the list of approved verbs for Windows PowerShell commands. See Approved Verbs for Windows PowerShell Commands in the MSDN Library.
    - Divide your runbook into modular activities to take advantage of persistence of workflows. This also allows you to add checkpoints between activities.
- Thoroughly test a runbook while authoring it in Draft mode, and then promote to Publish mode only after you are sure it's ready to be used.
- Make sure a child runbook is published in Azure Automation before you try to run or publish the parent runbook that calls this child runbook.
- If you run a runbook in Draft mode, it executes just as if it were published. That is, it will provision, modify, and delete real resources from the script. So, if your runbook provisions, allocates, releases, and deletes Azure resources, realize that, as far as resource management is concerned, running a runbook in test mode is just the same as running it in published mode. In other words, it's not a what-if scenario—it's real provisioning.
- Windows PowerShell has an OutputType attribute to be used for runbooks that have output. This is a design-time attribute that shows the output types of the scripts. Include the OutputType attribute in all runbooks that have output. Use this attribute before the parameter

declarations at the start of the runbook. It will be more important to use this attribute as the refinement and expansion of Azure Automation progresses.

If an array of some type A is going to be used as output, the output type should still be just the type A, rather than of type [array of type A]. Here is an example using this attribute in a runbook.

```
 workflow My-Runbook
{
    [OutputType([string])]
    $myValue = @()
    # Code and logic to set myValue output parameter
    Write-Output $myValue
}
```

For additional best practices and runbook authoring tips, see Quick Tips and Tricks for Runbook Writing.

Concurrent editing of runbooks

- When a runbook's status is In Edit, no one should try to modify it, except the person who put the runbook in that state. Wait until the author transitions the runbook out of this state.
- If you cannot wait for that In Edit period to end and absolutely need to make changes right away, then reach out to any coadmins on that subscription who might be editing it. Receive a copy of what they have updated to incorporate it into your edits.
- Don't work on a runbook until it's in the Published state (not In Edit). This means it has no draft, and you can start working on a new version of the runbook.

Azure Automation accounts

- Create Azure Automation accounts for different units, teams, divisions, and environments.
- Note the limit of 30 accounts per subscription. If you need to create more than 30 automation accounts, look at your Enterprise Agreement subscription hierarchy and consider making the number of accounts more granular. Increasing the granularity allows you to better define and control the accounts within a subscription.
- You can use the export and import functionality to share items between Automation accounts. Alternatively, you can incorporate a source control system for the runbooks to manage importing them into Azure Automation. For more information about this approach, see Azure Automation: Integrating Runbook Source Control using Visual Studio Online.
- Use a source control system and a set of runbooks to continuously integrate and manage

importing runbooks, modules, and assets into an Azure Automation account. For information about using Azure Automation to continuously query a Visual Studio Online GIT repository to automatically publish new resources, see Azure Automation: Integrating Runbook Source Control using Visual Studio Online.

- When it's importing a script (rather than a workflow), Azure Automation converts the script from a Windows PowerShell script to a Windows PowerShell workflow by wrapping the script in InlineScript. For more information about the Azure Automation Script Converter, see Introducing the Azure Automation Script Converter.

Checkpoints

- Checkpoints are a key part of PowerShell Workflow so use them whenever applicable to ensure that certain key actions are not repeated unnecessarily in a runbook. A checkpoint can help with idempotency so that, if a runbook faults and is then resumed, critical operations are not repeated.
- For instance, insert a checkpoint immediately following the creation of a VM so that a duplicate VM isn't created if the job is suspended and then resumed.
- Use checkpoints to create Azure Automation runbooks that are more fault-tolerant and are reliable. This keeps you from redoing expensive work, and protects long-running tasks at critical points. It ensures that runbook restarts do not redo any work that should not be redone.
- The concept of "Fair Share" in Azure Automation permits any runbook that runs for over 30 minutes to be suspended so that other runbooks are allowed to run. When the suspended runbook is resumed, it will restart from the last checkpoint that was taken.
- Therefore, if a runbook is going to run more than a half an hour, insert checkpoints before any half-hour points (you can find these by testing well before publishing the runbook). If you don't add any checkpoints, the runbook might continually resume from the beginning,
- If you need to use checkpoints within InlineScript, you can do so by implementing many InlineScript blocks and placing checkpoints between them, rather than using one big InlineScript block with everything in it.
- Based on your logic, you might want to design your runbook with planned suspensions in mind. For instance, you might want to suspend a runbook until some manual or scheduled process is completed and then manually resume it.
- Here are some good locations at which to consider adding checkpoints to your workflow:
    - Following a section that does work that is not idempotent—you don't want to execute those sets of commands more than one time.

    - Following a time-intensive or resource expensive activity—to save Azure costs.
    - Within any runbook that will exceed the "Fair Share" limit.
    - Before any activity that has a large chance that it could lead to failure and suspension of a workflow. For instance, any code that calls an external system can be prone to failures.
- Here are some locations in runbooks in which checkpoints should not be used:
    - After code that the workflow should do again if it is suspended and resumed.
    - After work that is idempotent.
    - After inexpensive work that is less costly to repeat than creating a checkpoint.
    - InlineScript blocks cannot use checkpoints because the code in InlineScript is run as Windows PowerShell script, not a Windows PowerShell workflow.
- For variables that use large amounts of data but don't need to be saved, copy them to $null immediately before the checkpoint. Sending the result of an operation to $null simply causes it to be discarded. This way that data will not be checkpointed, reducing data transfer for a checkpoint.
- You can force the creation of a checkpoint right after an activity finishes that you call by including the PSPersist common parameter; for instance, Restart-VM –PSPersist $True.
- You can also include $PSPersistPreference = $True at the start of a module to cause a checkpoint to be taken after each activity that follows the preference statement. If you set this preference at the start of the runbook, a checkpoint will be made after each activity in the runbook.
- You can turn off the automatic checkpointing by including the statement $PSPersistPreference = $False (which is the runbook default), after which activities will run without automatic checkpoints.
- Persisting after every activity might not always be good for runbook performance. At a checkpoint, the workflow state is persisted to the database. At times, checkpointing might not match the business logic you'd want to use if a certain step fails. Sometimes you'll want to repeat multiple steps if a step fails, which checkpointing after every activity would not allow.
- Retrieve credentials again from the Automation asset store after a checkpoint.
- Avoid using checkpoints that require that a large amount of data be persisted to a database.

- Define and manage as many of these entities as assets so you can standardize and centralize their definition and usage rather than have to define and keep current these values embedded in many different runbooks.
- Define and assign your assets at the start of your runbook code to make the code easier to read.
- If storing any security-sensitive entities, such as the userid or passwords, don't store them in plain text. Instead, create credential, encrypted variable, or certificate assets.

Importing integration modules

- Integration modules that work with Service Management Automation (SMA) are the best choice of workflow to integrate into Azure Automation. Windows PowerShell modules do not fall into that format and can cause problems during import operations.
- The most common issue encountered during importing a module is that the zipped module package must contain a single folder within the zip file, which has the same name as the zip file. Within this folder, there needs to be at least one file with the same name as the folder, and using the extension .psd1, .psm1, or .dll. Also, the Integration Module package is a compressed file with the same name as the module and a .zip extension. It contains a single folder also with the name of the module. The Windows PowerShell module and any supporting files, including a manifest file (.psd1) if the module has one, must be contained in this folder.
- Versioning of modules is important to manage. Suppose runbook job AA is running and you import a new module during that time. If you then start a new runbook job AB with the same runbook but different version of the module, you can end up with job AA using module 1 and job AB using module 2. This might or might not be what you want, but it's important to understand how the versioning of imported modules can work.

Credentials and connections

- Instead of using management certificates as the means of authentication, use Azure AD. Using Azure AD provides a simpler credential-based means of authentication.
- Create a dedicated Azure AD account for the sole purpose of authenticating the running of the Automation scripts.
- If you have to use management certificates, you can use the same certificate for multiple subscriptions, but, although relating the same certificate to multiple subscriptions is convenient, it's like using the same password for multiple accounts. If someone gets the private portion of

the certificate, they have access to all the subscriptions. Therefore, it's recommended to have one certificate per subscription, and to avoid sharing certificates across subscriptions.

Schedules

- When you configure a schedule, the time that is initially shown in the schedule configuration dialog box is one-half hour past the current time of the desktop of the user running the Azure Management Portal. Schedule creation is in the local time of the user's browser.
- Create schedules that can be used commonly by more than one runbook whenever possible. Name them clearly and appropriately to ensure they are available to be reused; for example, DailyAt9AM.

Authoring runbooks

- Although the Azure Management Portal offers a good environment to develop scripts for runbooks, consider the option of using the Windows PowerShell Integrated Scripting Environment (ISE). Using this tool, you can create Windows PowerShell workflows on your laptop. When they're done, you can copy and paste them into Azure PowerShell workflows in the Azure Management Portal's runbook editor.
- Develop runbooks in an order where they are called starting from the lowest level child runbook and working your way backward up the call chain. The dependent (called) runbooks are created first. After the innermost runbook is created, work your way up the hierarchy and develop the parent (calling) runbooks.
- Ensure that changes made to a called workflow are included when run from the calling workflow. To do this, rebuild the child workflow and then rebuild the parent workflow to ensure the changes are included.
- If you need to make any changes to the child workflow to get it to work correctly with the parent workflow you're writing, make sure to update the child runbook as well with this new code. Publish the child runbook before you try to run the parent workflow as a runbook.
- When bringing workflow-level values inside an InlineScript block, you can reference them with an identical variable name via the $using directive. Define all $using directives at the start of an InlineScript block. For example, a variable $dog is referenced as $using:dog in an InlineScript block.
- Large objects should not be returned from, or passed to, blocks of code that are identified using InlineScripts.
- Use the Start-AzureAutomationRunbook cmdlet for parallel operations with a runbook. Avoid

using a parallel execution pattern if the tasks to be executed in parallel are internally dependent on each other.

- When logging information during runbook execution, use Write-Verbose to log human readable messages and troubleshooting information. Write-Output and the output stream should be used for data to be returned to the caller of this runbook, for example, a parent runbook.
- When logging information, do not allow confidential values to be stored as plain text.
- Log a start and stop message at the beginning and the end of each workflow or function.
- Avoid using Write-Debug or Write-Output cmdlets to log debug messages.
- Use try and catch statements where necessary within your scripts. The try block contains the guarded code that could cause the exception. When an exception is thrown, the catch statement then handles this exception. If an exception is not caught, it will cause the runbook to suspend.
- Because the runbooks will not have interactive operations with a console at runtime, avoid using any commands in the scripts that require an interactive console, or that are based on formatted output. Many cmdlets that otherwise would require user interaction expose a Force parameter to force confirmation and skip user interaction.
- Create a standard comment block at the start of each runbook that contains the following sections:
  - **.SYNOPSIS** Provide a brief overview of the purpose of the workflow.
  - **.DESCRIPTION** Provide a detailed description of the purpose of the workflow.
  - **.PARAMETER** Include the parameter name, type, and brief purpose of the parameter for each of the parameters.
  - **.EXAMPLE** Give an example calling the workflow.
  - **.NOTES** Add any information that gives you extra assistance in using the workflow.

Chapter 7

Scenarios

Up to this point, we have discussed Azure Automation key concepts along with best practices and recommendations. Now, let's apply that information in the context of some useful Azure Automation scenarios. These scenarios focus on the use of Azure infrastructure, or IaaS, because it's typically the most common use of Azure Automation. Of course, Azure Automation can also be used for the deployment and troubleshooting of your PaaS applications to automate the complete end-to-end process.

IaaS in Azure requires a deeper level of involvement on your part than does PaaS in Azure. PaaS only requires you to manage the deployment and updates of your application, its data, and the means to access that data. All the support layers—the runtime, the operating system, the VM itself, compute functionality, virtual networks, and storage—are managed for you by Azure.

Conversely, with Azure IaaS, it's your responsibility to still manage your application, data, and access. But, with IaaS, you are given a core operating system to start with as a base. It's up to you to provision and manage the operating system updates and maintenance. With IaaS, you also manage the VM on which your version of the initial operating system is running. In the Azure PaaS world, as shown in Figure 7-1, from the runtime down is managed for you so there are less manual and repetitive tasks for which you are responsible.

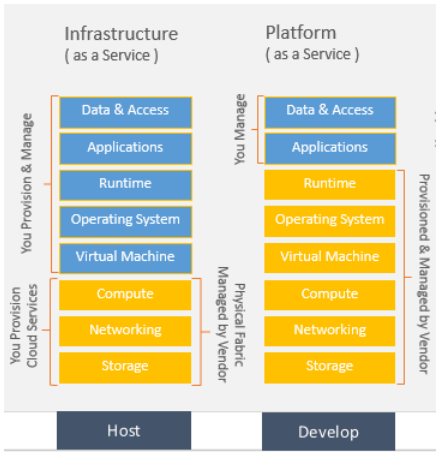


FIGURE 7-1 Layers of management responsibility for IaaS and PaaS.

Common scenarios in the real world support Azure IaaS. Therefore, those are the types of scenarios discussed in this chapter:

- Provisioning of IaaS resources.

- Maintenance and updates to Azure IaaS resources.

Both scenarios are commonly used today with customers in companies of all different sizes. Microsoft has developed scripts to support these scenarios, which we look at extensively.

Scenario: Provisioning of IaaS resources

A very common scenario is to provision a group of related Azure IaaS resources from the Azure Management Portal. This scenario happens commonly when you establish and refine a base configuration during the development and test cycles. Consistency of deployment is a major reason to want to automate a deployment. That ensures you get the same group of resources and their internal dependencies both in creation order and existence. If developers can create repeatedly the identical environment via Azure Automation that the testers will in turn create to test the code, the efficiency of the development and test process, simplicity of deployment, and higher levels of testing consistency and reproduction of problems will occur.

Another use of this scenario could be in a cold disaster recovery (DR) situation. For example, the main region housing the primary version of your deployment is down, and you need to get it up and running manually and quickly in another region. To have that provisioning script already created, and more important, loaded into an Azure Automation runbook in the backup region, gives your cold DR scenario a better chance to handle the disaster.

It is not a requirement that the subscription under which the Azure Automation runbook is installed and running be the same subscription in which the resources are being provisioned. Although it typically is the case that a runbook runs against the subscription of which it is a part, there could often be the need to use one script from a single subscription to provision resources in other subscriptions. This can be a very powerful paradigm. As long as you have the proper credentials (using Azure AD, which is preferred, or management certificates), you can connect. Using credential assets makes the authentication process a lot easier by abstracting the authentication details.

Provisioning resources

A key point in this scenario is that most of the resources need to be provisioned in a specific order to function correctly after they are created. For an enterprise Azure deployment, the following is a typical configuration of resources to provision:

1. **Azure affinity group** Ensures proximity to all the resources assigned to it, such as VMs, cloud services, storage, and virtual networks, as shown in Figure 7-2. Affinity groups are a concept that might be strategically phased out, but they are applicable as of this writing. If you decide later not to use affinity groups, you can remove them from your Azure Automation script.
2. **Azure storage accounts** Includes blob storage, table storage, Azure files, and Azure storage queues. One or more storage accounts can be associated with an application. For instance, you

could create a storage account to hold any Azure VM images, another storage account for application storage, and a separate storage account for Azure diagnostic and logging data.

3. **Azure virtual machines** Supports the Azure applications by hosting enterprise software such as SharePoint Server, Active Directory, SQL Server, custom applications such as load balancers or security monitoring software, legacy applications, or any other custom installation needed to support a solution.

4. **Azure cloud services** Hosts the Web and worker roles of an Azure application. A cloud service is where your application will be hosted and exposes a public DNS name for which the cloud service will be accessed.

Note Typically provisioning Azure VMs, cloud services, and affinity groups, is part of the process of provisioning an Azure Virtual Network. This offers benefits such as addressing, security, and easy accessibility between any VMs on a virtual network. Remember, cloud services are hosted on Azure VMs as well. However, for the sake of simplicity, this scenario doesn't include an Azure virtual network.

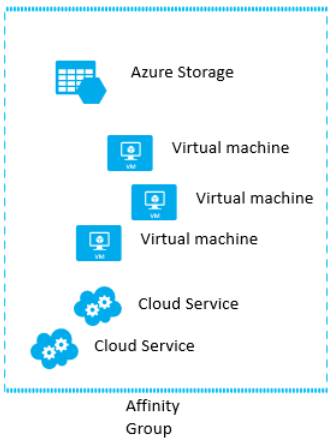


FIGURE 7-2 A typical IaaS installation using an affinity group consisting of Azure storage, cloud services, and VMs.

Authentication processing

As with any provisioning of Azure resources, the script needs to authenticate against the Azure subscription to which it should allocate and manage resources. Use Azure AD to handle script authentication. Call Azure AD through the use of a credential asset. This credential asset authenticates against a user account created just for the purpose of running Azure Automation scripts. For that specific use, you will create an Azure credential asset and enter the username and password of the account in Azure AD to give identity to the running script. The user needs to be a coadministrator on the subscription you are managing. For more information about how to set this account up, see Azure Automation: Authenticating to Azure using Azure Active Directory.

Include checkpoints strategically when their use makes a difference. This allows the runbook to be called many times due to issues that prevented it from running smoothly the first time without the fear of duplicating allocation of resources.

Using the New-AzureEnvironmentResourcesFromGallery runbook

The New-AzureEnvironmentResourcesFromGallery runbook is a detailed script that uses multiple checkpoints and Azure Automation assets. We will look at key parts of it for this scenario. (You can download it from Provision Azure Environment Resources from Gallery.)

To make a runbook as flexible for use as possible, you have two options:

- **Define as many parameters as possible** This option allows reuse of the runbook from many scripts and Azure Automation accounts by passing in parameters at runtime. Although using a lot of parameters typically takes a bit more work initially, the effort is worth it when you reuse the runbook many times. Using lots of parameters tends to make the calls to the runbook more complex and you still have to define those values somewhere before you pass them. Parameters are useful only for that specific call of that specific instance of the runbook.
- **Define variable assets** This approach is an alternative to passing a lot of parameters. Assets are global, defined once, and can be accessed from within multiple runbooks without having to be explicitly passed in. They keep the same value across runbooks (unless changed with a call to Set-AutomationVariable).

Parameters and assets have characteristics that might make one a better choice than the other in different situations. You can also use a combination of the two approaches. Deciding which option to use might come down to corporate coding standards or personal preference. (From my experience, assets make life easier once I invest the time initially to define them.) Generally, the goal is not to hard-code any information into a specific runbook so that it can be used in multiple scenarios and included in other runbooks.

Creating assets for the runbook

Now that you have a runbook (New-AzureEnvironmentResourcesFromGallery), the first step is to create five Azure Automation assets that the Get-AutomationVariable activity will use. If for some reason you choose to not use assets for those values, you can define more parameter values to be passed in when it is called. You can always hard-code values in a runbook, but that severely limits its reusability and isn't recommended.

In the following code block, those assets are accessed to use in the subsequent code.

```
#Define location of the affinity group
$AGLocation = Get-AutomationVariable -Name 'AGLocation'
#Define the filename of the image that is a .vhd file
```

```
$VMImage = Get-AutomationVariable -Name 'VMImage'
#Define the OS of the VM image
$VMImageOS = Get-AutomationVariable -Name 'VMImageOS'
#Define the username for the VM
$AdminUsername = Get-AutomationVariable -Name 'AdminUsername'
#Define the password for this username
$Password = Get-AutomationVariable -Name 'Password'
```

In the preceding example, AGLocation is the region where the affinity group is defined. VMImage is a .vhd file that already exists. In Azure Virtual Machines, images are VHD files that that have been made generic in nature by running the SYSPREP utility on them. They are not bootable images themselves unless you create a VM from that image. The image must be a part of the Azure images for the subscription to which the script is connecting.

Defining parameters and variables

There are four input parameters to this runbook. The first three parameters are required, and the last parameter is optional, as shown in the following example.

```
workflow New-AzureEnvironmentResourcesFromGallery
{
    param
    (
        [Parameter(Mandatory=$true)]
        [string]$ProjectName,
        [Parameter(Mandatory=$true)]
        [string]$VMName,
        [Parameter(Mandatory=$true)]
        [string]$VMInstanceSize,
        [Parameter(Mandatory=$false)]
        [string]$StorageAccountName
    )
```

Let's look at the parameters more closely:

- **ProjectName** Name of the deployment project for these resources. Later in the runbook, this value is used as a base to the names of some of the Azure resources that will be provisioned.
- **VMName** Name of the virtual machine created by this script.
- **VMInstanceSize** Name that identifies the VM chosen.
- **StorageAccountName** Optional; provide this if you want to create a storage account name. If you don't provide a name, a default name will be created for you. It's recommended that you provide a name that helps you track and identify the storage account. The default names are cryptic and not very useful from a human readability standpoint. The storage account name will be created, using the project name plus random lowercase characters. You will want to provide a name, however, if this isn't the first time the runbook is being invoked because there is an existing storage account.

Here is an example of the runbook being called with parameters. The project name is MyProjectName, the VM name is MyVMName, the instance size is Large, and the storage name is “mystorageaccount001n3o3m5u0u1l”.

```
New-AzureEnvironmentResources
-ProjectName "MyProjectName"
-VMName "MyVMName" `
-VMInstanceSize "Large"
-StorageAccountName "mystorageaccount001n3o3m5u0u1l"
```

The first executable code you write is to set the value of the variables, as shown in the following example. Note some variables are set based on parameters being passed in and some are set by calling the Get-AutomationVariable cmdlet to obtain the value of the variable asset.

```
     # Set Azure Affinity Group Variables
    $AGName = $ProjectName
    $AGLocation = Get-AutomationVariable -Name 'AGLocation'
    $AGLocationDesc = "Affinity group for {0} VMs"  -f $ProjectName
    $AGLabel = "{0} {1}" -f $AGLocation,$ProjectName
    # Set Azure Cloud Service Variables
    $CloudServiceName = $ProjectName
    $CloudServiceDesc = "Service for {0} VMs" -f $ProjectName
    $CloudServiceLabel = "{0} VMs" -f $ProjectName
    # Set Azure Storage Account Variables
    if (!$StorageAccountName) {
        $StorageAccountName = $ProjectName.ToLower()
        $rand = New-Object System.Random
        $RandomPadCount = 23 - $StorageAccountName.Length
        foreach ($r in 1..$RandomPadCount)
{ if ($r%2 -eq 1) { $StorageAccountName += [char]$rand.Next(97,122) }
else { $StorageAccountName += [char]$rand.Next(48,57) } }
    }
    $StorageAccountDesc = "Storage account for {0} VMs" -f $ProjectName
    $StorageAccountLabel = "{0} Storage" -f $ProjectName
    # Set Azure VM Image Variables
    $VMImage = Get-AutomationVariable -Name 'VMImage'
    $VMImageOS = Get-AutomationVariable -Name 'VMImageOS'
    #Set Azure VM Variables
    $ServiceName = $ProjectName
    $AdminUsername = Get-AutomationVariable -Name 'AdminUsername'
    $Password = Get-AutomationVariable -Name 'Password'
    $Windows = $true
    $WaitForBoot = $true
```

Configuring authentication

Within the runbook, access the Azure Automation credential asset. The runbook code gets the credentials from Azure Automation using the Azure Automation credential asset and then uses it to authenticate when it connects to Azure. The following example creates the user and credential asset before we invoke the runbook.

```
# Get the credential to use for Authentication to Azure and Azure Subscription Name
$Cred = Get-AutomationPSCredential -Name 'Azure AD Automation Account'
$AzureSubscriptionName = Get-AutomationVariable -Name 'Primary Azure Subscription'
# Connect to Azure and Select Azure Subscription
$AzureAccount = Add-AzureAccount -Credential $Cred
$AzureSubscription = Select-AzureSubscription
          -SubscriptionName $AzureSubscriptionName
           $WaitForBoot = $true
```

The Get-AutomationPSCredential command gets the credential asset from the subscription named Azure AD Automation Account, and that subscription name is put into $AzureSubscriptionName. The credential asset is used in the call to Add-AzureAccount, which is then passed to select the subscription. At the end of this code, you have connected to that Azure account and selected an Azure subscription to which you will soon provision the resources.

Processing details

After authentication is complete, it's time to create the affinity group, the cloud service, the Azure storage account, and VM. Be aware of the following key points about these processes:

- A checkpoint is taken after provisioning each resource to ensure that, if failure occurs, any resources will not be provisioned again.
- After each checkpoint, there is a reconnection to the Azure subscription to process the following provisioning commands. This reconnection is required in the case where the runbook job is persisted to storage and then resumed on a different automation host from the last checkpoint.
- The creation of an Azure storage account is often not instantaneous. Thus, a delay of 1 minute is introduced to wait until the storage account exists. Then, the Azure storage account is used when first verifying, and then creating the Azure VM in the following code blocks.
- Before entering one of the provision blocks, a check is made to ensure the previous allocation was successful. If not, error messages are displayed and written under the category of ResourceUnavailable.
- At the end of the provisioning, a check for the completion or failure of the processing occurs and either a completed notice is output or error messages are displayed and written.
- The Write-Verbose cmdlet writes text to the verbose message stream in Windows PowerShell. Typically, the verbose message stream is used to deliver information about command processing that is used for debugging a command.
- The Write-Error cmdlet declares a nonterminating error. By default, errors are sent in the error stream to the host program to be displayed, along with output.

We won't go into every line of code related to the creation of the affinity group, the cloud service,

the Azure storage account, and VM. Following are key excerpts from the runbook for these creation processes:

```
# Create/Verify Azure Affinity Group
    if ($AzureAccount.Id -eq $AzureSubscription.Account) {
        Write-Verbose "Connection to Azure Established - Specified Azure Environment Resource
Creation In Progress..."
        $AzureAffinityGroup = Get-AzureAffinityGroup -Name $AGName -ErrorAction SilentlyContinue
        if(!$AzureAffinityGroup)
        {
            $AzureAffinityGroup = New-AzureAffinityGroup -Location $AGLocation -Name $AGName
-Description $AGLocationDesc -Label $AGLabel
            $VerboseMessage = "{0} for {1} {2} (OperationId: {3})" -f
$AzureAffinityGroup.OperationDescription,$AGName,$AzureAffinityGroup.OperationStatus,$AzureAffin
ityGroup.OperationId
        }
      else
      { $VerboseMessage = "Azure Affinity Group {0}: Verified" -f $AzureAffinityGroup.Name    }

       Write-Verbose $VerboseMessage



    } else {
        $ErrorMessage = "Azure Connection to $AzureSubscription could not be Verified."
        Write-Error $ErrorMessage -Category ResourceUnavailable
        throw $ErrorMessage
     }
    # Checkpoint after Azure Affinity Group Creation
    Checkpoint-Workflow
   # (Re)Connect to Azure and (Re)Select Azure Subscription
    $AzureAccount = Add-AzureAccount -Credential $Cred

###############################################################################################

    # Create/Verify Azure Cloud Service
    if ($AzureAffinityGroup.OperationStatus -eq "Succeeded" -or $AzureAffinityGroup.Name -eq
$AGName) {
         $AzureCloudService = Get-AzureService -ServiceName $CloudServiceName -ErrorAction
SilentlyContinue
       if(!$AzureCloudService) {
            $AzureCloudService = New-AzureService -AffinityGroup $AGName -ServiceName
$CloudServiceName -Description $CloudServiceDesc -Label $CloudServiceLabel
            $VerboseMessage = "{0} for {1} {2} (OperationId: {3})" -f
$AzureCloudService.OperationDescription,$CloudServiceName,$AzureCloudService.OperationStatus,$Az
ureCloudService.OperationId

        } else { $VerboseMessage = "Azure Cloud Serivce {0}: Verified" -f
$AzureCloudService.ServiceName }
        Write-Verbose $VerboseMessage
    }
    else {
        $ErrorMessage = "Azure Affinity Group Creation Failed OR Could Not Be Verified for: $AGName"
        Write-Error $ErrorMessage -Category ResourceUnavailable
```

```
        throw $ErrorMessage
     }
    # Checkpoint after Azure Cloud Service Creation
    Checkpoint-Workflow
    # (Re)Connect to Azure and (Re)Select Azure Subscription
    $AzureAccount = Add-AzureAccount -Credential $Cred
    $AzureSubscription = Select-AzureSubscription -SubscriptionName $AzureSubscriptionName
    ##############################################################################################
#      Create/Verify Azure Storage Account
    if ($AzureCloudService.OperationStatus -eq "Succeeded" -or $AzureCloudService.ServiceName -eq
$CloudServiceName) {
        $AzureStorageAccount = Get-AzureStorageAccount -StorageAccountName $StorageAccountName
-ErrorAction SilentlyContinue
        if(!$AzureStorageAccount) {
            $AzureStorageAccount = New-AzureStorageAccount -AffinityGroup $AGName
-StorageAccountName $StorageAccountName -Description $StorageAccountDesc -Label
$StorageAccountLabel
            $VerboseMessage = "{0} for {1} {2} (OperationId: {3})" -f
$AzureStorageAccount.OperationDescription,$StorageAccountName,$AzureStorageAccount.OperationStat
us,$AzureStorageAccount.OperationId
        }
      else { $VerboseMessage = "Azure Storage Account {0}: Verified" -f
$AzureStorageAccount.StorageAccountName }
        Write-Verbose $VerboseMessage
    }
    else {
        $ErrorMessage = "Azure Cloud Service Creation Failed OR Could Not Be Verified for:
$CloudServiceName"
        Write-Error $ErrorMessage -Category ResourceUnavailable
        throw $ErrorMessage
     }
    # Checkpoint after Azure Storage Account Creation
    Checkpoint-Workflow
    # (Re)Connect to Azure and (Re)Select Azure Subscription
    $AzureAccount = Add-AzureAccount -Credential $Cred
    $AzureSubscription = Select-AzureSubscription -SubscriptionName $AzureSubscriptionName
   ##############################################################################################

    # Sleep for 60 seconds to ensure Storage Account is fully created
    Start-Sleep -Seconds 60
    # Set CurrentStorageAccount for the Azure Subscription
    Set-AzureSubscription -SubscriptionName $AzureSubscriptionName -CurrentStorageAccount
$StorageAccountName
    ##############################################################################################

    # Verify Azure VM Image
    $AzureVMImage = Get-AzureVMImage -ImageName $VMImage -ErrorAction SilentlyContinue
    if($AzureVMImage) { $VerboseMessage = "Azure VM Image {0}: Verified" -f $AzureVMImage.ImageName
}
    else {
        $ErrorMessage = "Azure VM Image Could Not Be Verified for: $VMImage"
        Write-Error $ErrorMessage -Category ResourceUnavailable
        throw $ErrorMessage
    }
```

```
    Write-Verbose $VerboseMessage
    # Checkpoint after Azure VM Creation
    Checkpoint-Workflow
    # (Re)Connect to Azure and (Re)Select Azure Subscription
    $AzureAccount = Add-AzureAccount -Credential $Cred
    $AzureSubscription = Select-AzureSubscription -SubscriptionName $AzureSubscriptionName
   ###############################################################################################

    # Create Azure VM
    if ($AzureVMImage.ImageName -eq $VMImage) {
        $AzureVM = Get-AzureVM -Name $VMName -ServiceName $ServiceName -ErrorAction SilentlyContinue
        if(!$AzureVM -and $Windows) {
            $AzureVM = New-AzureQuickVM -AdminUsername $AdminUsername -ImageName $VMImage -Password
$Password `
                -ServiceName $ServiceName -Windows:$Windows -InstanceSize $VMInstanceSize -Name
$VMName -WaitForBoot:$WaitForBoot
            $VerboseMessage = "{0} for {1} {2} (OperationId: {3})" -f
$AzureVM.OperationDescription,$VMName,$AzureVM.OperationStatus,$AzureVM.OperationId
        }
       else { $VerboseMessage = "Azure VM {0}: Verified" -f $AzureVM.InstanceName }
        Write-Verbose $VerboseMessage
    }
    else {
        $ErrorMessage = "Azure VM Image Creation Failed OR Could Not Be Verified for: $VMImage"
        Write-Error $ErrorMessage -Category ResourceUnavailable
        $ErrorMessage = "Azure VM Not Created: $VMName"
        Write-Error $ErrorMessage -Category NotImplemented
        throw $ErrorMessage
     }
    ###############################################################################################
    if ($AzureVM.OperationStatus -eq "Succeeded" -or $AzureVM.InstanceName -eq $VMName) {
        $CompletedNote = "All Steps Completed - All Specified Azure Environment Resources Created."
        Write-Verbose $CompletedNote
        Write-Output $CompletedNote
    }
    else {
        $ErrorMessage = "Azure VM Creation Failed OR Could Not Be Verified for: $VMName"
        Write-Error $ErrorMessage -Category ResourceUnavailable
        $ErrorMessage = "Not Complete - One or more Specified Azure Environment Resources was NOT
Created."
        Write-Error $ErrorMessage -Category NotImplemented
        throw $ErrorMessage
     }
}
```

This scenario has discussed one of the most common Azure Automation scenarios of consistently automating the provisioning of Azure resources. Automating the provisioning allows you to manage any dependencies, and the outcome is the same each time the automaton process occurs.

Scenario: Maintaining and updating Azure IaaS resources

One of the most common uses for scripting in any IT environment is to automate updates to computers. Client computers, server machines, database servers, and application servers all need updates. It doesn't matter if the configuration is in an on-premises or cloud environment. Updates are tasks that work best transparently in the background with as little user intervention as possible. An example is the Automatic Updates administration feature in Windows that you can configure to automate the installation of updates.

Within Azure IaaS VMs, you have full control over what you install on your VM beyond the base operating system install. But with that freedom comes the responsibility of managing the update process and deciding when and what is updated. You will want to update VM software in a way that does not affect all your users at once via a graduated rollout that is staggered over multiple VMs.

For example, suppose you create a set of identically configured SharePoint Server IaaS VMs and put them in an availability set. By having the servers in an availability set, you are telling Azure that any of the servers in that set can replace any other server as needed should that VM become unavailable. This unavailability could be due to either unexpected downtimes or planned update periods.

In this scenario we show you how to manage the automated update process for Azure VMs.

Summary of upgrade process

The Upgrade VM demonstration script is a good match for this scenario. You can download it from the Microsoft Script Center Repository at Manage Windows Updates on an Azure VM using Azure Automation.

Within this demonstration script are a group of related runbooks that form a solution to guide you through the process of managing VM updates. More specifically, the runbooks help update files on a VM within the Update-AzureVM runbook. The runbooks must be called in the following order to ensure everything works correctly. Recall that any child runbook must be published prior to any parent runbooks that invoke it.

- **Connect-Azure runbook** Use this runbook to set up a connection to an Azure subscription. Input parameter is an Azure connection asset for the subscription ID and the name of the certificate asset that holds the management certificate. This certificate is copied into the local machine certificate store. This runbook has been deprecated in favor of using the OrgID credential to connect to Azure. If you're using Azure AD, the certificate is not used for the authentication process.
- **Connect-AzureVM runbook** *This runbook sets up a connection to an Azure VM, which must have* been *enabled ahead of time* with *the Windows Remote Management Service. During its processing, it* invokes the Connect-Azure runbook to set up a connection to an Azure subscription while importing a certificate asset. This certificate allows the remote Windows

PowerShell calls to authenticate. Input parameters include the Azure connection asset, the name of the cloud service in which the VM exists, and the actual VM to which the connection will be made.

- This runbook begins a remote Azure PowerShell session with an Azure VM. This runbook requires that Azure PowerShell already be configured on your local machine. Run Get-AzurePublishSettingsFile and Import-AzurePublishSettingsFile to configure Azure PowerShell once it has been installed.
- **Copy-ItemToAzureVM runbook** This runbook copies a local file from the Automation host running the job to a location in an Azure VM. Input parameters include the Azure connection asset, the name of the cloud service, the name of the VM to which the file will be copied, the PowerShell credential asset that contains the userid and password to log onto the VM, the local file path to the source file on the Automation host, and the remote destination path on the VM to where the file will be copied.
- **Copy-FileFromAzureStorageToAzureVM runbook This runbook copies a file from its location in blob storage to a file location on the VM. Here you will select an Azure subscription and a storage account with a specific blob container.**
- **Install-ModuleOnAzureVM runbook** Use this runbook to install the module on a VM if it is not already present. The file is first unzipped to the modules directory and then the copy is unzipped to the destination location. Input parameters include the Azure connection asset, the PowerShell credential asset, the storage account and container name, the module blob name, the VM name, and the module name.

Figure 7-3 gives you a hierarchical view of the calling order of these runbooks. Note that Connect-AzureVM is called twice in Figure 7-3 but is not repeated in the preceding description.

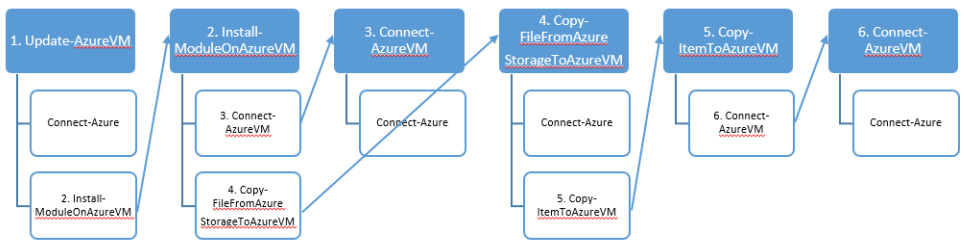


FIGURE 7-3 Hierarchical view of calling order for all children runbooks of Update-AzureVM.

Using the Update-AzureVM runbook

The Update-AzureVM runbook ties together the runbooks described earlier and manages the entire update process from a high level. This runbook enumerates all the VMs in an Azure subscription and

manages their update process.

To help with the Windows Update process on a VM, Update-AzureVM needs the PSWindowsUpdate PowerShell zip file downloaded and extracted onto the VM. There are cmdlets inside the PSWindowsUpdate runbook for installing updates, such as to validate existence of a module that is to be updated. Once present, you can invoke a cmdlet from the PSWindowsUpdate module on the Azure VM to obtain a list of available updates from Windows Update.

To get PSWindowsUpdate on the VM to be updated, follow these steps:

1. Download the PSWindowUpdate.zip file to your local drive, or blob storage, from https://gallery.technet.microsoft.com/scriptcenter/2d191bcd-3308-4edd-9de2-88dff796b0bc.
2. If it is downloaded locally, use an Azure storage tool to move the zip file to an Azure blob storage account in your subscription.
3. Go to the blob storage location and download the module (in its zipped form) to the Azure VM.
4. After the copy is complete, unzip the module into the PSPath on the Azure VM.

Before you invoke the Update-AzureVM runbook, the following operations must occur, in no specific order:

- If you use a management certificate (as opposed to authenticating using Azure AD), it must be first loaded to the Azure subscription. However, it is recommended to use Azure AD instead of the management subscription.
- All the runbooks must be imported into Azure Automation, and then the children (called) runbooks should be published before the parent (calling) runbooks.
- You must create the following assets:
    - A PowerShell credential asset that contains the UserID and Password for the remote session into all of your VMs. As a best practice, use the same logon credentials for as many VMs as possible to simplify the process and make it easier to remember and track.
    - A connection asset that contains the subscription ID.
    - The certificate asset mapped to your management certificate.

Let's look at some code from the parent Update-AzureVM runbook and some key parts of the child runbooks that it calls. The first part of the runbook initializes some key variables that you will need to set ahead of time, as they apply to your subscription to ensure the runbook works correctly. Input parameters include the Azure connection asset, the PowerShell credential asset, the storage account and container name, the module blob name, the VM name, and the module name. The runbook then connects to Azure using the connection asset in the call to the Connect-Azure runbook. The connection asset must be defined ahead of time within the subscription inside which this runbook is invoked.

The main work begins in the call to Install-ModuleOnAzureVM. The code sets the value of the predefined Azure connection asset into the $AzureConnectionName variable. It then takes the name of the PowerShell credential asset that abstracts the VM's login credentials and stores that in the $CredentialAssetNameWithAccessToAllVMs variable. That credential is then passed into the Get-AutomationPSCredential activity to obtain the credential asset to use with all VMs. It then performs a few more self-explanatory operations, such as setting storage account, container, and blob name to the specific values.

```
workflow Update-AzureVM {
    $AzureConnectionName = "joeAzure"
    $CredentialAssetNameWithAccessToAllVMs = "joeAzureVMCred"
    $CredentialWithAccessToAllVMs = Get-AutomationPSCredential -Name
$CredentialAssetNameWithAccessToAllVMs
    $WUModuleStorageAccountName = "joestorage123"
    $WUModuleContainerName = "psmodules"
    $WUModuleBlobName = "PSWindowsUpdate.zip"
    $ModuleName = "PSWindowsUpdate"

Connect-Azure -AzureConnectionName $AzureConnectionName
    Write-Verbose "Getting all VMs in $AzureConnectionName"
       -AutomationAccountName $AutomationAccountName `
                   -ErrorAction "Stop"

` # Get all VMs in subscription
    $VMs = InlineScript {
        Select-AzureSubscription -SubscriptionName $Using:AzureConnectionName
        Get-AzureVM
    }
    # Install PSWindowsUpdate module on each VM if it is not installed already
    foreach($VM in $VMs) {
        Write-Verbose ("Installing $ModuleName module on " + $VM.Name + " if it is not installed
already")
        Install-ModuleOnAzureVM `
            -AzureConnectionName $AzureConnectionName `
            -CredentialAssetNameWithAccessToVM $CredentialAssetNameWithAccessToAllVMs `
            -ModuleStorageAccountName $WUModuleStorageAccountName `
            -ModuleContainerName $WUModuleContainerName `
            -ModuleBlobName $WUModuleBlobName `
            -VM $VM `
            -ModuleName $ModuleName
    }
```

After the variables are set, the script iterates through each of the VMs in a subscription. From there the PSWindowUpdate.zip file (stored in the WUModuleBlobName variable) is copied onto any of the VMs that do not have the file installed yet. This is done by calling Install-ModuleOnAzureVM. Initially, this runbook is placed into the modules directory as an unzipped file before it is copied to a specific blob ($WUModuleBlobName) with a certain container ($WUModuleContainerName) for a specific $WUModule destination storage account.

```
# Install latest Windows Update updates onto each VM
```

```
    foreach($VM in $VMs) {
        $ServiceName = $VM.ServiceName
        $VMName = $VM.Name
        $Uri = Connect-AzureVM -AzureConnectionName $AzureConnectionName -ServiceName $ServiceName
-VMName $VMName
        Write-Verbose "Installing latest Windows Update updates on $VMName"

InlineScript {
            Invoke-Command -ConnectionUri $Using:Uri -Credential
$Using:CredentialWithAccessToAllVMs -ScriptBlock {
                $Updates = Get-WUList -WindowsUpdate | Select-Object Title, KB, Size, MoreInfoUrls,
Categories

                foreach($Update in $Updates) {
                    $Output = @{
                        "KB" = $Update.KB
                        "Size" = $Update.Size
                        "Category1" = ($Update.Categories | Select-Object
Description).Description[0]
                        "Category2" = ($Update.Categories | Select-Object
Description).Description[1]
                    }
                    "Title: " + $Update.Title
                    $Output
                    "More info at: " + $Update.MoreInfoUrls[0]
                    "------------------------------------------------"
                }
            }
        }
```

For each VM, its cloud service name and the name of the VM itself is obtained. Connect-AzureVM is called, passing in the AzureConnectionName, the service name, and the VM name. It returns the URI of the VM.

After the URI is obtained, the Invoke-Command is called to execute a script on a remote computer that is reached and connected to using the $Uri parameter, and authenticated with the Credential parameter. This action passes the code found within the –ScriptBlock { } directive to each VM to execute on those machines. The enclosed script code gets a list of available updates and stores them in the $Updates variable via a call to Get-WUList. When called, this gets a list of the available updates that meet the criteria listed in the logic within the foreach loop. It returns the KB size of the update, descriptive information, title, and various category information.

Supporting runbooks

It is worth taking a look at some key code snippets from the supporting runbooks previously mentioned that are used in the VM updating process. We can start from the top, where the call to the Install-ModuleOnAzureVM runbook is made.

Install-ModuleOnAzureVM runbook

The child Install-ModuleOnAzureVM runbook is called directly from its parent, the Update-AzureVM runbook. To make the connection to the VM, it takes as input parameters the name of an Azure connection asset and the name of an Azure PowerShell credential asset. For the storage location it will use, it takes in the name of the storage account, the container that will store the module to be installed, and the Azure blob storage name. It also takes the name of the module to be installed and the VM name for which it is to be installed. The path to the module and the module zip file are defined, authentication occurs, and a $Uri value is returned after the script connects to the Azure VM via a call to Connect-AzureVM. When the connection is made to the VM, the Invoke-Command is called with the URI and the credentials to log on to the VM. It passes in a script block that calls Test-Path with an argument list to ensure the path is valid.

After the runbook determines that the VM currently does not have the module installed, the runbook Copy-FileFromAzureStorageToAzureVM is invoked to copy the module file that is stored in Azure blob storage onto the VM, as shown in the following example.

```
# Install  $PathToPlaceModule = "C:\Windows\System32\WindowsPowerShell\v1.0\Modules\$ModuleName"
    $PathToPlaceModuleZip = "C:\$ModuleName.zip"
    $CredentialWithAccessToVM = Get-AutomationPSCredential -Name
$CredentialAssetNameWithAccessToVM
    $Uri = Connect-AzureVM -AzureConnectionName $AzureConnectionName -ServiceName $VM.ServiceName
-VMName $VM.Name
   Write-Verbose ("Checking if " + $VM.Name + " contains module $ModuleName")
    $HasModule = InlineScript {
        Invoke-Command -ConnectionUri $Using:Uri -Credential $Using:CredentialWithAccessToVM
-ScriptBlock {
            Test-Path $args[0]
        } -ArgumentList $Using:PathToPlaceModule
    }
    # Install module on VM if it doesn't have module already
    if(!$HasModule) {
        Write-Verbose ($VM.Name + " does not contain module $ModuleName")
        Write-Verbose ("Copying $ModuleBlobName to " + $VM.Name)
        Copy-FileFromAzureStorageToAzureVM `
            -AzureConnectionName $AzureConnectionName `
            -CredentialAssetNameWithAccessToVM $CredentialAssetNameWithAccessToVM `
            -StorageAccountName $ModuleStorageAccountName `
            -ContainerName $ModuleContainerName `
            -BlobName $ModuleBlobName `
            -PathToPlaceFile $PathToPlaceModuleZip `
            -VM $VM
            }
        }
```

The call to Copy-FileFromAzureStorageToVM passes in an Azure connection and credential asset to connect to the VM. The storage account name, the container name, the blob name, and the path to the zip file are input to this call. Along with the VM being passed in, the file path of the module is also input to locate the zip file on that VM's file directory.

The following script code moves the zip file to the destination path on the VM for which the update can potentially occur.

```
InlineScript {
            Invoke-Command -ConnectionUri $Using:Uri -Credential $Using:CredentialWithAccessToVM
-ScriptBlock {
                $DestinationPath = $args[0]
                $ZipFilePath = $args[1]
                # Unzip the module to the modules directory
                $Shell = New-Object -ComObject Shell.Application
                $ZipShell = $Shell.NameSpace($ZipFilePath)
                $ZipItems = $ZipShell.items()
                New-Item -ItemType Directory -Path $DestinationPath | Out-Null
                $DestinationShell = $Shell.Namespace($DestinationPath)

                $DestinationShell.copyhere($ZipItems)
                # Clean up
                Remove-Item $ZipFilePath
            } -ArgumentList $Using:PathToPlaceModule, $Using:PathToPlaceModuleZip
        }
```

Within the InlineScript, the Azure PowerShell code again calls Invoke-Command, using the URI of the connection and the credential asset. Within the script block, the destination path and the file path of the zipped module are specified. New-Object creates a custom shell COM object using the Shell.Application property. The ZipFilePath and the items are copied to working variables, and then the New-Item cmdlet is invoked to create a new file folder. The original file is then copied to that new location and removed from the file system.

Copy-FileFromAzureStorageToAzureVM runbook

The child Copy-FileFromAzureStorageToAzureVM runbook is called from its parent, the Install-ModuleOnAzureV runbook, which in turn was called from the top-level runbook node, Update-AzureVM.

The purpose of this runbook is to copy a file into a folder on the VM from blob storage. Input parameters include the Azure connection and credential assets, the storage account, container, and blob name. Additionally, it includes the path to where the copy will be made and the name of the VM.

```
    $TempFileLocation = "C:\$BlobName"
    Connect-Azure -AzureConnectionName $AzureConnectionName
    Write-Verbose "Downloading $BlobName from Azure Blob Storage to $TempFileLocation"
    InlineScript {
        Select-AzureSubscription -SubscriptionName $Using:AzureConnectionName
        $StorageAccount = (Get-AzureStorageAccount -StorageAccountName
$Using:StorageAccountName).Label

        Set-AzureSubscription `

            -SubscriptionName $Using:AzureConnectionName `
            -CurrentStorageAccount $StorageAccount
        $blob =
```

```
            Get-AzureStorageBlobContent `
                -Blob $Using:BlobName `
                -Container $Using:ContainerName `
                -Destination $Using:TempFileLocation `
                -Force
    }
    Write-Verbose ("Copying $BlobName to $PathToPlaceFile on " + $VM.Name)
    Copy-ItemToAzureVM `
        -AzureConnectionName $AzureConnectionName `
        -ServiceName $VM.ServiceName `
        -VMName $VM.Name `
        -VMCredentialName $CredentialAssetNameWithAccessToVM `
        -LocalPath $TempFileLocation `
}
```

Some of the operations here, such as connecting to Azure and selecting a subscription, were discussed earlier, so we move past those here. The Get-AzureStorageBlobContent cmdlet is called to obtain a specific blob, passing in the blob and container path, and the destination location. Using the Force parameter means it will overwrite an existing file without confirmation. The output of this cmdlet is an Azure storage container.

The actual copying of the module that is stored in the blob into the desired file location is done in the Copy-ItemToAzureVM runbook. When this is invoked, it passes in the Azure connection name along with the VM's cloud service and its actual VM name. The credential is passed in to authenticate and the localpath of the temporary file location is specified.

Copy-ItemToAzureVM runbook

Input parameters are the usual suspects, plus the local and remote path parameters. Within the InlineScript block, Azure PowerShell code performs the copy operation.

```
   # Store the file contents on the Azure VM
    InlineScript {
        $ConfigurationName = "HighDataLimits"
        # Enable large data to be sent
        Invoke-Command -ScriptBlock {
            $ConfigurationName = $args[0]
            $Session = Get-PSSessionConfiguration -Name $ConfigurationName
            if(!$Session) {
                Write-Verbose "Large data sending is not allowed. Creating PSSessionConfiguration
$ConfigurationName"
                Register-PSSessionConfiguration -Name $ConfigurationName
-MaximumReceivedDataSizePerCommandMB 500 -MaximumReceivedObjectSizeMB 500 -Force | Out-Null
            }
        } -ArgumentList $ConfigurationName -ConnectionUri $Using:Uri -Credential $Using:Credential
-ErrorAction SilentlyContinue


        # Get the file contents locally
        $Content = Get-Content -Path $Using:LocalPath -Encoding Byte
        Write-Verbose ("Retrieved local content from $Using:LocalPath")
        Invoke-Command -ScriptBlock {
```

```
            $args[0] | Set-Content -Path $args[1] -Encoding Byte
        } -ArgumentList $Content, $Using:RemotePath -ConnectionUri $Using:Uri -Credential
$Using:Credential -ConfigurationName $ConfigurationName
        Write-Verbose ("Wrote content from $Using:LocalPath to $Using:VMName at $Using:RemotePath")
    }
}
```

To store the file contents to a specific location on the Azure VM, the familiar Invoke-Command is called with another custom PowerShell ScriptBlock to accomplish this transfer. The Get-PSSessionConfiguration cmdlet is typically called only to do additional operations when managing PowerShell session configurations. The properties of a PowerShell session configuration object vary with the options set for the session configuration and the values of those options. Here, it's called to obtain a session configuration identified by a specific name for the VM that has been registered. If no session configuration is obtained, it calls Register-PSSessionConfiguration to create and register a new PowerShell session configuration. When registering a new PowerShell session configuration, Register-PSSessionConfiguration sets both the minimum amount of data that can be received and the maximum to 500 MB in size. It suppresses all user prompts and restarts the service without prompting to make the change effective.

The second block Invoke-Command contains script with a call to Set-Content. This cmdlet writes the content of the new file to the destination found in the RemotePath argument.

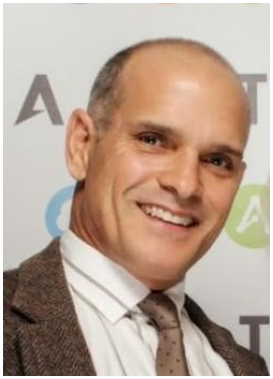
Some final thoughts

The Update-AzureVM runbook exists mainly for the sake of teaching people how to update Azure VMs. In its current state, it might not be quite ready to use as a production runbook because of a few key issues. Chiefly, the script does not actually install any updates; it just lists them out. The script would need to be modified to manage the updates. For instance, the Update-AzureVM runbook can be run on a schedule so that updates could be applied during maintenance. When actually updating VMs, the process most likely involves more steps than just installing the updates on the VMs. For example, you need to take the VM out of the load balancer while it is being updated, or you might want to put it into a maintenance mode if there is a monitoring program attached to it.

Additionally, due to the Fair Share execution limitation policy, checkpoints should be used to help roll back processing to the last checkpoint spot if the script is interrupted, either intentionally or unexpectedly. If the VM needs to reboot to install updates, the runbook must gracefully handle the termination of the connection to the VM from the VM side. It then must attempt occasionally to reconnect so that you can reconnect after the VM is back online.

Finally, the Update VM demo set of runbooks could use a few optional improvements. As mentioned previously, authenticating using Azure AD instead of management certificates is preferable. Also, don't store a module on the path that is intended for system modules only on the VM. You could also acquire the module from the Azure Automation module path instead of from blob storage. The script also targets all the VMs in a subscription, which might not be what you want in a production environment.

About the author



Mike McKeown is a Microsoft Azure MVP who is employed as a Principal Cloud Architect with Aditi Technologies. He spent almost two decades with Microsoft in various roles and has spent over 25 years working within various IT roles. This has given Mike a very unique breadth, as well as depth, of the IT environment from the view of development, management, infrastructure, sales, and the customer. He has experience in the cloud around both the Infrastructure and Platform as a Service solution models. His passion is to help stakeholders or customers define their business/system requirements, and then apply cloud architecture patterns and best practices to meet those goals.

Mike writes white papers for MSDN, blogs about Azure on his blog at www.michaelmckeown.com, develops Azure video training content for Pluralsight, and is a speaker at both regional and national conferences. You can follow his experiences with Azure on Twitter at @nwoekcm.

Mike lives in Charlotte, NC with his wife Tami and five kids Kyle, Brittany, Adrianna, Michael Jr, and Sean. He plays the drums, is active in his church, and loves to work out regularly.

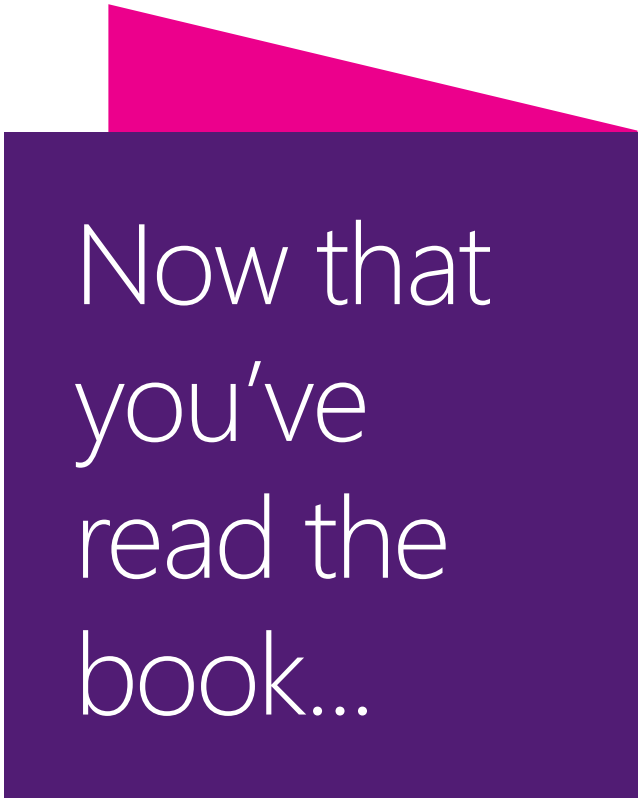
From technical overviews to drilldowns on special topics, get *free* ebooks from Microsoft Press at:

www.microsoftvirtualacademy.com/ebooks

Download your free ebooks in PDF, EPUB, and/or Mobi for Kindle formats.

Look for other great resources at Microsoft Virtual Academy, where you can learn new skills and help advance your career with free Microsoft training delivered by experts.

Microsoft Press



Tell us what you think!

Was it useful?
Did it teach you what you wanted to learn?
Was there room for improvement?

Let us know at http://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press,
and we read every one of your responses. Thanks in advance!

Microsoft