Microsoft System Center

Designing Orchestrator Runbooks

David Ziembicki · Aaron Cushner · Andreas Rynes
Mitch Tulloch, Series Editor

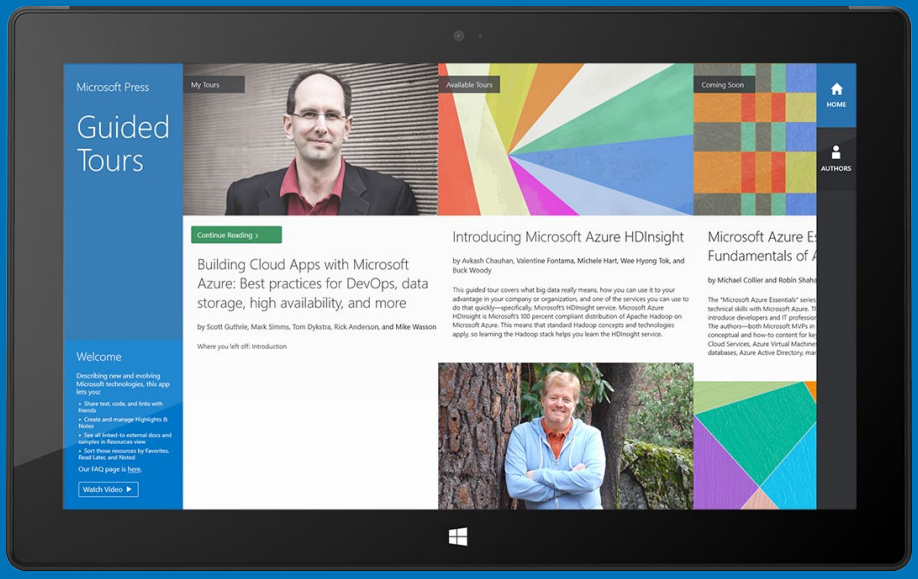
Visit us today at

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

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

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

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

Library of Congress Control Number: 2013948711
ISBN: 978-0-7356-8298-6

Printed and bound in the United States of America.

First Printing

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

Acquisitions Editor: Anne Hamilton
Developmental Editor: Karen Szall
Project Editor: Karen Szall
Editorial Production: Diane Kohnen, S4Carlisle Publishing Services
Cover Illustration: Twist Creative • Seattle
Cover Design: Microsoft Press Brand Team

Contents

Introduction

Welcome to *Microsoft System Center: Designing Orchestrator Runbooks*. We believe that orchestration and automation are becoming increasingly important in IT organizations of all sizes and across all infrastructure types ranging from on-premises to cloud-based. Orchestration and automation can help reduce the cost of IT while improving consistency and quality of IT service delivery. Like any powerful technology. however, it can be both used and abused.

Our objective with this book is to provide a framework for runbook design and IT process automation to help you get the most out of System Center Orchestrator 2012 and to help you utilize Orchestrator in concert with the rest of the System Center for an enterprise-wide and systematic approach to process automation. We will provide detailed guidance for creating what we call "modular automation" where small, focused pieces of automation are progressively built into larger and more complex solutions. We detail the concept of an automation library, where over time enterprises build a progressively larger library of interoperable runbooks and components. Finally, we will cover advanced scenarios and design patterns for topics like error handling and logging, state management, and parallelism. But before we dive into the details, we'll begin by setting the stage with a quick overview of System Center 2012 Orchestrator and deployment scenarios.

About the companion content

The companion content for this book consists of Windows PowerShell scripts and other code samples. It can be downloaded from the following page:

http://aka.ms/SCrunbook/files

Errata & book support

We've made every effort to ensure the accuracy of this content and its companion content. Any errors that have been reported since this content was published are listed on our Microsoft Press site at oreilly.com:

http://aka.ms/SCrunbook/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

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 book at:

http://aka.ms/tellpress

The survey is short, and we read every one of your comments and ideas. 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

Introducing System Center 2012

Microsoft System Center 2012 is Microsoft's solution for cloud and datacenter management as well client device management and security. From its origins nearly 20 years ago as primarily a desktop management solution, System Center has evolved into a leading enterprise management solution across physical, virtual, and cloud infrastructure including devices, applications, and services.

System Center 2012 is comprised of a suite of components, each focused on part of the infrastructure management lifecycle such as provisioning, monitoring, backup, and disaster recovery. From an IT process automation perspective, the System Center components are the "arms and legs" of the automation capability, which act on end systems while System Center Orchestrator, and the runbooks created within it, are the "brains" of the automation, controlling the order and flow of activities and responding to events during the automated process.

In Figure 1-1, each of the focus areas of System Center are listed as well as the System Center components that deliver those capabilities.

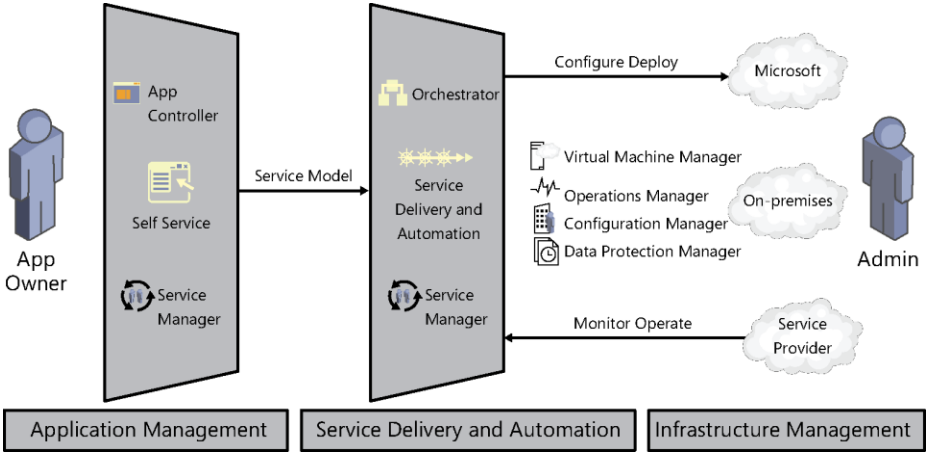


FIGURE 1-1 The System Center 2012 suite.

We will briefly introduce all of the System Center components in this chapter from the perspective of their use and value in IT process automation. For detailed information on each component, please refer to Microsoft TechNet.

System Center Virtual Machine Manager

System Center Virtual Machine Manager (VMM) is Microsoft's solution for heterogeneous datacenter virtualization and management. VMM assists in establishing the datacenter foundation from bare-metal deployment of Hyper-V host servers to creating Hyper-V clusters to updating Hyper-V infrastructures. VMM can integrate with and manage a variety of storage and network infrastructure components. For heterogeneous environments, VMM can manage both VMware and Citrix XenServer environments in addition to Hyper-V. With the virtualization infrastructure established, VMM enables the deployment and management of both virtual machines and service templates, which are multiple virtual machine configurations enabling the deployment of complex or multitier applications.

Using all of the above capabilities, VMM is a key component in establishing private cloud infrastructure as a service (IaaS). From an IT process automation perspective, VMM, with its ability to manage compute, network, storage, and virtual resources, backed by hundreds of Windows PowerShell cmdlets, will be one of the most important System Center components utilized by many automated processes.

System Center Operations Manager

System Center Operations Manager is the monitoring and alerting component of System Center across physical, virtual, and applications/services. In recent versions, Operations Manager has expanded to support monitoring Linux systems as well as network and storage resources. Operations Manager continues to be extended by a wide range of partners through management packs. From an IT process automation perspective, Operations Manager is frequently the sources of alerts and events which are the triggers for process automation or Orchestrator runbooks. Examples include a performance alert triggering a runbook to scale out a web farm, or a hardware fault triggering a runbook to place a Hyper-V host into maintenance mode.

System Center Service Manager

System Center Service Manager deals with the ITIL-based service management and human workflow side of process automation. Until Service Manager was released, System Center had long been missing a centralized configuration management database (CMDB) consolidating all of the discovered inventory and configuration information from the entire System Center suite—from devices inventoried by Configuration Manager to users from Active Directory to virtual resources from VMM. Service Manager implements ITIL-based service management processes, such as Incident and Change Management, by enabling a human workflow engine for topics such as help desk ticketing, approvals, and routing. Service Manager includes a customizable self-service portal and extensible service catalog.

Service Manager functions as a key component of IT process automation by serving as the "front end" through the self-service portal and service catalog. As the library of process automation grows over time, each process can be added to the service catalog enabling administrators or users to initiate a request or automated process through the Service Manager self-service portal. Examples might include a request to provision a virtual machine or development environment, a request to reboot a server, and so on.

The Service Manager CMDB can also be a critical component in process automation as the primary source for device and configuration information such as relationships between a user and their requests or between a virtual machine and the cloud it is associated to.

System Center Data Protection Manager

System Center Data Protection Manager (DPM) provides backup and disaster recovery functionality for Microsoft applications and services. From backing up data or Microsoft applications such as SharePoint or SQL Server to recovery services in an alternate site, DPM is designed to provide a cost-efficient solution for backup and disaster recovery.

Frequently, backup and disaster recovery is a very complex activity requiring a large number of actions to be performed across a variety of IT infrastructures in order to successfully restore service after an event. Backup and disaster recovery are ideal candidates for automation as they require a strict sequence of events, must be tested periodically, and must be executed as quickly and consistently as possible.

System Center Configuration Manager

System Center Configuration Manager provides client device and application management. From deployment of desktops and devices to managing application delivery and virtualization, Configuration Manager is a key component of an enterprise management infrastructure.

Given that desktop and client devices frequently outnumber servers in most environments, automation becomes critical given the larger number of endpoints and the frequency of activities such as updating or software deployment.

In many cases, client device management is one of the more costly areas of IT due to large numbers, the need to involve the help desk, and the need for administrator intervention. When selecting processes to automate, typically the most repetitive or error prone have the largest ROI, and in many cases those are client device or user related. The combination of Orchestrator and Configuration Manager can in many cases take some of the most frequently occurring needs and automate those, such as deploying software in certain conditions, or automating the assessment and upgrade of desktop devices.

System Center Orchestrator

System Center Orchestrator 2012 will be covered in detail in the next chapter. Orchestrator adds a workflow engine, authoring experience, and execution infrastructure for runbooks, which are instances of IT process automation. While each System Center component discussed in this chapter includes automation of certain processes, they typically deal with only part of the management lifecycle. For processes which need to span the lifecycle, or which need to integrate with multiple System Center or third-party systems, Orchestrator is essential.

CHAPTER 2

System Center Orchestrator

Microsoft System Center 2012 Orchestrator is the primary IT process automation component of the System Center suite. With Orchestrator, IT pros and/or infrastructure developers can create repeatable automation of repetitive or error prone IT processes in the form of Orchestrator runbooks. Orchestrator runbooks are conceptually similar to scripts in that they perform some set of operations in a repeatable manner. Where they differ is that Orchestrator runbooks can be created by IT pros without as deep of a background in scripting or programming initially, but can also include script components in more advanced scenarios. This chapter provides a brief overview of the features and capabilities offered by Orchestrator from a runbook author's perspective.

Runbook Designer

The Runbook Designer is the heart of the runbook authoring experience in Orchestrator and along with integration packs (described shortly) the two differentiating factors from traditional scripting. The Orchestrator Runbook Designer shown in Figure 2-1 is a graphical interface for authoring runbooks. This Microsoft Visio-like interface presents a much more approachable authoring experience for both basic and advanced automation than traditional scripting does.

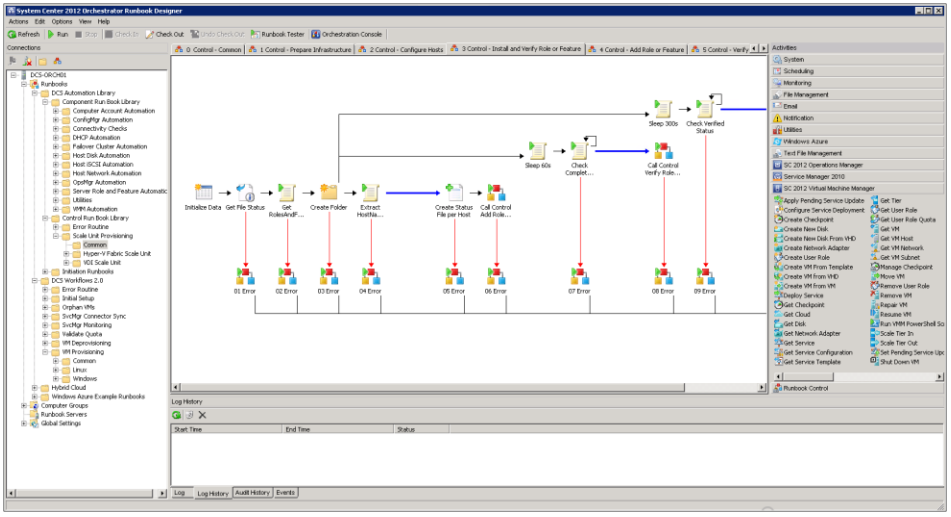


FIGURE 2-1 The Orchestrator Runbook Designer.

The Runbook Designer provides five major areas of functionality for the runbook author to utilize when designing Orchestrator solutions.

Connections and runbook hierarchy

This pane presents a hierarchy of folders and runbooks enabling you to organize and apply permissions to manage access and the ability to edit. Right-clicking a runbook or folder provides a number of different options such as editing permissions or importing/exporting the runbook or folder. The import and export functionality is critical to systematic runbook design as it enables you (manually) to establish backups and version control of runbooks (described in subsequent chapters) and to move runbooks between environments (such as dev/test/production). This pane also includes the Runbook Servers tab listing all of the runbook servers in the deployed Orchestrator solution and Global Settings where variables and counters are implemented. Best practices for utilizing this hierarchy, permissions, and variables will be detailed in later chapters.

Menu and command bar

The Menu and Command bar contains a number of important elements. One is the Connections menu where you configure connections required by any imported management packs to other management systems. These connections require credentials with appropriate permissions on the target management systems. The Command bar includes buttons for checking in and checking out runbooks for edit. The check in / check out process is simply a lock on the selected runbook so that it can only be edited by the person who checked it out. This functionality does not include any version control, so once a runbook has been changed and checked in, you cannot revert to a previous version (unless you manually exported the previous version and re-import).

Runbook design surface

The runbook design surface is where the actual visual editing of runbooks is performed. The runbook folder structure is where you select a runbook to edit. Once selected, the runbook will be presented in the design surface.

Activity list

The activity list contains all of the built-in activities and the activities from any deployed integration pack. You can drag activities from the list and drop them onto the design surface for use in runbooks. Double-clicking an activity on the design surface opens the activity for editing its parameters.

Logging

The logging pane includes information about the status of the selected runbook. Logging includes currently running instances of the runbook as well as a history of completed executions of the runbook. The amount of logging data retained and automatic periodic

purging of the logging data (recommended for good console performance) is configured by right-clicking the Orchestrator server at the top of the runbook folder hierarchy and selecting Log Purge.

The process of designing, testing, and deploying runbooks will be covered in significant depth in subsequent chapters. In general, it consists of mapping out the process to be automated, streamlining it logically to be as efficient as possible (that is, the fewest steps possible, most loosely coupled approach, and so on) then determining what systems must be orchestrated, determining if integration packs for all systems are available, and then finally laying out the activities and process flow in the Runbook Designer.

While initially appearing simple, the Runbook Designer is quite powerful. The designer enables branching, looping, and parallelism all with conditional logic.

Integration packs

As mentioned, integration packs (IPs) are the primary method of extending Orchestrator. Orchestrator ships with a set of foundation objects and activities for basic tasks such as file management, email integration, and other basic activities that are non-system specific. Microsoft then provides a number of additional IPs for the System Center suite and select third-party systems such as VMware. Beyond that, there is a large and growing ecosystem of integration packs from other hardware and software makers which further extend Orchestrator.

Integration packs typically consist of a set of activities specific to the target management system. As an example, the System Center Virtual Machine Manager (VMM) integration pack includes activities such as starting and stopping virtual machines, creating a virtual machine, and so on. To use an integration pack it must be imported and deployed to all of your Orchestrator runbook servers and a connection established between Orchestrator and the target management system (VMM in this example) using a service account with adequate (typically administrator) permissions on the target management system.

Runbook Tester

The Orchestrator Runbook Tester is another key feature that assists in the runbook design process by providing the ability to test runbook functionality prior to implementation of your runbooks in a production environment. The Runbook Tester is effectively a debugger for Orchestrator runbooks. From the Runbook Designer you can navigate to the runbook you want to test and then select the Runbook Tester button. This will launch the Runbook Tester and open the current runbook in the tester. The Runbook Tester, like a script debugger, lets you set breakpoints in your runbook execution which pause the runbook at that point, enabling you to verify any expected results to that point, check the value of any variables or

logs created so far, and so on. You can then resume the runbook or execute it step by step to continue to evaluate the results. This capability is important in the runbook authoring process, particularly for large or complex runbooks.

There are some limitations to the Runbook Tester such as only being able to test an individual runbook and not larger or more complex scenarios where multiple runbooks are nested or sequentially executed. It is also important to note, and we will review this later, that the Runbook Tester executes runbooks under the context of the logged-on user, not under the context of the Orchestrator service account which is used for executing runbooks in production scenarios. In many cases, a runbook may work in the tester but not in production due to differences in permissions between the service account and the runbook author in the tester.

Figure 2-2 illustrates a runbook in the tester with a breakpoint set at the second step. In the left pane, details of the selected activity are displayed. In the lower middle pane, the completed steps of the runbook are displayed.

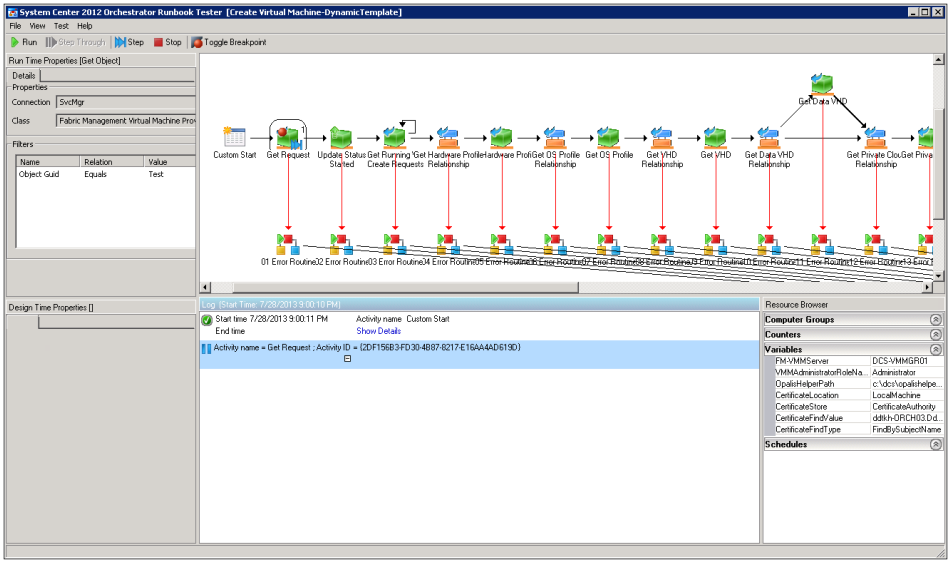


FIGURE 2-2 The Orchestrator Runbook Tester.

The right pane shows the value of variables on the data bus. Each completed step in the lower middle pane can be selected and the log history details of the step can be viewed. This enables analysis of the results of those steps to verify results. Once verification is completed, the remainder of the runbook can be executed by manually stepping through each step or by letting it run to the end.

Orchestration console

The Orchestration console is a web-based user interface for initiating and monitoring runbook execution. From this console you can see all running runbooks and their status. Figure 2-3 illustrates the console and the list of available runbooks.

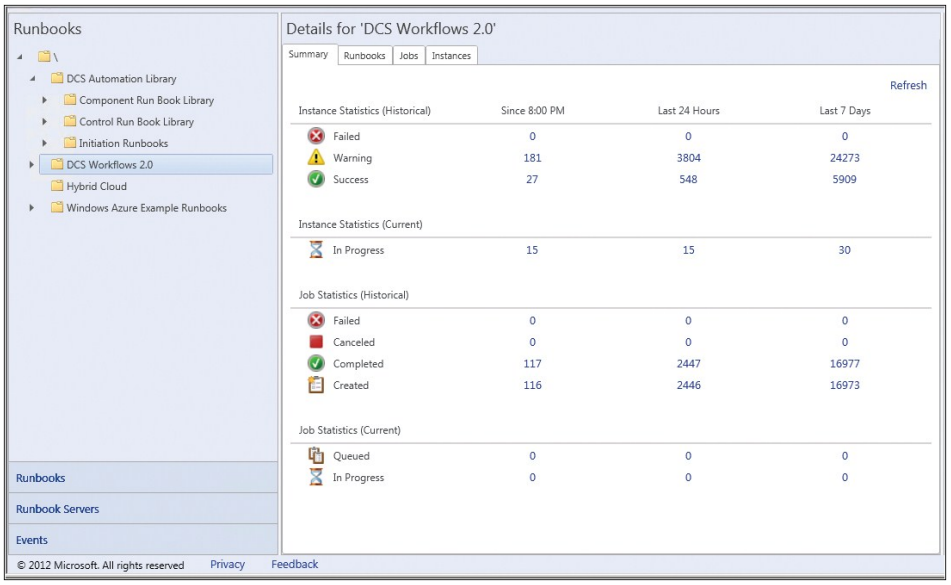


FIGURE 2-3 The Orchestration console.

The console can also be used to initiate runbooks. If the selected runbook is configured to require input parameters, the Orchestration Console prompts you to enter values for those parameters. In Figure 2-4, a runbook has been started and is prompting for a single parameter. Once the parameter is entered, the runbook will be executed.

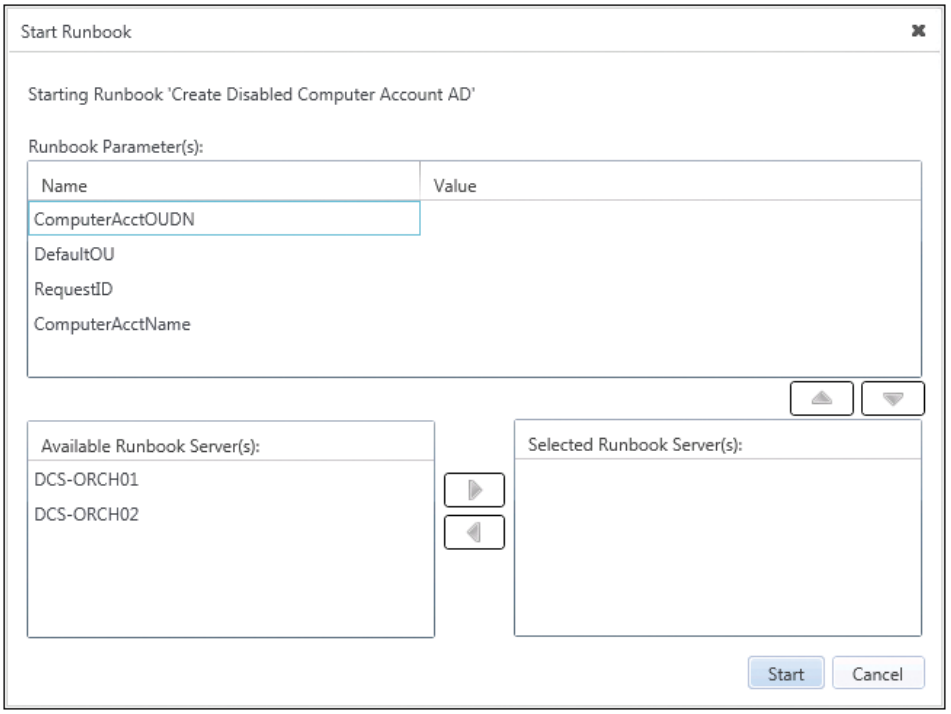


FIGURE 2-4 The Orchestration console runbook parameter dialog box.

Orchestrator Integration Toolkit

Having discussed the concept of integration packs which extend Orchestrator to connect to other management systems and the large ecosystem of third parties creating integration packs, the question often remains: "What do I do if there is no management pack for the system I want to orchestrate?" This question may arise if neither the foundation objects included with Orchestrator nor any third-party integration packs enable the connectivity and activities you need to orchestrate another system. Examples may include a large line of business applications you'd like to orchestrate as part of some process automation. Fortunately, Orchestrator has an answer in the Orchestrator Integration Toolkit (OIT). The OIT enables you to write your own integration packs for systems or applications that don't have an integration pack but support some form of automation such as web services application programming interfaces (APIs). Table 2-1 lists the components included in the OIT.

TABLE 2-1 The Components Included in the Orchestrator Integration Toolkit

COMPONENT	DESCRIPTION
Command-Line Activity Wizard	A utility that allows users to define activities that contain commands that run via Windows command shell, PowerShell, or SSH, and package them into an assembly (.DLL) that can be used with the .NET IP or packaged into a new integration pack.
Integration Pack Wizard	A utility designed to package Orchestrator-compatible activity assemblies and dependent files into a deployable Integration Pack file.
Integration Toolkit .NET IP	An integration pack for running .NET-based Orchestrator-compatible activity assemblies directly. Contains the Invoke .NET and Monitor .NET activities.
Integration Toolkit SDK Library	A set of files that are used by developers utilizing the System Center 2012 - Orchestrator SDK to write custom activities.

This book does not cover creating custom integration packs as it is beyond the scope of the typical IT Pro usage of Orchestrator but it is important to be aware that this capability exists and as usage of Orchestrator grows in your organization, there will likely be instances where creating an integration pack for your mission-critical applications or services may be warranted and the OIT is easily utilized by developers to create them. For more information about integration pack development and for other options for building custom solutions with Orchestrator, refer to the "System Center 2012 Integration Guide," which you can find on Microsoft TechNet at *http://social.technet.microsoft.com/wiki/contents/articles/13188.system-center-2012-integration-guide.aspx*.

CHAPTER 3

Orchestrator architecture and deployment

This chapter covers the system architecture of System Center 2012 Orchestrator, the various components that are part of the product and how to deploy it in an environment. It discusses the different options that are available to deploy Orchestrator, the scenarios and requirements that you have to think about first before getting started, and details the high availability concepts for each of the components. We also describe the data bus, a key concept of Orchestrator that will help you build your runbooks and allows you to pass data between activities and runbooks without writingcode.

Architecture

This section introduces the system components of Orchestrator, their purpose, and how they are going to work together.

System architecture

The system architecture of Orchestrator contains different components—some of them are required, some of them are optional depending on the requirements and usage. This topic will provide an overview of Orchestrator and describe the components and its capabilities.

Runbook

A runbook is the visual representation of your workflow. You will use a graphical interface, the Runbook Designer, to design and build the IT process automation and create those runbooks. A runbook consists of activities and colorful icons, each performing a specific task (for example, create an alert in System Center Operations Manager or run a Windows PowerShell script) that are linked together. The links between activities provide options to do filtering and give you the ability to use multiple paths and conditional logic such as if-then-else.

Management server

The Orchestrator Management server is the layer that is responsible for the communication between the designer of runbooks (Runbook Designer) and the database that holds all required data such as runbook definitions, instances of runbooks, etc.

Runbook server

A runbook server controls the running instances of a runbook; it directly communicates with the database without using the management server. The Runbook server is a Windows service that can be deployed using the Orchestrator Deployment Manager. You can have multiple runbook servers in your environment to distribute load and increase the total capacity of your automation architecture. It is also used to provide high availability.

Orchestrator database

The Orchestrator database utilizes Microsoft SQL Server and contains all runbook definitions, various logging information, as well as configuration data for Orchestrator. It is a single database that holds all the information required for the entire Orchestrator deployment.

Runbook Designer

The Runbook Designer shown in Figure 3-1 is a tool that is part of Orchestrator which you can use to build, edit, and manage your runbooks. It consists of a design pane and a folder structure to organize your runbooks as well as a toolbox that provide the different activities that you will use to build your workflows.

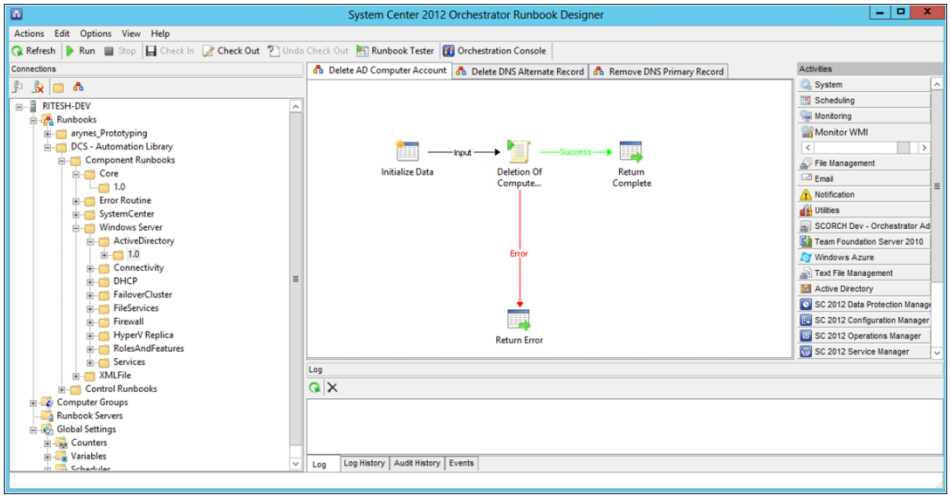


FIGURE 3-1 The System Center 2012 Orchestrator Runbook Designer.

Although the left side of the window shows a similar interface to File Explorer, all runbooks, regardless of how you manage and group them, are stored in the Orchestrator database and not in the file system.

Runbook Tester

The Runbook Tester is another tool that comes with Orchestrator that will help you to test your runbooks at run-time. It will provide you with detailed information and data that are part of

your runbook and is very helpful in troubleshooting problems that you might facing during implementation.

Orchestration console

The Orchestration console lets you manage your runbooks in real-time. You are able to start and stop them and gather real-time status information about current running instances of your runbooks. The tool is a web browser interface that can be accessed remotely.

Orchestrator web service

The Orchestrator web service interface is a Representational State Transfer (REST)-based service that enables custom applications to connect to Orchestrator. It will help to integrate the runbook infrastructure to other portals and tools and give you the option to manage your runbooks such as starting or stopping them or retrieving information about runbook operations.

The Orchestration console uses the Orchestrator web service to communicate with Orchestrator.

Deployment Manager

If you want to use integration packs (IPs) that extend the capabilities of Orchestrator, such as integration of other platforms and tools (for example, HP Service Manager), you need to register and deploy them into your Orchestrator environment. This can be done using the Deployment Manager, a tool that is used to deploy runbook servers and Runbook Designers. The tool is shown in Figure 3-2.

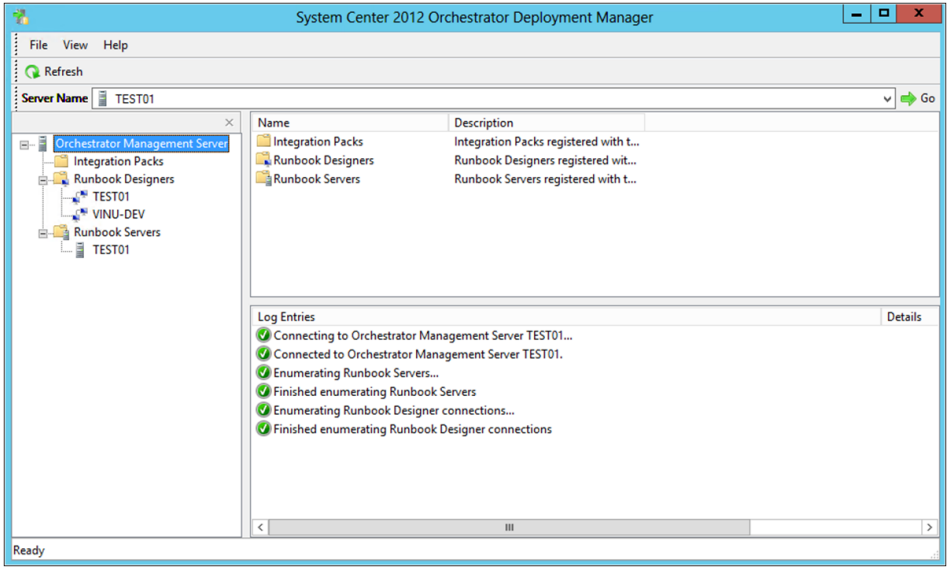


FIGURE 3-2 The System Center 2012 Orchestrator Deployment Manager.

Data bus

Automation is all about moving data between components, platforms, and tools. In most cases you take input data, work with that data, pass it to different systems, get some results back, and at the end of the runbook it's common to return data such as the result state and, optionally, error messages.

The data bus is a key concept of Orchestrator that will help you to do exactly that—pass that data around without writing any code.

To achieve that, the data bususes another core concept called Published Data. Every step (called activities in Orchestrator) within a runbook will publish data to the data bus automatically. You can also easily add additional data to the data bus. Every subsequent activity in the runbook can access that data bus, subscribe to it, and use the Published Data from there again without any coding.

This makes it very easy to work with data across different vendors and tools to provide automated processes, and it will also decrease the time of implementing those processes significantly.

Architectural diagram

The architectural diagram shown in Figure 3-3 illustrates each of the Orchestrator features and the communication between each component.

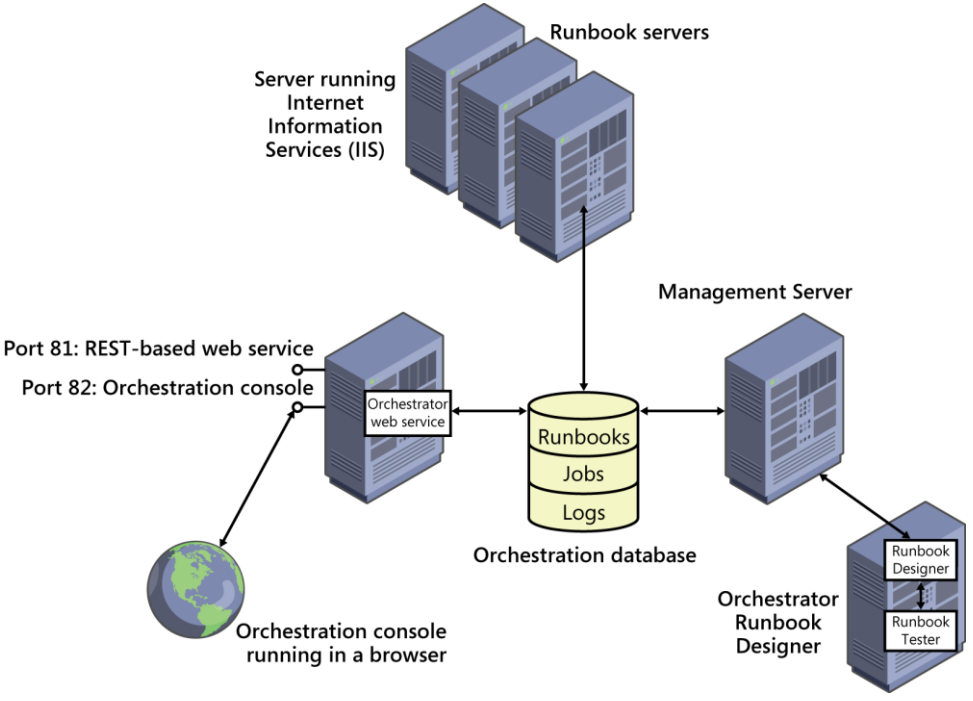


FIGURE 3-3 The Orchestrator system architecture.

The center of every Orchestrator environment is the orchestration database running on Microsoft SQL Server. This is really one of the key critical components of an automation environment using Orchestrator. The database will contain all your runbooks, configuration settings, and logs.

You also need at least one (or more) runbook servers that will communicate directly with the database to retrieve runbooks, to run and store information about the jobs created from the runbooks. So this is the second critical key component that must be available to run your automation processes.

The web services layer also communicates directly with the orchestration database and provides a web browser connection for the Orchestration console. The unavailability of the web service won't impact your automation solution as long as you don't rely on that web service within your implemented runbooks.

The management server is required as a communication layer between the Runbook Designer and the orchestration database, but it is not required for actually running the runbooks, which means a management server that is not available doesn't impact the runbooks from doing their work, but it will impact building new runbooks or changing existing runbooks.

High availability considerations

If you're going to implement automation it will always be the heart of each datacenter and reliability and availability are top most concerns. So this section describes—based on the different components of the product—where high availability is important and how to achieve that.

Management server

This component is actually limited to one single management server, so within every Orchestrator environment there can only be one management server. From an availability perspective there is no requirement that the Orchestrator management server must be up and running or available. Even if the management server is down the runbook servers or runbooks will still run and work.

If the management server is not available, there is an impact on the Runbook Designer. The Runbook Designer cannot be used to publish runbooks or start, monitor, or stop runbooks. Instead, the Orchestration console or the web services must be used to start, monitor, and stop runbooks.

Orchestration database

An Orchestrator 2012 database is hosted on Microsoft SQL Server 2008 or SQL Server 2012 with the basic database engine features, so there is no need for additional features. The database is one of two components that are critical to the entire environment. So it's key to

have an available database for Orchestrator. From a high availability perspective we would recommend you to use Failover Clustering and at least two nodes for SQL Server.

Orchestrator web service

The Orchestrator web service must be installed on a server that is running IIS. If you use the Orchestrator web service in your runbooks then high availability might be required and important to your environment. In general, it is used to start, monitor, or stop runbooks, but it does not have to be available for runbook servers or runbooks to function. For high availability we recommend you install the Orchestrator web service on multiple IIS servers and use a load balancer to both provide additional capacity as well as continuous web service support in case of a failure of one of your web servers.

Orchestration console

The Orchestration console must be installed on a server that is running IIS. The console can be used to start, monitor, or stop runbooks and similar to the web service it is not required to be available for the runbook servers or runbooks to function.

In case you want high availability, we would recommend that you install Orchestration console on more than one IIS server configured for load balancing. This will provide you high availability, but also additional capacity for requests that will use the console.

Runbook servers

Runbook servers are required for runbooks to function. If there isn't a runbook server available, no runbooks will be executed. Therefore high availability is required and important for your Orchestrator environment. But note that runbook servers are not designed to run on a cluster node. To achieve high availability in that case, we recommend deploying at least two runbook servers—depending on your environment and workload it might be required or recommended to have even more than two runbook servers. If the primary runbook server for a runbook is unavailable, the runbook can run on another server.

There are actually two mechanisms in place that are combined, the "spill over" mechanism helps spread the load of runbook instances across the existing runbook servers. So the more runbook servers that are available, the more concurrent instances can run in the same environment. The second mechanism will check for the health of a runbook server using a heartbeat signal. As soon as the runbook server is unhealthy (after 3 missed heartbeats, or 45 seconds) the assignment of a runbook instance will be changed to another runbook server. This makes sure that an unavailable runbook server will not bring down the automation environment and other runbook servers can take over.

You might come across the "Runbook Server Monitor" service in your environment. It monitor the health of the runbook servers and generates Orchestrator platform events should a runbook server show signs of problems. It does not provide any other high availability function in the Orchestrator environment.

Runbooks

A runbook doesn't provide high availability; the runbook server must be high available to make sure your workflows will continue to run in case of a failure. By default, each runbook server is configured to run a maximum of 50 runbooks concurrently. You can change that number based on your experience, resource requirements of your runbooks, and the available resources of your runbook servers.

Consider the resource requirements of the runbooks on a particular server and based on that you can change the default value of 50 to another number by using the runbook server Runbook Throttling tool. In cases where the server has a number of runbooks with high resource requirements, you might want to run fewer concurrent runbooks on the runbook server. In cases of more simple runbooks with small or minimal requirements, you might consider increasing the number of concurrent runbooks on the runbook server.

For additional high availability you can deploy multiple runbook servers, which is described earlier in the "Runbook server section.

Orchestrator 2012 architecture patterns

This section discusses some typical design patterns for Orchestrator 2012 deployments with different requirements and environments. It also gives some examples where you might want to use it, but it doesn't offer general recommendations as there are always requirements that might change the architecture.

The following architecture patterns will be discussed:

- Single-server Orchestrator 2012 infrastructure
- High availability Orchestrator 2012 infrastructure
- High availability and multisite Orchestrator 2012 infrastructure

Single-server Orchestrator 2012 infrastructure

This single-server Orchestrator 2012 infrastructure is a basic deployment where all components are deployed either on a single physical machine or on a single virtual machine. This single machine hosts the management server, Orchestration database, runbook server, Runbook Designer, and Orchestration console.

This design pattern shown in Figure 3-4 is fully functional, but does not provide any high availability or redundancy. In case of a failure, Orchestrator will not be able to process runbooks anymore. So you shouldn't deploy this pattern in a production environment, but it can be used for proof of concepts, demos, or development environments.

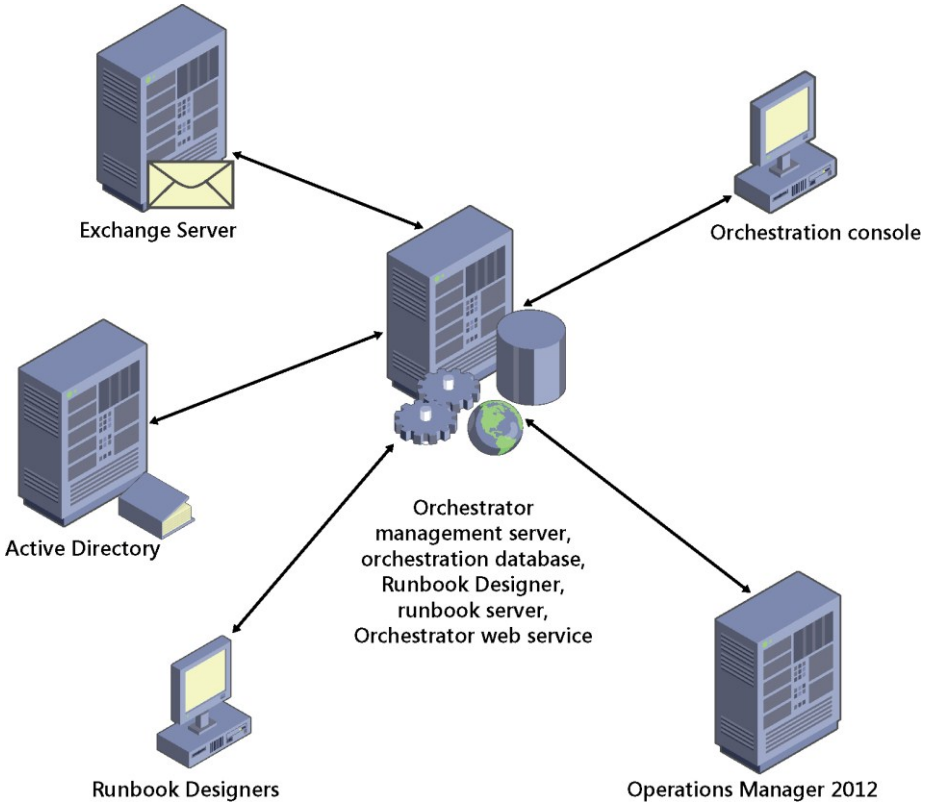


FIGURE 3-4 The Single-server Orchestrator 2012 infrastructure.

Make sure you are aware of the hardware and software requirements. You'll find these at *http://technet.microsoft.com/en-us/library/hh420361.aspx*.

High availability Orchestrator 2012 infrastructure

A high availability Orchestrator 2012 design pattern presented in Figure 3-5 is the most common and widely used one. The critical components of Orchestrator 2012 that were described earlier in this chapter are configured to enable high availability:

- Orchestration database
- Runbook servers
- Orchestrator web service

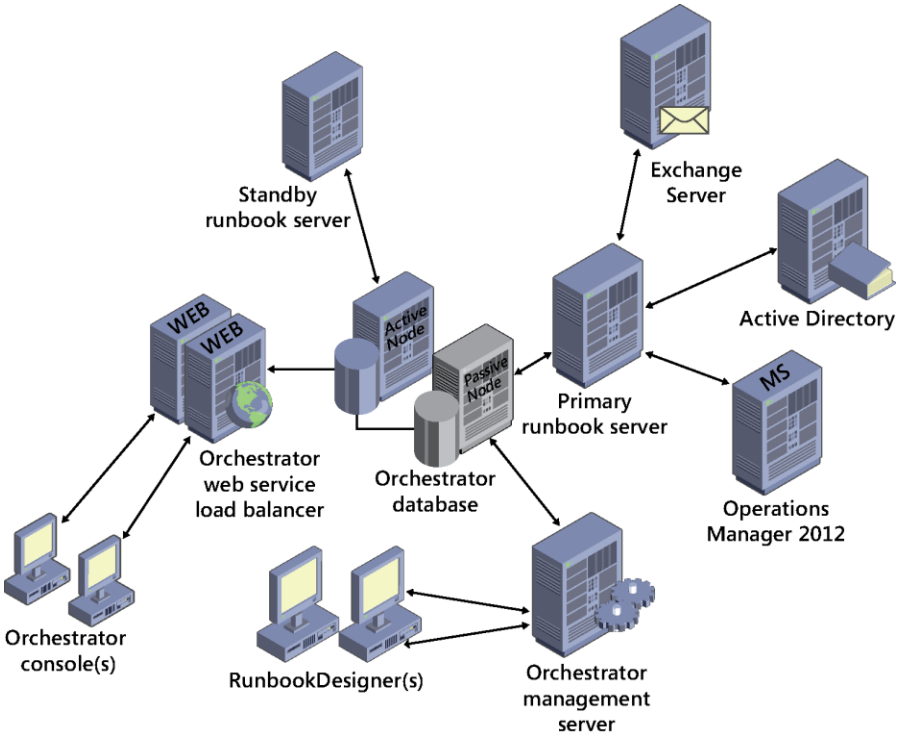


FIGURE 3-5 The High Availability Orchestrator 2012 infrastructure.

Orchestration database

The Orchestration database used to store configuration information, runbooks, and logs is deployed and configured on a two-node SQL Server cluster.

Runbook servers

In Figure 3-5 two runbook servers are deployed for failover and load balancing, but you can use more than two and deploy additional runbook servers to provide additional capacity. The exact number of runbook servers depends on the business and technical requirements of your environment.

Scaling out runbook servers can be performed using the Deployment Tool of Orchestrator, where a wizard guides you through the deployment process in a few steps.

Orchestrator web service

The Orchestration console uses the Orchestrator web service and those two components depend on the performance of the orchestration database and the IIS server that hosts the Orchestrator web service. This is why we would recommend you install the Orchestrator web service on load balanced servers running IIS to provide high availability and additional

capacity. Similar to the runbook servers, the more servers that are available to provide the service, the better the performance and the more requests can be handled.

High availability and multisite Orchestrator 2012 infrastructure

The Orchestrator design pattern presented in Figure 3-6 is intended for managing devices and/or integrating systems from another datacenter in case your network is too slow and/or security plays a more important role than in the other scenarios.

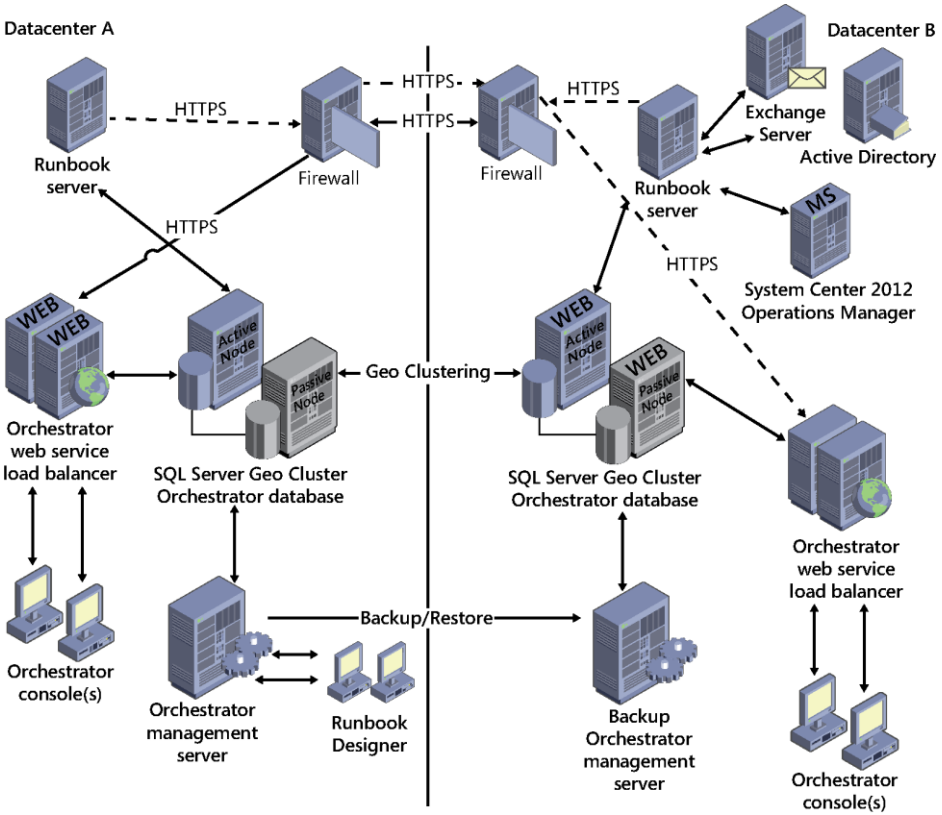


FIGURE 3-6 The High availability and multisite Orchestrator 2012 infrastructure.

On slow networks there is always a risk of packet loss, but configuring a runbook server across a wide area network (WAN) for the purpose of managing devices and/or integrating with systems is not recommended and will result in failures in your runbooks. In cases where latency in your network is greater than 30ms, the recommended approach is to configure a dedicated Orchestrator infrastructure in each datacenter and build runbooks that pass data between the datacenters using the Orchestrator web services. So instead of implementing the

runbooks that will cross datacenter boundaries you will pass the data through the web service interface to another runbook in the remote datacenter.

For SQL you would implement a SQL Server geographically distributed cluster on which you deploy the Orchestrator database. The Orchestrator web services will be deployed separately in each datacenter. Those will access the SQL instance in the same datacenter. Runbook servers will be deployed in each datacenter, each accessing their local SQL instances. If they need to communicate across datacenter sites, the web service of the remote site can be used to leverage https connections.

There is only one active management server, with backups/restores another standby management server is kept updated and in case of a datacenter loss, the backup management server can take over.

CHAPTER 4

Modular runbook design and development

In this chapter we will introduce the topic of runbook design and the foundational components that Microsoft System Center Orchestrator 2012 provides for creating runbooks. The more familiar you are with all of the properties, behavior, and functionality of these building block components, the more efficiently you will be able to build runbooks to automate your IT processes. This section is similar to the process a developer must go through when learning a new development language. Developers learn the capabilities of all the statements and commands while also learning how to implement basic programming and algorithmic concepts such as looping and branching.

What is a runbook?

A runbook is a set of activities performed in a particular order to achieve a goal such as automating a particular IT process. The concept of a runbook dates to the early days of information technology and typically took the form of written documentation of IT processes and procedures for administering complex systems such as mainframes. The intent was the same as it is today: predictable and repeatable administration of IT systems in a structured way, not dependent on the knowledge of a particular individual or improvisation. Most large outages begin as small outages which are then compounded by administrators rushing to try to restore services as quickly as possible. Standard procedures for routine activities which are tested and refined over time greatly reduce the odds of human error. Runbooks in document form proved successful and the next logical step was to use scripting, management tools, and now orchestration tools to further automated processes.

In Orchestrator 2012, runbooks can range from simple two- or three-step automated processes up to large integrated collections of dozens or more runbooks automating entire IT processes such as patching or disaster recovery.

Creating runbooks

While runbook design and testing is primarily targeted at IT professionals and does not require developer expertise, many of the concepts used in script and code development do apply. Examples include modular or service-oriented design, designing for test, creating reusable

libraries, and so on. Before we get into advanced runbook design, we'll cover the basics which are the key building blocks for more advanced scenarios.

Runbook Designer

Introduced in previous chapters, the Orchestrator Runbook Designer is the product's defining feature. The Runbook Designer provides a Microsoft Visio or Visual Basic-like design surface for runbook authoring, as shown in Figure 4-1. The combination of a visual designer and a wide range of standard activities and integration packs provide an IT pro or infrastructure developer a much more approachable solution for runbook automation than pure scripting, which requires more of a developer background. Orchestrator enables quite powerful automation to be created without requiring code or scripting, while also enabling code and scripts to be used if needed in advanced scenarios. Once the boundaries of the built-in activities and integration packs are reached, runbooks can then include script code (such as Windows PowerShell) or more advanced scenarios (custom .NET integration packs).

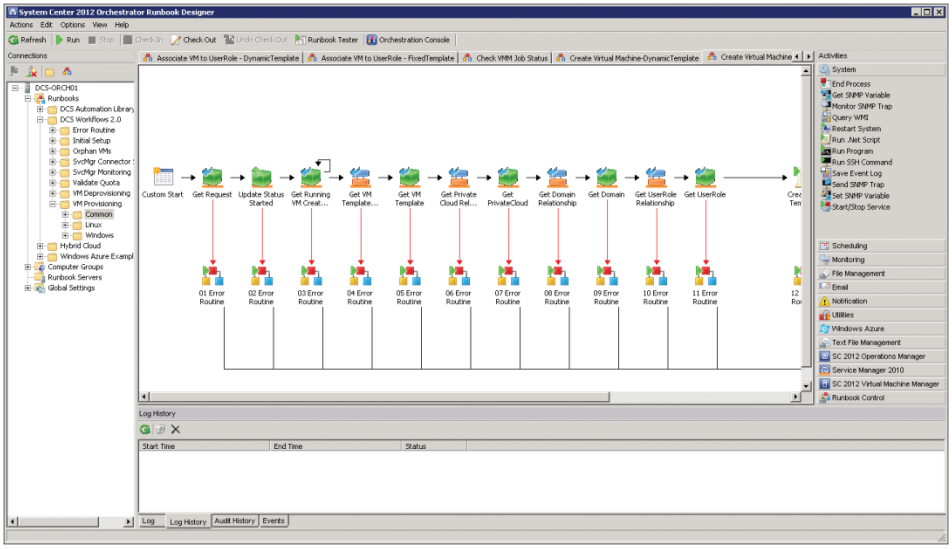


FIGURE 4-1 The System Center Orchestrator Runbook Designer.

Runbook properties

Runbooks include a variety of properties such as name and description. Both can be critical in terms of version control. Runbook properties include scheduled execution, meaning the author can specify that on particular date/time intervals the runbook will be executed. The system time of the runbook server is utilized for scheduling.

Runbook permissions

Runbook permissions can be set by right-clicking the folder or specific runbook. By default, only users in the Orchestrator Users group have full access to runbooks. Additional access can be granted to users to run, start, stop, view, and change runbooks at either the folder level or the runbook level. Keep in mind the permissions to edit or start a runbook are separate from the permissions or security context a runbook executes under. A runbook executes in the security context of the Orchestrator runbook service service account or if using an integration pack, the account used in the integration pack connection to the target management system.

Using runbook activities

Activities are key Orchestrator components that perform an individual function such as copying a file, opening an SSH connection, or powering on a virtual machine. Multiple activities and the links between them are what comprise a runbook. Activities may get or set parameters and execute scripts or tasks, among many other possible actions.

Standard activities

The standard activities are those that are built into a default installation of Orchestrator. These activities tend to be "utility" activities such as file management, email, and runbook control activities. Most runbooks will use a variety of standard activities. The full list of standard activities is listed at *http://technet.microsoft.com/en-us/library/hh403832.aspx*.

Monitoring activities

Monitoring activities are activities which monitor for specific events or conditions and then begin execution of the runbook they are part of. Examples include monitoring a folder for the existence of a specific file, or the state of service on a target machine. A monitor activity, if utilized, must be the first activity in the runbook and any runbooks beginning with a monitor activity must be started in order for the monitoring to be in effect. These conditions have the following implications:

- Since the runbook must be running in order for the monitor to be effective, it consumes one runbook of the maximum number of runbooks the server can execute, which is 50 by default (this can be configured higher if the server has the resources to execute more).
- You need to ensure that upon runbook server reboots or other operations that all your runbooks with monitors are started (which itself is a great example of a problem that can be solved with a runbook, one which starts all of your other runbooks containing monitors).
- A common use of a monitor activity is the Date/Time activity which monitors for specific dates, times, or intervals and executes the rest of the runbook. If you want to check the availability of a server every 15 minutes, you can create a runbook that starts with a Date/Time monitor set to fire every 15 minutes then run your activities to check the target server as a subsequent step.

Customized activities

Customized activities are those that are delivered as part of an Orchestrator integration pack (IP) such as the Microsoft-provided System Center integration pack or from custom objects and integration packs created using the Orchestrator Integration Toolkit (OIT). Use of the OIT is beyond the scope of this book, more information can be found here: *http://msdn.microsoft.com/library/dd834977.aspx*

Common activity properties

Common activity properties are those that all Orchestrator activities contain. Examples include Name and Description under the General tab of the activity. Activities also have a Details tab which may include required properties or other fields. Finally, there is a Run Behavior tab which includes settings for Returned Data Behavior and event notification. The Returned Data Behavior settings are crucial to understand. An activity might return a large amount of data. Consider Query Database standard activity which might return 100 rows of data depending on the configured query. Two possible uses for the returned data are supported depending on the activity settings. You may want to execute a subsequent step on each row of data returned, which Orchestrator lets you do and is one of its most powerful features (parallel execution of multiple runbooks) or you may want to "flatten" the data, and pass all 100 rows of data to the next step as one large published data item. Orchestrator provides the option to return data "flattened" using the separator you specify (such as a comma).

Event Notification enables you to tell Orchestrator to log an application when an activity takes longer than a duration you specify to execute or if the activity fails. For instance, if the above database query example takes more than 60 seconds to execute, you could configure Orchestrator to log a notification event.

Controlling runbook workflow execution

The workflow control activities within Orchestrator are the foundation that all runbooks are built from. Understanding the features and functionalities of the workflow control activities is important because a full understanding opens a wide range of new scenarios for building runbooks and systems of runbooks that work well together.

Starting point

Starting point activities in Orchestrator are the activities which all runbooks must start with. A runbook can only have one starting point activity. Typically runbooks will utilize an Initialize Data activity or a Monitor activity as the starting point. The starting point activity begins when the runbook is started by a console operator, invoked by another runbook, or invoked via the Orchestrator web service. Monitoring activities were discussed previously. The Initialize Data activity is also commonly used as it provides the ability to specify input parameters for the runbook. Any parameters configured on the Initialize Data activity will be presented as input fields when the runbook is executed from the Orchestration Console. Input parameters are also

available when the runbook is executed through the web service or when triggered from System Center Service Manager if the Orchestrator connector from Service Manager is configured. Examples of common input parameters include the computer name of the target system to be managed, a transaction identifier, and so on.

Links

Links connect one activity to another in Orchestrator. Links include properties which allow you to establish conditional logic. An example is a link that only allows proceeding to the next activity if the previous activity was successful. Another example would be a link that only proceeds if the previous activity returned a specific value (that is, a success exit code from a script). An activity can have multiple links on its input side and/or multiple links on its output side. This capability enables a wide range of branching scenarios to support complex and multistep processes. The example in Figure 4-2 illustrates one activity that has multiple links on its output side resulting is several branches which may execute depending on the link conditions.

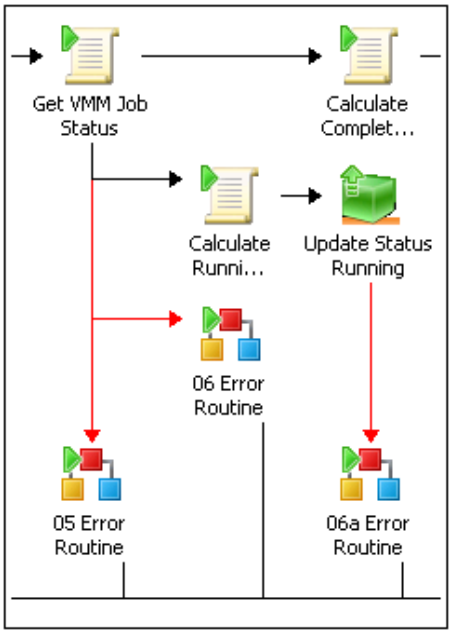


FIGURE 4-2 One activity may have multiple output paths.

Loops

Activities can be configured with a loop so that the activity can be repeated if it fails or to test the output of the activity for specific conditions. Loops can be used for runbooks or processes which might have high latency activities such as rebooting a server and pinging it to determine its availability. Loops can be configured with conditions for when to continue the loop, when

to exit the loop, and an optional delay time between loop attempts. One of the options for exit conditions is a configurable maximum number of loop executions. Using the ping example, the loop could have a success exit criteria for when the ping is successful and could have an exit criteria of a maximum of 5 loop executions with a 60 second delay between them. This loop would then attempt to ping the server over the course of five minutes and have two paths out of the loop, a success path and a failure path.

While loops are powerful, they must be used carefully. A key design goal of efficient runbooks is relatively small, fast-executing runbooks which are assembled into larger processes (a design approach that will be described in detail in subsequent sections). Loops introduce longer run times and latency in overall process execution. In many cases this is an acceptable or required tradeoff.

Invokes

The Invoke Runbook activity executes any existing runbook that you specify. Data can be transferred to the invoked runbook by configuring an Initialize Data activity in the invoked runbook with input parameters. Data can be returned from the invoked runbook by configuring a Return Data activity. Using just those two capabilities, larger structures consisting of multiple runbooks or multiple tiers of runbooks can be created. As an example, this enables a modular and tiered structure to runbooks where a top level "control" runbook can call other runbooks in a particular order or under particular conditions. This helps keep individual runbooks to a manageable size and also encourages a modular approach to design where the individual component runbooks are usable in many different processes rather than repeating all that development in large monolithic runbooks which become difficult to manage.

Orchestrator data bus

The data bus in Orchestrator is a mechanism that passes information from one activity in a runbook to another activity. Data from one activity is "published" to the data bus which makes it available to any downstream activities in the runbook. This is another critical feature in Orchestrator that enables advanced process automation. Orchestrator is most often used to orchestrate actions across multiple management systems. The data bus enables a runbook to query multiple systems for data and to allow subsequent steps in the runbook to utilize all of the data collected. As an example, certain activities being automated against a virtual machine might need data from Virtual Machine Manager (VMM) as well as Service Manager. In this case, a runbook can collect data from both systems which will be available on the data bus for later steps to utilize.

The screenshot in Figure 4-3 illustrates the power of the Orchestrator data bus. In this example the runbook contains a Windows PowerShell script. The Windows PowerShell script is able to utilize the data bus to populate the value of variables in the script. At runtime, the script used subscribes to the published data which is substituted in the script (represented by the blue hyperlinks below). The right side of the screenshot illustrates that any data published by previous steps in the runbook on the data bus is able to be selected.

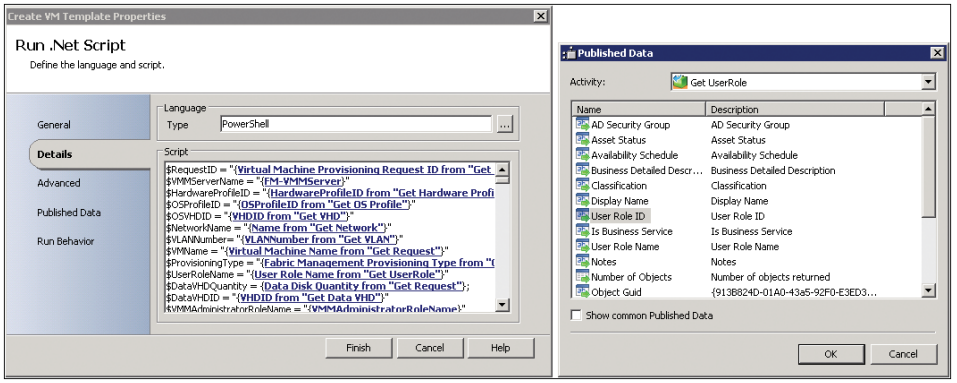


FIGURE 4-3 An example of the Orchestrator data bus.

Return data activities

The Orchestrator data bus is an extremely powerful feature. While typically the data bus is used within a single runbook, there is also the ability to return data from one runbook to another. The Return Data activity allows you to return data from the current runbook to a runbook that invoked the current runbook. You configure the runbook data by configuring the data parameters in the Runbook Properties dialog box. This is a powerful concept and key enabler of modular runbook design and the framework that is described in subsequent chapters. The implications of this capability are that processes can be broken into small modular tasks (runbooks) and that each individual task or runbook can return status and data to a higher level runbook. This is logically equivalent to a function in code which can be called and which can return data. The screenshot in Figure 4-4 illustrates the configuration of a return data activity. These activities are typically the last step in each of the paths in a runbook. In this example, this is the Return Data activity for the error path of a particular runbook. What it shows is that the runbook is going to return six parameters and all but one are configured with data. The three parameters with data in brackets are examples of subscribing to data from the Orchestrator data bus. What that means is the value of those three parameters on the data bus at the time the Return Data activity is executed will be returned.

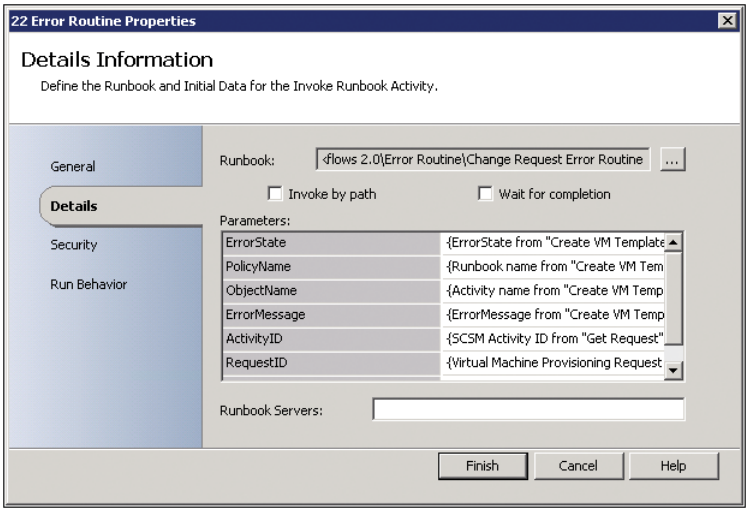


FIGURE 4-4 An example of an Orchestrator Return Data activity.

Extend functionality with integration packs

All of the Orchestrator activities described so far are included in a default Orchestrator installation. Similar to System Center 2012 Operations Manager with management packs, Orchestrator can be extended by integration packs, which are groups of new runbook activities specific to a particular purpose or management system. For example, Microsoft provides integration packs for the various System Center components such as the VMM integration pack. Orchestrator also has a large third-party ecosystem of integration packs created by Microsoft partners.

Microsoft-provided integration packs

Microsoft provides integration packs for all of the System Center products, as well as other Microsoft and third-party products and technologies.

The following integration packs are currently available:

- Active Directory Integration Pack for System Center 2012 - Orchestrator
- Exchange Admin Integration Pack for Orchestrator in System Center 2012 SP1
- Exchange Users Integration Pack for Orchestrator in System Center 2012 SP1
- FTP Integration Pack for Orchestrator in System Center 2012 SP1
- HP iLO and OA Integration Pack for System Center 2012 - Orchestrator
- HP Operations Manager Integration Pack for System Center 2012 - Orchestrator
- HP Service Manager Integration Pack for System Center 2012 - Orchestrator
- IBM Tivoli Netcool/OMNIbus Integration Pack for System Center 2012 - Orchestrator

- Representational State Transfer (REST) Integration Pack Guide for Orchestrator in System Center 2012 SP1
- System Center Integration Pack for Microsoft SharePoint
- Windows Azure Integration Pack for Orchestrator in System Center 2012 SP1
- VMware vSphere Integration Pack for System Center 2012 - Orchestrator
- Integration Packs for System Center:
    - Virtual Machine Manager
    - Operations Manager
    - Service Manager
    - Configuration Manager
    - Data Protection Manager

Third-party integration packs

A wide range of integration packs are available through Microsoft partners. In some cases, vendors of management systems and hardware create integration packs for their Orchestrator to integrate with their solutions and in other cases, Microsoft partners, such as Kelverion, develop commercial integration packs for a number of third-party management systems (*http://www.kelverion.com/products/*).

Community-developed integration packs

Another source for Orchestrator integration packs and other utilities is the CodePlex site. CodePlex is Microsoft's free open source project hosting site. The Orchestrator team maintains a list of CodePlex projects and resources at *http://orchestrator.codeplex.com/*.

Modular runbook design

Modular runbook design is a key objective which results in the maximum benefit from implementing System Center Orchestrator. The concept is very similar to object-oriented or service-oriented software design which emphasizes code libraries and code reuse. Our approach to systematic runbook design utilizes a modular approach to enable as high a return on investment of runbook development efforts as possible.

Modular management architecture

Before describing our modular runbook framework in detail, we must first reiterate that System Center Orchestrator operates through other management systems which must be in place in order for Orchestrator to be utilized. While basic runbooks can be created and executed using the built-in foundation objects regardless of whether other management systems are in place, typically for real-world scenarios Orchestrator will integrate with and act

through other management systems such as the other System Center components or third-party management systems.

Generally, a mature management infrastructure provides several different layers of functionality such as those outlined in Figure 4-5.

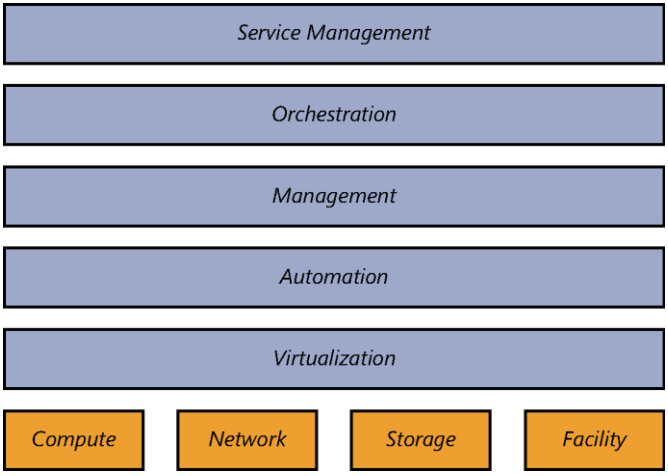


FIGURE 4-5 An illustration of modular management architecture.

A rich automation layer is required. The automation layer must be enabled across all hardware components—including server, storage, and networking devices—as well as all software layers, such as operating systems, services, and applications. The Windows Management Framework—which comprises Windows Management Instrumentation (WMI), Web Services-Management (WS-Management), and Windows PowerShell—is an example of a rich automation layer that was initially scoped to Microsoft products, but that is now leveraged by a wide variety of hardware and software partners.

A management layer that leverages the automation layer and functions across physical, virtual, and application resources is another required layer for higher IT maturity. The management system must be able to deploy capacity, monitor health state, and automatically respond to issues or faults at any layer of the architecture.

Finally, an orchestration layer that manages all of the automation and management components must be implemented as the interface between the IT organization and the infrastructure. The orchestration layer provides the bridge between IT business logic, such as "deploy a new web-server VM when capacity reaches 85 percent," and the dozens of steps in an automated workflow that are required to actually implement such a change.

The integration of virtualization, automation, management, and orchestration layers provides the foundation for achieving the highest levels of IT maturity.

Automation layer

The ability to automate all expected operations over the lifetime of a hardware or software component is critical. Without this capability being embedded in a deep way across all layers of the infrastructure, dynamic processes will grind to a halt as soon as user intervention or other manual processing is required.

Windows PowerShell and several other foundational technologies, including WMI and WS-Management, provide a robust automation layer across nearly all of Microsoft's products, as well as a variety of non-Microsoft hardware and software. This evolution provides a single automation framework and scripting language to be used across the entire infrastructure.

The automation layer is made up of the foundational automation technology plus a series of single-purpose commands and scripts that perform operations such as starting or stopping a virtual machine, rebooting a server, or applying a software update. These atomic units of automation are combined and executed by higher-level management systems. The modularity of this layered approach dramatically simplifies development, debugging, and maintenance.

Management layer

The management layer consists of the tools and systems that are utilized to deploy and operate the infrastructure. In most cases, this consists of a variety of different toolsets for managing hardware, software, and applications. Ideally, all components of the management system would leverage the automation layer and not introduce their own protocols, scripting languages, or other technologies (which would increase complexity and require additional staff expertise).

The management layer is utilized to perform activities such as provisioning the storage-area network (SAN), deploying an operating system, or monitoring an application. A key attribute is its abilities to manage and monitor every single component of the infrastructure remotely and to capture the dependencies among all of the infrastructure components. System Center 2012 has evolved to meet the requirements of managing a heterogeneous datacenter infrastructure.

Orchestration layer

The orchestration layer leverages the management and automation layers. In much the same way that an enterprise resource planning (ERP) system manages a business process, such as order fulfillment, and handles exceptions, such as inventory shortages, the orchestration layer provides an engine for IT-process automation and workflow. The orchestration layer is the critical interface between the IT organization and its infrastructure. It is the layer at which intent is transformed into workflow and automation.

Ideally, the orchestration layer provides a graphical interface in which complex workflows that consist of events and activities across multiple management-system components can be combined, so as to form an end-to-end IT business process such as automated patch management or automatic power management. The orchestration layer must provide the ability to design, test, implement, and monitor these IT workflows. System Center Orchestrator

provides the foundation for such an orchestration layer, however, a structured and modular approach to its utilization is also required.

Runbook design fundamentals

The Orchestrator designer does not enforce any standards or patterns for runbook design and is effectively an infinite canvas. While there are no strict limitations, there are several best practices for runbook design.

Each activity in Orchestrator (other than starting point activities) has both an input side and an output side, as well as a set of properties. Activities can have multiple inputs and outputs. Using just these constructs as well as the general left to right execution flow they encourage, we recommend using a "three rail" design. A "three rail" design is where the center rail performs the main action, audit or notification functions are at the top, and error handling is below. Figure 4-6 illustrates a small runbook utilizing this design layout.

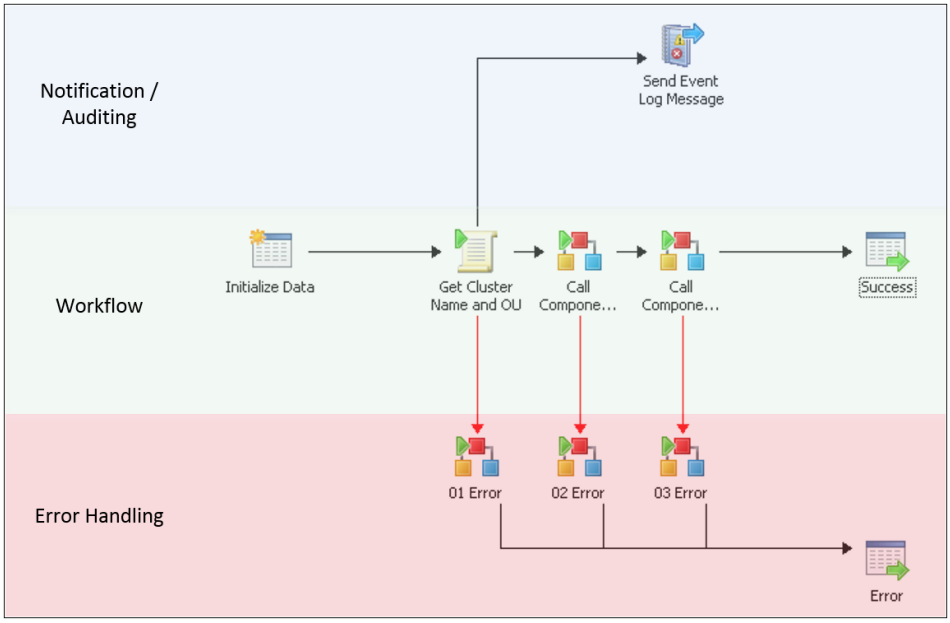


FIGURE 4-6 A runbook that uses the "three rail" design paradigm.

The primary functionality of the runbook is contained in the middle rail, proceeding from left to right. Each left to right link contains conditional logic capturing the success conditions required to proceed. All failure conditions, as well as "catch all" conditions for unexpected scenarios, should be captured in links down to the lower, error handling rail.

The use of link colors, labels, and line thickness can also visually enhance the runbook properties. We use default black links for success conditions and red links for error conditions. In cases where a subsequent step is executed multiple times in parallel, we use a different color

and a label to indicate the previous step is returning multiple results which will each execute the subsequent steps in parallel. Figure 4-7 illustrates this approach.

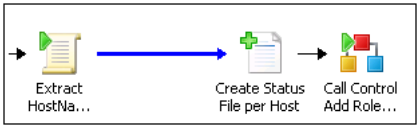


FIGURE 4-7 Link labels, coloring, and thickness are used to indicate functionality.

The visual encoding of process logic in runbooks is a key differentiator between runbooks and large Windows PowerShell or other scripts. The visual nature of runbooks assists in understanding the functionality of the runbook particularly in cases where the original author has changed roles or someone new is now responsible for the runbook. The more standard and descriptive the naming (described in later chapters) and the more consistent the usage of the three-rail design, labels, and colors, the easier it will be to maintain a large library of runbooks.

Error handling

The second rail of the three-rail design pattern for runbooks is error handling. A key mentality of enterprise runbook design is similar to software development where for each step you need to consider expected success states, expected error states, and unexpected states. That is the case both for each activity in the runbook and the runbook itself. In our framework, nearly every single activity in every runbook has an error path on the output side of the activity. The error path is enabled by configuring a link from the given activity to an Invoke Runbook activity which calls a common error handling routine (described shortly). The error path link must be configured with conditions such that it will only execute in error or unexpected conditions. The screenshot in Figure 4-8 illustrates an example of this.

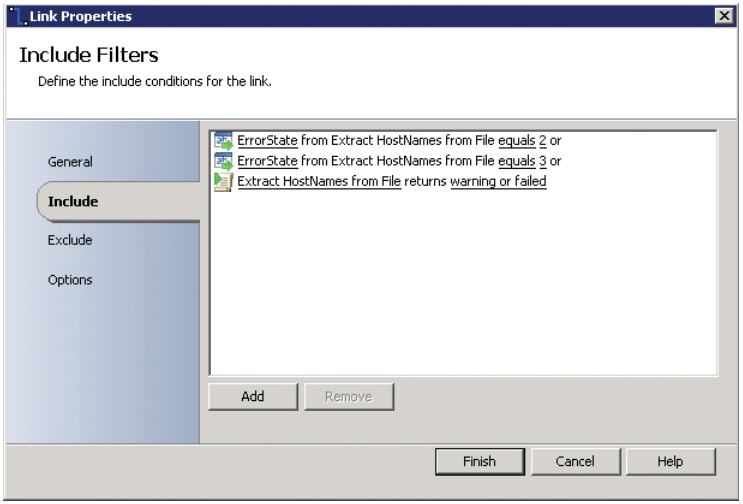


FIGURE 4-8 An example of error handling link properties.

There are three cases where this error path will be taken in the runbook. The activity this link is attached to is a Run .Net Script activity which is executing a small Windows PowerShell script. The Windows PowerShell script will return a variable called ErrorState which contains the execution status of the Windows PowerShell script (values of 0 and 1 are successful while values of 2 or 3 indicate errors). So this error path in the runbook will execute if the value of ErrorState is 2 or 3. That covers cases of logical or expected errors in the script, meaning these will be errors that are handled in the script code. For unexpected errors where there is either a terminating or syntax error in the Windows PowerShell code or there is an internal failure in Orchestrator, the last condition in the screenshot should catch it. That condition contains the status of the activity itself (in this example, the Run .Net Script activity was named "Extract HostNames from File"). So the three conditions in this link should capture both expected and unexpected errors. We believe it is critical to configure every runbook step with such error handling.

The second aspect of our error handling methodology is also critical to runbook portability and reuse and that is that the error path of each runbook activity links to an Invoke Runbook activity which calls a single error handling runbook. The screenshot in Figure 4-9 illustrates this visually.

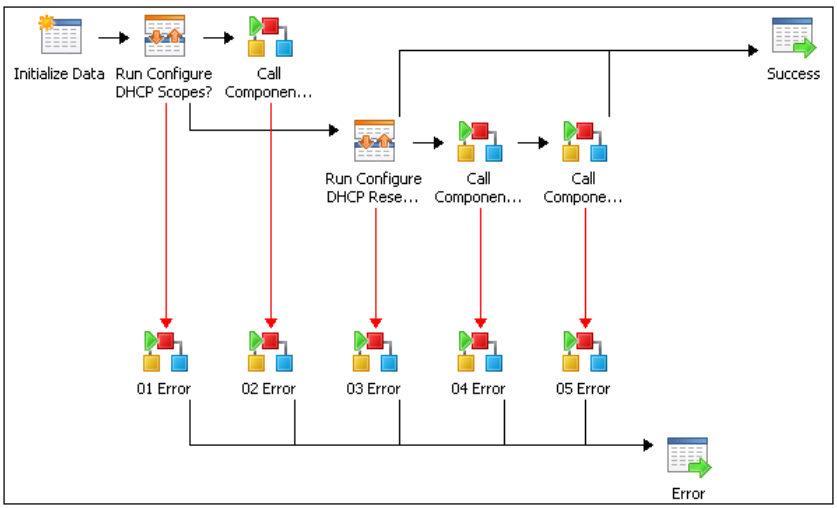


FIGURE 4-9 An example of an error handling design pattern.

Each of the XX Error invoke runbook activities links to a single error handling runbook. The reason each activity links to an individual invoke command as opposed to just one invoke for all of them is that we want to be able to call the error handling runbook and pass data from the activity with the error condition. Examples include the name of the activity that failed as well as tracing data from that activity and so on. This is only possible with an invoke runbook attached to each activity.

While each activity has an associated invoke, all of the invokes call the same error handling runbook. The reason for this is to enable a single location (runbook) where error handling and logging is configured. An example of what the error handling runbook might do is create an Operations Manager alert or create an incident in Service Manager. Alternatively, if a non-Microsoft system, such as Remedy, is used for trouble ticketing, the error handling runbook could be configured to open a ticket in Remedy. The key concept here is that the error handling runbook functionality is configured in one place only and called by all other runbooks when needed. This abstraction means that you can change your error handling runbook and functionality without having to change all of your runbooks. This enables runbook portability to other environments where different ticketing or alerting systems are used while requiring only a change to the single error handling runbook to accommodate the new systems.

Finally, once the error handling runbook has been invoked, the final step of the runbook is a Return Data activity which will return the overall status of the runbook (which in this case would be an error status since the error path was taken).

Logging

Logging in runbook design is also a key consideration. Particularly as you automate more complex processes, different steps may fail or systems may return errors. Troubleshooting large and complex runbooks or modular runbooks with many components is challenging without a robust approach to logging.

Runbook activity pattern

The diagram is from the perspective of the Create New VM activity in the middle of the diagram. Every activity in Orchestrator in general (other than starting point activities) typically begins with input from a previous step, in many cases including Orchestrator published data, then does some activity like run an activity or Windows PowerShell, and publish resulting data or status. The runbook must then determine if the step was successful, then determine whether to take the Success path or the Error path. The dialog boxes in Figure 4-10 show example inputs, published data, success, and error conditions for the highlighted activity.

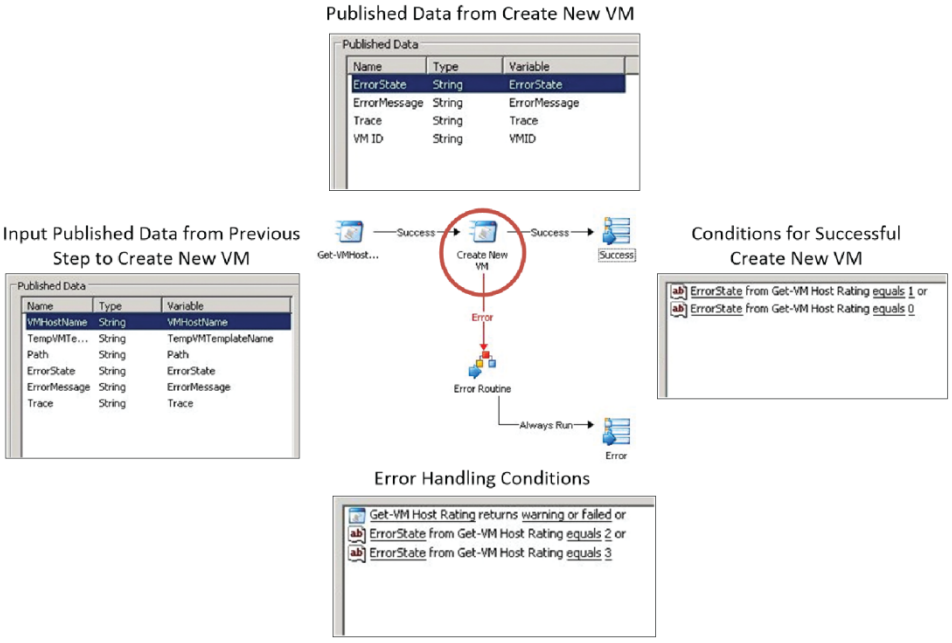


FIGURE 4-10 An example of the Orchestrator activity design pattern.

This example follows the pattern used by every runbook step that executes a Windows PowerShell script using the design patterns in this book. All Windows PowerShell scripts must publish three pieces of data, the most important being the ErrorState variable. There are four acceptable values:

- 0 – Success
- 1 – Success with Info
- 2 – Error
- 3 – Fatal Error

The branching logic after each Windows PowerShell step is triggered based on either the ErrorState value, or the status of the Orchestrator activity itself. So to succeed on the Success path, the value of ErrorState must be 0 or 1. If the value of ErrorState is 2 or 3 or the Orchestrator activity itself threw a warning or error, the runbook will branch to the Error path.

The Error path in all runbooks using this framework is an Invoke Runbook activity that calls a separate Error Routine control runbook which contains the desired error logging functionality (such as generate an event log message, create an Operations Manager alert, or create a Service Manager incident).

Each runbook must also publish a final status (represented by the Success and Error publish policy data objects). This status can be used by a parent control or initiation runbook calling the component runbook to determine what to do next based on the outcome of the runbook called.

Modular runbook architecture

Our modular approach to runbook design utilizes the tiers of runbooks: Component, Control, and Initiation runbooks. Component runbooks are the lowest level and most granular, aligned to the automation layer of management architecture described previously. Control runbooks are the intermediate tier aligning with the management layer of the architecture. Finally, Initiation runbooks are the top tier of the structure and align to the orchestration and service management tiers of the management architecture. Figure 4-11 illustrates these relationships.

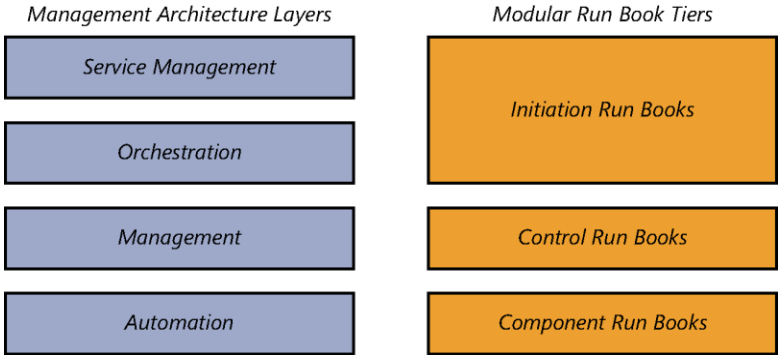


FIGURE 4-11 The layers of the modular runbook architecture.

The purpose of this structure is to deliver maximum reuse of runbook development efforts. Component runbooks are combined by a control runbook to achieve a particular purpose. The control runbook is called by an initiation runbook to begin the process automation. This structure enables a large library of component runbooks to be created over time and utilized by a wide range of control runbooks to automate various processes. The control runbooks can be called by different initiation runbooks so that process automation can be called from different sources such as service management systems, self-service portals, and so on.

Component runbooks

Component runbooks in our framework are low level, typically single purpose runbooks with no external dependencies. Component runbooks are analogous to Windows PowerShell cmdlets or functions in scripting. They take a set of input parameters, perform an action, then exit with success or failure. The difference between a component runbook and an individual activity in Orchestrator is the addition of input parameters, error handling, and multiple output paths. In our framework, a component runbook typically consists of two or three Orchestrator activities. Some examples of component runbooks include: Install a Role or Feature in a Windows Machine or Reboot a Machine. Similar to Windows PowerShell cmdlets, the component runbooks use a verb/noun construct to indicate the action and target of the runbook.

As you can see from these examples, component runbooks are designed to be general purpose and usable by a wide range of higher level automation. Many different processes and higher level runbooks may need to install a role in Windows or reboot a given machine. Rather

than rewrite this functionality in hundreds of different locations, our framework leverages the ability of one runbook in Orchestrator to call another runbook and evaluate the results to implement a modular structure.

Component runbooks, given their low level focus, typically do not include process logic. Including process logic at this level would limit the component runbook's utility in multiple scenarios or other processes.

In our framework, we suggest categories of component runbooks dealing with similar functionality. For example, a category (and associated runbook folder) called Computer Management might contain component runbooks for starting, rebooting, or turning off a computer. Other runbooks such as pinging a computer could also be created.

Control runbooks

Control runbooks are the next level up from component runbooks. The purpose of a control runbook is to encode process logic and call appropriate component runbooks to execute functionality. Control runbooks typically include process and branching logic such as "proceed to the next step only if the current step is successful or returns a particular value," An example would be a runbook that first pings a computer to see if it is online, then attempts to check the server to see if a particular process is running on the server and if not, reboot the server. The three individual steps (ping, check for process, and reboot) would be defined as component runbooks. The control runbook would encode the process logic and order of execution of the component runbooks and have different execution paths for success, failure, and unexpected conditions.

In the example so far, a combination of three component runbooks and one control runbook has been described. While modular, so far this example could have been accomplished with one larger combined runbook. The power of the modular approach emerges when considering additional processes that might be automated. For example, a patching or updating process may need to reboot a given machine one or more times. Using the modular approach, a control runbook for the patching process could call the same server reboot component runbook as the previous example used. Two different control runbooks automating two distinct processes could use some of the same components, meeting the objectives of code reuse and maximum return on investment.

Initiation Runbooks

As modular as component and control runbooks are, one more tier is required in our framework. The highest level tier is called an initiation runbook. The purpose of an initiation runbook is to call one or more control runbooks which then call one or more component runbooks. The reason for this third tier is that there are a variety of ways in which a process may be initiated. A management system such as Operations Manager or Service Manager may need to trigger an automated process. A self-service portal or service catalog may be a location where automated processes are listed and made available for execution.

Separating the method for initiating automation from the automation itself is key to a modular design. This improves modularity and reuse of investments in component and control runbooks. The modularity described so far enables the different tiers of runbooks to be updated and maintained independently, provided they maintain the expected input and output results, each tier or runbook can be updated without affecting the other tiers or runbooks. This enables the creation of a growing library of component runbooks as well as a library of fully automated processes using control runbooks, and finally a variety of initiation runbooks linking external triggering systems to the automation library.

Developing a systematic approach to IT process automation

Many IT organizations have identified a need to streamline IT operations and processes, reduce the burden on IT resources, and improve their ability to meet the complex needs of the businesses that they support. This can often be accomplished by automating time-consuming and repetitive manual processes, a method used to keep the world's largest and most efficient datacenter facilities operating with minimal manual oversight. Armed with a basic understanding of System Center Orchestrator, a wide range of possibilities for automation typically emerges. The key to getting the most out of Orchestrator is a structured approach to deciding what processes to automate founded on a return on investment (ROI) methodology.

The greatest ROI for IT process automation is typically found in areas with high repetition or high complexity requiring a large amount of involvement by IT staff. An analysis of help desk calls and activities over an annual basis is an excellent place to start. Often a large percentage of calls are the result of a small number of root issues which are prime candidates for automation. Analysis of deployment activities (such as desktops, servers, applications) are also prime candidates for process automation. Finally, surveys of IT staff can also identify areas of inefficiency that are candidates for automation. The initial goal is to generate a large list of potential automation targets. The next step is to apply cost information to each of the existing processes to determine how much one instance of the process costs to perform today using existing systems or manual effort. Next, attempt to determine the number of times this process is required in a year. This leads to a current total annual cost of executing the process. With this calculated for all of the identified processes, they can be stack ranked from most to least expensive. As processes are automated, the library of component, control, and initiation runbooks will begin to grow. Over time the results in the ROI analysis for new runbooks becomes more attractive as well as many of the steps for new processes will already exist in the component runbook library from prior efforts.

With the highest ROI candidates identified, detailed requirements gathering for each high ROI process is the next step.

Runbook requirements gathering

The requirements gathering process for automation using Orchestrator runbooks is the first step in a process of breaking down a high level objective such as "automate scale-out of a web farm based on performance monitoring" in progressively more detailed levels. In the requirements gathering phase, the following items should be identified and documented:

- The process to be automated (for example, . scale-out a web farm)
- The conditions that trigger when the automation should be performed (for example, when transactions per node exceed 1000 transactions/sec)
- The definition of scale out for this process (add another web server virtual machine, deploy the web application, add to load balancer)
- Success validation criteria (for example, requests per node drop to under 1000 transactions/sec)
- Expected failure conditions (for example, web virtual machine fails to deploy, load balancer fails to distribute load)
- How to capture unexpected failure conditions (for example, requests per node fails to drop under 1000 transactions/sec or any individual step in the automation fails)
- Does the process require any human approvals or can it be automated end to end
- What data needs to be logged or traced through the execution of the process

The items in the requirements list define the process and surrounding conditions. The requirements list does not include detailed information about the process itself or how it will be automated, which is the next step in the development process.

Process mapping and optimization

It is important to optimize processes before automating them. The existing process may not be as efficient as it could be, and automating an inefficient process will never be as effective as automating an efficient process. In this next step of process automation and runbook development, the existing process should be fully mapped and documented. In many cases the existing process may be manual, may span organizational units, and have any number of different activities and challenges. Full understanding and documentation of the current process is critical in both understanding what must be performed as well as identifying areas that can be eliminated or streamlined.

One method for breaking down an existing process or system is capturing this data in a service map that contains all of the components that define a service such as the three-tier application or web farm used as an example previously. The Microsoft Operations Framework (MOF) defines a simple but powerful structure for service mapping that focuses on the software, hardware, dependent services, customers, and settings that define an IT service and the various teams that are responsible for each component. In this case, the application consists of the database, application, web farm virtual machines, the physical servers on which they run, dependent services such as Active Directory and DNS, the network devices that

connect the servers, the LAN and WAN connections, and more. Failure of any of these components causes a service interruption. You can find a sample service map diagram at *http://aka.ms/SCrunbook/files*.

The service mapping exercise helps to fully define the full set of hardware, software, services, and settings that might be impacted by the particular process being automated.

The next step is to take the desired process (for example, scale out a web farm based on performance monitoring) and list all of the steps in the current process. A relatively poor manual process might look like this:

- Customers call the help desk complaining of slow website performance
- The help desks waits until multiple tickets over several days all complain about performance
- The ticket is escalated to tier two which looks at monitoring data and sees that the website is available and not currently showing performance issues
- More tickets come in complaining of performance issues so the tickets are escalated to tier three
- Tier three support identifies that for several peak periods every day, website performance is poor due to load on the web farm
- Tier three transfers the ticket to engineering which then decides to add a new virtual machine to the web farm
- An engineer then manually deploys and configures a new virtual machine into the web farm
- The engineer does not confirm that the new VM reduced the load on the farm but closes the ticket anyway
- The next day more tickets come in about slow performance
- The help desk knows that a new server was added to the farm so they assume that's not the issue and begins analyzing the application and database tiers as well as storage
- Eventually, a tier three help desk resource identifies that the newly added web server virtual machine was not properly added to the load balancer so is not servicing any requests
- Finally, once the new web server is added to the load balancer, the root issue is resolved and performance is within the expected parameters

Clearly the above process is slow and inefficient. Despite that, many of the steps performed are hard requirements and must be captured. Also, it is important that several different parts of the IT organization shared responsibility throughout the process. It is critical to identify all of those groups and steps so that the process can be streamlined as much as possible to the point that the remainder is the minimum number of steps required to meet the capture requirements.

The new runbook design should contain information about the target functionality, an outline of all the steps included in the runbook, and the layers or products in the infrastructure where those steps must be performed.

Another process commonly targeted for automation is the update or patch management process for servers. The following diagram illustrates how to break down a process and map each of its steps against the management architecture layers discussed previously. This process shows a combination of automated steps as well as several points of human interaction and approvals at the service management layer. The final, streamlined process should be captured in a process map similar to Figure 4-12.

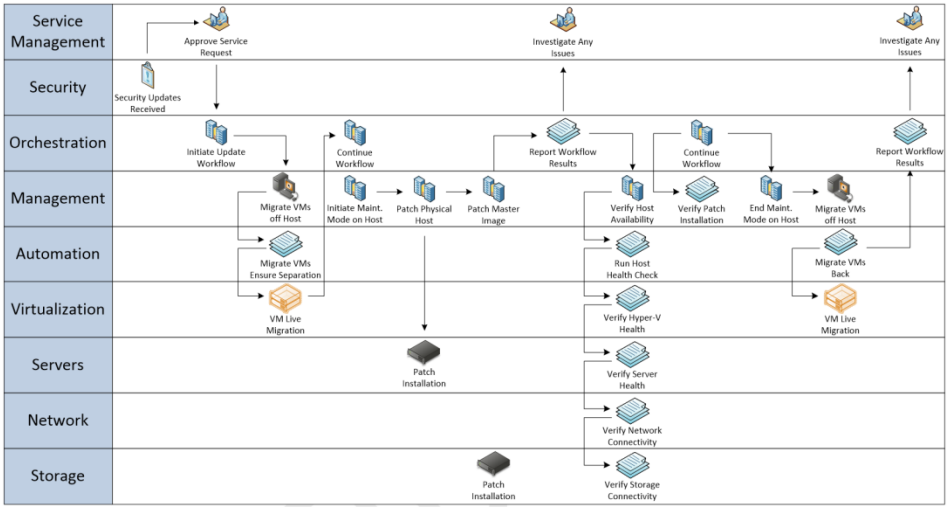


FIGURE 4-12 A sample process map.

Once the streamlined process has been mapped, the next phase of runbook development can begin which is the documentation of functional specifications for each of the steps in the process (components), the overall process itself (control), and the conditions or triggers for the process (initiation).

Documenting runbook functional specifications

The first step in documenting functional specifications is to take the target process and its process map and break it into the three types of runbooks we defined in previous sections. Each low level step of an individual task in the process should be defined as component runbooks. Using the scaling out of a web farm example, the following component runbooks would be required:

- Create New Virtual Machine
- Deploy Web Application to Virtual Machine
- Add Virtual Machine to Load Balancer
- Monitor Web Farm Performance

With those components outlined, the overall workflow or structure of the process should be captured using a control runbook. The control runbook would execute whenever the web farm needs to be scaled out and would first create a new virtual machine, deploy the web application, add it to the load balancer, then validate improved performance. The control runbook would call each component runbook in order and determine whether each was successful or failed and determine the final status of the process. Finally, with the component and control runbooks outlined, one or more initiation runbooks would be defined. In this case, two would be relevant. The first would be an initiation runbook which would be synchronized with System Center Service Manager such that the Service Manager service catalog would have a service request called "Scale Out Web Farm" allowing the operator to manually trigger the execution of the initiation runbook, which would then call the control runbook which would subsequently call all the component runbooks. The second initiation runbook would be a monitor runbook which would monitor web farm performance metrics in Operations Manager and when thresholds are exceeded, it would call the control runbook to scale out the web farm.

In the previous example we have defined a relatively small set of runbooks to automate a relatively complex process. We defined it in a way that allows two methods for the process to be triggered (manually or automatically) and by using the modular structure, much of the work (that is, all the component runbooks) are usable across a wide range of processes.

With this logical outline in place, each of the runbooks defined must have a set of functional specifications created so the runbook authors can create it in Orchestrator. The functional specifications for runbooks should contain the following elements:

- Name
- Description
- Use cases
- Inputs
- Runbooks utilized (if this is a component runbook this would be blank, for control or initiation runbooks this should list the other runbooks it calls)
- Scripts or code requirements
- Integration packs required
- Variables required or utilized
- Connections to other systems
- Other dependencies

While this may seem like a significant amount of documentation for what might be a relatively simple process, it is critical to realize that with the modular framework outlined in this book, most runbook work and documentation will be utilized many times over and likely improved over time through new versions of the runbooks. A solid foundation of documentation is important for long term optimization. In subsequent sections we outline standards and patterns for runbook documentation and versioning.

Runbook authoring and development

With the initial functional specifications drafted, runbook authoring and development can begin. Typically, runbook development will begin by creating and testing all of the required component runbooks. Since component runbooks represent the bulk of the functionality utilized, they must be created and tested first. Once each component runbook is completed and tested (analogous to unit testing in code development), then work on the control runbook can begin. The control runbook will call each component runbook and control the flow of execution such as any sequential or parallel steps. The control runbook will proceed or branch depending on the results. The control runbook and overall flow should be tested under many different conditions and scenarios (analogous to integration testing in code development). Finally, the initiation runbooks can be created and tested.

Since the investment was made to document functional specifications at each level, multiple runbook authors can work at the same time with the team's efforts culminating in the control and initiation runbook testing. In subsequent sections we describe runbook naming, versioning, and collaborative development. During the process of development, the runbook functional specifications should be updated with as-built design documentation, diagrams of the runbooks and process flow, and so on. At the end of the development process the functional specification should represent all of the runbooks requirements and design.

Runbook testing

The Orchestrator Runbook Tester is a key feature that assists in the runbook design process by providing the ability to test runbook functionality prior to implementation of your runbooks in a production environment. The Runbook Tester provides capabilities similar to a code or script debugger in that it allows you to set breakpoints in your runbook, step through your runbook activities one at a time, and view the status of all variables and value on the runbook's data bus. The Runbook Tester is critical to the development of component run books (since the Runbook Tester can only test one run book at a time and not follow invokes to other run books). The Runbook Tester enables unit testing the component runbooks.

For testing control and initiation runbooks, a lab environment with all the systems required for the given process must be in place. It is critical to test many different scenarios and permutations to validate the enterprise readiness of the process being automated. At a minimum, the full process should be tested end to end and results validated. Additionally, many other scenarios should be tested such as multiple simultaneous executions of the process, injection of failure conditions or losses of connectivity, execution of the runbooks while other processes and runbooks are running, and so on. The goal is to test as many production conditions as possible and to verify not only the success paths through the runbooks but also the failure paths, different triggering conditions, error handling, and logging.

Runbook versioning and management

This section discusses the naming, folder structure, and versioning of runbooks.

Naming

A carefully specified naming convention will help runbook authors understand how runbooks relate to one another as they flow within the folder structure in which they are organized, and will better facilitate on-going agile release cycles that may involve the addition of new runbooks.

Folder structure

The recommended folder structure for a runbook library specifies a single root folder containing three additional folders named Component Runbooks, Control Runbooks, and Initiation Runbooks. This structure better enables versioning, and allows for the future addition of new runbook types.

COMPONENT RUNBOOKS

Because component runbooks are not bound to any specific scenario, but instead are leveraged by multiple scenarios, it is most effective to organize them into feature-specific runbook collections. For instance, a folder named Firewall should contain all runbooks related to firewall configurations. This approach enables those feature-specific runbooks to be centrally maintained while providing functionality that can be accessed by multiple control runbooks; each expressing a different scenario. For example, a control runbook used for Hyper-V Replica and a control runbook for VDI deployment can both leverage one instance of a runbook dedicated to firewall configuration.

Next, folders should be created within each feature-specific component runbook folder to accommodate each version. Those folders should be named using a major version number that corresponds to the supported platform, and a minor version number that expresses each change made within that platform. For example, the feature-specific Firewall folder should contain a folder named 1.0 to support Windows Server 2008 R2, and a folder named 2.0 for Windows Server 2012. Those folders should then contain all the component runbooks related to the appropriate platform; for example, Windows Server 2008 R2 Firewall configurations, or Windows Server 2012 Firewall configurations. Additional folders should then be created within each platform-specific folder to accommodate each new version of those runbooks. For example, folders named 1.1, 1.2, and 1.3 should be created for versions of Windows Server 2008 R2 Firewall component runbooks, and folders named 2.1, 2.2, and 2.3 should be created for versions of Windows Server 2012 Firewall component runbooks.

It's always a best practice to use the latest version of a platform-specific component runbook for each new deployment. To that end, if a new runbook version includes interface changes, either adding or removing input parameters, then the new runbook version will stay in the same folder. Its name will remain unchanged although its interface signature will change. Conversely, if a new runbook version does not include interface changes then the new version will be advanced to the next version's folder; for example, moved from the 1.0 folder to the 1.1 folder.

To upgrade existing deployments it is possible to deploy the updated version side-by-side with the existing version. This way the option will exist to use the updated runbook version while having no impact on existing runbooks.

When a new feature is introduced along with a new platform, a new feature-specific folder should be created that contains only the platform-specific folder with which that feature was released. For instance, the data deduplication feature was released with Windows Server 2012; therefore, the feature-specific Data Deduplication folder should only contain a 2.0 folder, and not a 1.0 folder.

CONTROL RUNBOOKS

Because control runbooks are bound to a specific scenario, a new folder should be dedicated to each scenario being modeled. For instance, new folders should be created with names like Hyper-V Replica, and Azure VM Provisioning.

Next, folders corresponding to each version release should be created within each scenario-specific control runbook folder. Those folders should be named using a major version number that corresponds to the supported platform, and a minor version number that expresses each change made within that platform. For example, the scenario-specific Azure VM Provisioning folder should contain a folder named 1.0 to support Windows Server 2008 R2, and a folder named 2.0 for Windows Server 2012. Each version folder should contain all the control runbooks released along with that version. A new version folder should be created for each increment that includes new control runbooks that model new (additional) use-case scenarios.

To ensure versions can be installed side-by-side without breaking previous deployments, existing control runbooks that haven't changed should roll forward from the previous release while only new and changed control runbooks should be included in a new version release.

INITIATION RUNBOOKS

The concepts described for control runbooks also apply to initiation runbooks.

SAMPLE OF ORCHESTRATOR STRUCTURE

Figure 4-13 outlines a sample Orchestrator runbook folder structure using the modular approach and version control guidance outlined in this section.

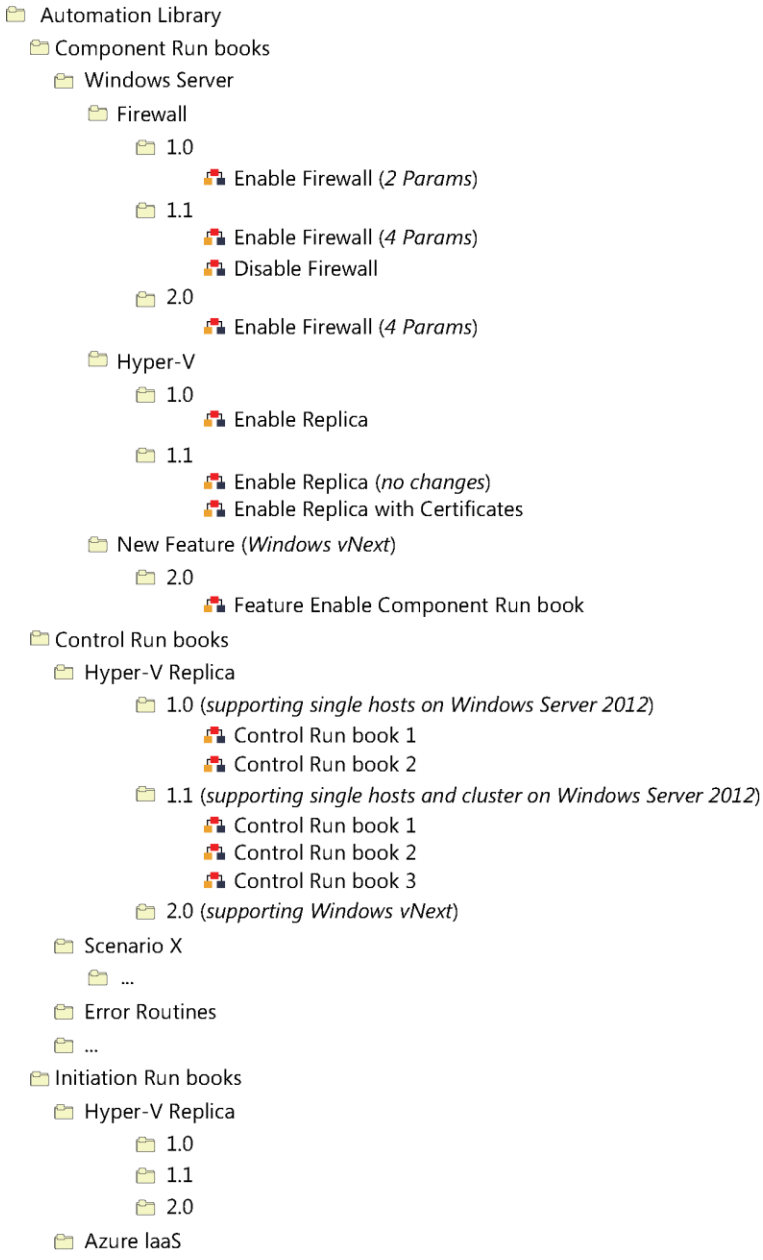


FIGURE 4-13 A sample runbook folder structure and versioning.

Runbook versioning

The primary goal of an automation library versioning system is to facilitate an ongoing, iterative development process where new control runbooks may be released, along with their supporting component runbooks, as their development and testing processes are completed.

What's more, the system should enable runbooks corresponding to separate use-case scenarios to be released independent of one another. The system should ultimately enable the automation library to grow organically over time. A stable release management process is required to avoid breaking existing code while deploying new runbooks.

To facilitate these goals the following schema will be used:

major.minor.revision.build

COMPONENT RUNBOOKS

For component runbooks, the major, minor, revision, and build schema will use the following definitions.

- major: An increment should be created to accommodate the addition of each new platform. For instance, 1.0 should be created for Windows Server 2008 R2, and 2.0 should be created for Windows Server 2012.
- minor: An increment should be created each time a change is made either to a component runbook's interface, or to its behavior. The addition of a new component runbook should also drive a corresponding increment. For instance, version 1.0 would contain the Enable Firewall runbook, but the addition of the Disable Firewall runbook would drive the creation of version 1.1.
- revision: An increment should be created each time a control runbook is updated either to alter its implementation or to address bugs.
- build: This value is set to the date of the build using the (YYYYMMDD) format each time the code is built. For instance, 1.2.2.20130119 represents the version 1.2.2 being built on January, 19th 2013.

CONTROL RUNBOOKS

For control runbooks, the major, minor, revision, and build schema will use the following definitions.

- major: This will be incremented in case of a new platform (for example, 1.0 for Windows Server 2012, 2.0 for Windows 2012 R2).
- minor: This will be incremented in case interfaces or behavior/code of control runbooks are changed (for example, same runbook but more/different parameters) or new control runbooks are being added to the solution (for example, 1.0 supports only single host Hyper-V replica, and 1.1 supports also clustered scenarios for Hyper-V Replica).
- revision: This will be incremented in case control runbooks are updated (either implementation changes or bug fixes).
- build: This will be used to set the value to the date of the build (YYYYMMDD). An example for that is 1.2.2.20130119 for January, 19th 2013.

Every release will include a change log that defines all changes to the last release and a defined new version number.

INITIATION RUNBOOKS

Initiation runbooks use the same approach as control runbooks.

STORING VERSION INFORMATION

Orchestrator does not provide a good means of versioning runbooks. To get around this limitation, and the fact that we are not using Orchestrator variables, we will use files that will host the version number for each runbook. This makes it easier to get and update the version information while deploying a new release of runbooks.

There are multiple options to store version information. The following is a subset of possible ways to implement that:

- Description field
- Database table
- Files on disk

Every solution has some pros and cons, but our strategy is using files as we are using them for variables and status-driven scenarios already.

For every runbook, version information will be stored in an xml formatted file, which will be stored in the file system.

```
<xml>
        <run bookName>Enable Hyper-V Replica</run bookName>
        <description>This run book is used to enable Hyper-V Replica on a single
host</description>
        <version>1.0.0</version>
</xml>
```

The location of the version information is located relative to the path of the Orchestrator structure. The root path for all files is stored in the Orchestrator global variable Runbook Root File Path. Details are described in the "File-based runbook variables" section of this document.

The name of the version information file is always be *run bookVersion.xml*

Microsoft Team Foundation Server integration

Team Foundation Server (TFS) enables collaborative software project development by offering features to facilitate team development; source code control, data collection, reporting, and project tracking.

Unfortunately, System Center Orchestrator doesn't support a native TFS Integration out of the box. Therefore, if using TFS for multiple author runbook development or development teams, it is critical to define standards and practices for using TFS with runbook development. Orchestrator supports exporting and importing runbooks as xml-notated files which can be stored in TFS for source control. Runbooks can be exported individually, on a folder-level, or the entire folder structure. As this process is manual it can take a significant amount of time to export all runbooks individually.

To automate the export and TFS Check-In of runbooks, a proof of concept exists that is described at *http://opalis.wordpress.com/2012/08/06/automating-the-export-and-tfs-check-in-of-workflows/*.

In addition to importing and exporting runbooks, the source code for scripts in runbooks, such as Windows PowerShell, can be maintained as source code files as well.

Runbook deployment and monitoring

Once all of the identified runbooks have been designed, authored, and tested they can then be deployed into the production environment. A key tenet of our modular runbook framework is that all runbooks built using the framework should be portable between different Orchestrator environments such as development and production. The final runbooks from the development environment can be exported and then imported into the production environment. Once the runbooks have been imported into the production environment, any initiation runbooks which utilize a monitor starting point activity should be manually started. The orchestration console can be used to monitor which runbooks are running and to validate that all monitor runbooks are running.

CHAPTER 5

Orchestrator runbook best practices and patterns

In this chapter we start by going over some general best practices to follow when building runbooks. Then we will delve into the best practices for using Windows PowerShell inside your runbooks to handle scenarios where the integration packs (IPs) for Orchestrator cannot be used. Lastly, we describe some patterns for runbook design that will increase reusability of your runbooks as well as patterns for handling state in long-running processes.

Runbook design best practices

This section describes best practices in the following areas of runbook design:

- Flow control
- Publishing data
- Logging execution data
- Looping
- Sequential vs. parallel activity execution
- Setting job concurrency

Flow control

When designing runbooks, you should strive to increase readability of the overall flow. In many instances it is a good idea to label links between activities to further explain execution flow without having to look at the link details. To be able to see the link labels, navigate to Runbook Designer | Options | Configure and select Show Link Labels as shown in Figure 5-1.

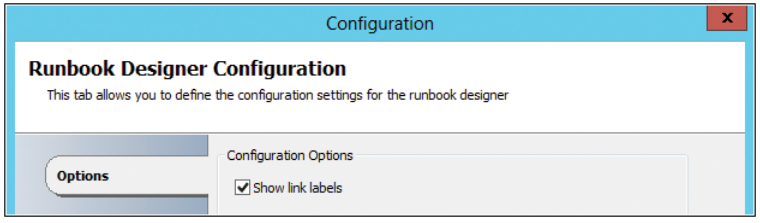


FIGURE 5-1 An Orchestrator Runbook Designer Configuration. dialog box.

To edit a link label, double-click the link to open its property window and then click the General tab and edit the Name field as shown in Figure 5-2.

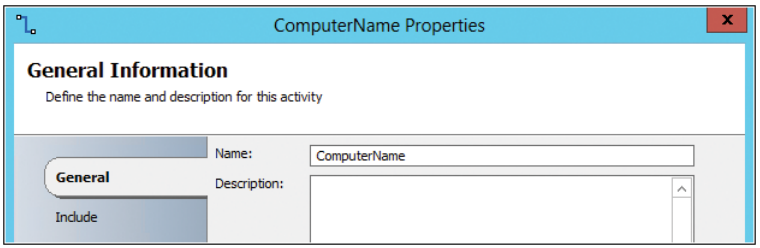


FIGURE 5-2 Editing the name of a link.

When the link label has been edited, it then provides a better understanding of execution flow as shown in Figure 5-3, where execution branches are based on a value on the data bus which has the option of being ComputerName, Name, or ID.

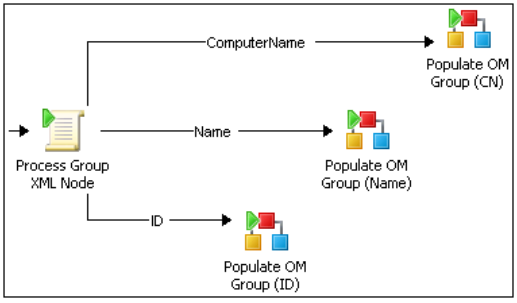


FIGURE 5-3 An example runbook showing meaningful link labels.

The color and/or thickness of the link can also be edited, which can help to visually track success or error paths. To change the color of a link, double-click the link to open the properties window, then select the Options tab and edit the Color property as shown in Figure 5-4.

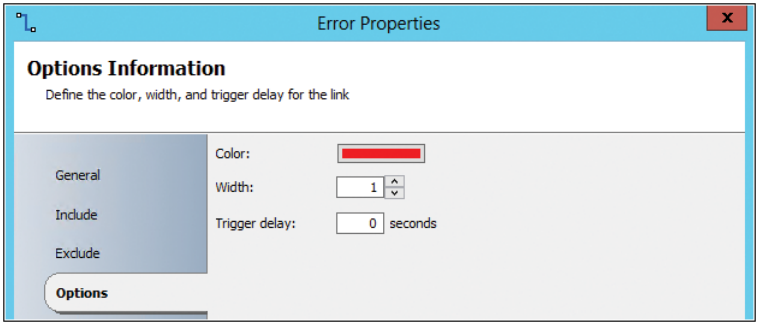


FIGURE 5-4 Setting the line color afor a link.

The resulting link looks like what is shown in Figure 5-5:

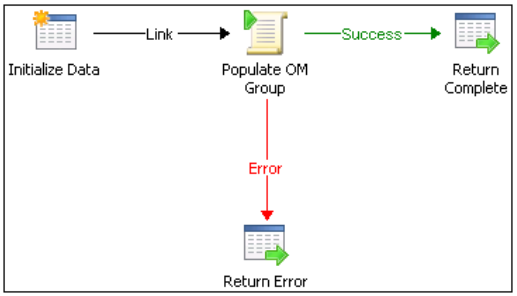


FIGURE 5-5 A sample runbook with color-coded links.

As a design best practice, all error paths should be colored red as shown above and all branching logic should be labeled to indicate what the branch is for.

Publishing data

In situations where you have one runbook that is going to be initiated by another runbook, it is often necessary to return execution state information as well as potentially returning business data to the calling runbook's data bus. To accomplish this, the runbook must be set up to return data. To set, right-click the runbook name, select Properties and click the Returned Data tab. Next, click Add to open the Add Returned Data Definition dialog box. Fill in the Name, Type, and optionally the Description fields, then click OK to save as shown in in Figure 5-6. Repeat this process for all required data and click Finish to save and close the Properties window.

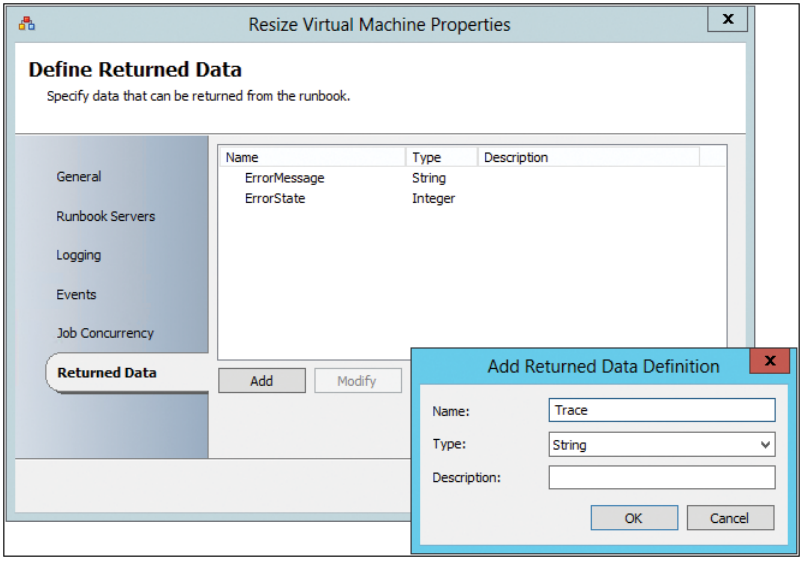


FIGURE 5-6 Setting up a runbook to publish data.

With this set, a runbook can now use the Return Data activity (under Runbook Control) to publish data. Simply drop the activity onto the design surface, linking it to the defined process. Next, double-click the Return Data activity to open its properties window. From the Details tab, all defined return data variables are shown and can be set manually or via Published Data as shown in Figure 5-7.

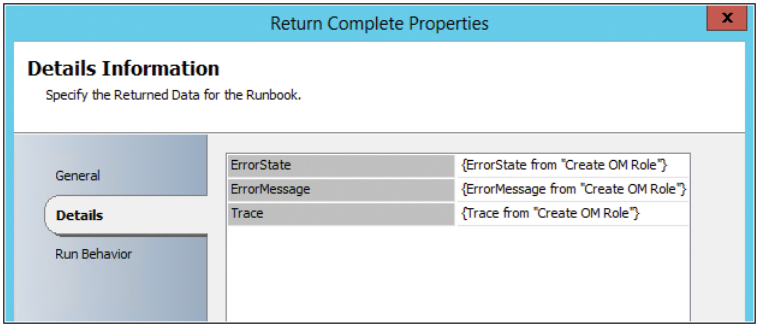


FIGURE 5-7 Settingup the Return Data activity.

Logging execution data

One technique to troubleshoot execution errors is to have Orchestrator log execution data to the Orchestrator database. By default, this is turned off for every runbook but it can be turned on as needed. To set this up, right-click the runbook name, select Properties and click the Logging tab. Then select the Store Activity-specific Published Data check box to have Orchestrator save this data to the database. To save additional general data to the database, check the Store Common Published Data check box. These options are shown in Figure 5-8.

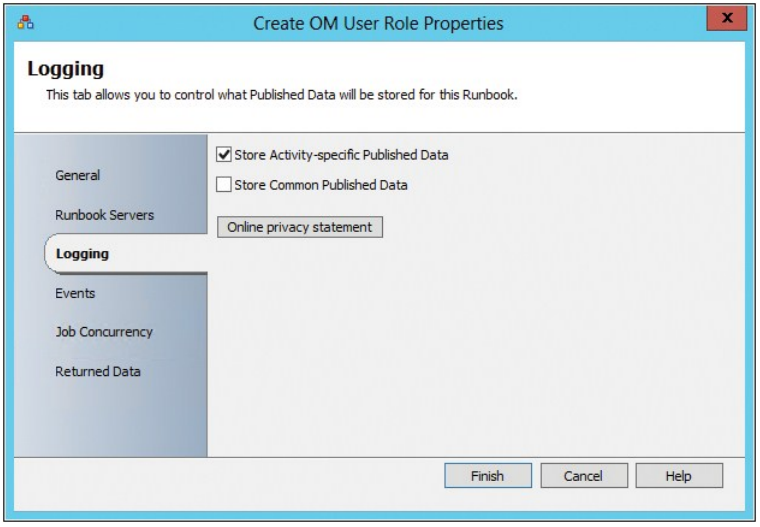


FIGURE 5-8 Configuring the runbook to log Published Data.

Please note that storing execution data will cause more database activity on the Orchestrator database and will increase the size of the database if there are no log-purging settings set up. To set up log purging, from the Runbook Designer, right-click the server name in the leftmost window and choose Log Purge. The Log Purge Configuration dialog box, as shown in Figure 5-9, will open and from there you can set how often you want the logs purged and how much data to retain.

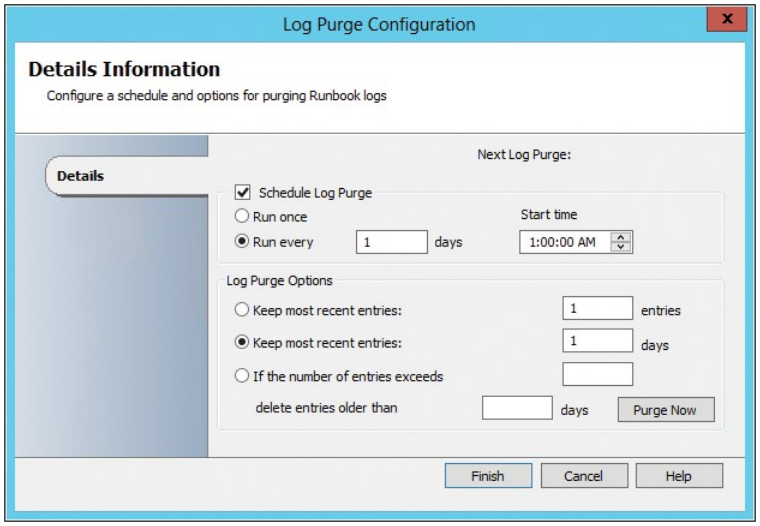


FIGURE 5-9 Configuring Orchestrator log purge settings.

Looping

Looping allows a runbook activity or a complete runbook to run multiple times until a condition is met to end the loop. This condition can be a successful condition (that is, able to connect to a server) or limiting condition (that is, run only a maximum of five times before exiting). To setup looping on an activity, right-click the activity and select Looping from the context menu. This will open up the looping dialog box as shown in Figure 5-10.

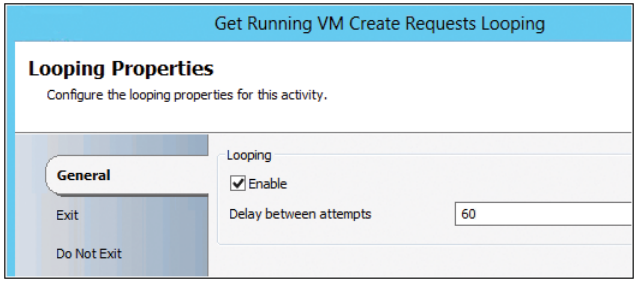


FIGURE 5-10 The Activity Looping dialog box.

On the General tab, make sure to check the Enable checkbox and set the Delay between attempts to the number of seconds between loop runs. On the Exit tab, set the exit conditions. These may both be successful and unsuccessful conditions. Always limit the number of runs using the Loop: Number of attempts property to make sure the loop will end at some point. An example is show in Figure 5-11.

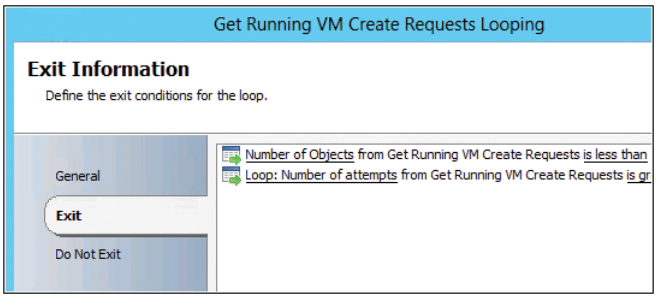


FIGURE 5-11 Setting exit conditions for a loop.

The Do Not Exit tab can be used to further limit the exit criteria. Be careful using the Do Not Exit rules and make sure any criteria set here does not cause an infinite loop.

Sequential vs. parallel activity execution

There is a lot of confusion as to whether multiple activations of an activity happen sequentially or in parallel. The bottom line is inside of a single runbook, all activities run sequentially with one exception. The Invoke Runbook activity, when the Wait For Completion check box is cleared, as shown in Figure 5-12, and the child runbook's job concurrency is set greater than one, allows the child runbook to run in near-parallel for multiple activations. The downside to this though is the inability to capture the result output from these executions.

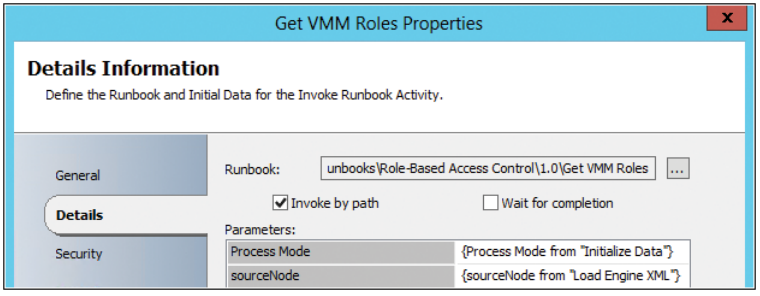


FIGURE 5-12 Setting the Invoke Runbook activity properties.

MORE INFO **For more information on the mechanics of sequential and parallel activity execution, read the blog article at** *http://blogs.technet.com/b/orchestrator/archive/2012/05/11/sequential-vs-parallel-processing-of-runbook-activities.aspx*.

Setting job concurrency

Job concurrency allows a single runbook to have multiple simultaneous executions when the value is greater than 1. To set this up, right-click the runbook name, select Properties, and click the Job Concurrency tab. Then enter a value greater than 1 in the Maximum number of simultaneous jobs textbox to allow multiple concurrent executions, or set to 1 to disable multiple concurrent executions. This is shown in Figure 5-13. The default setting is 1.

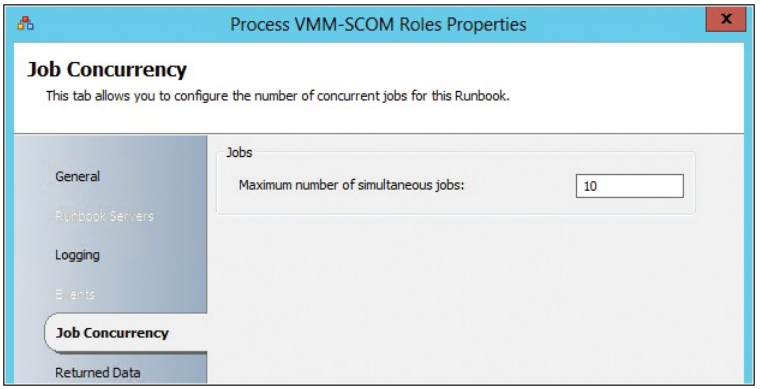


FIGURE 5-13 Setting a runbook's number of simultaneous jobs (job concurrency).

This is an important and impactful setting, so choose wisely. Unfortunately there is no right or wrong answer for this setting as it will truly depend on the runbook in question. There are examples where you don't want to have multiple instances at the same time such as:

- working with external data that cannot be locked to a single process
- working with Orchestrator counters as the counter values can become unreliable if changed by multiple runbooks at the same time

Conversely, there are also instances where it is perfectly okay to set the number of simultaneous jobs to a value greater than one:

- checking that a list of servers are on and able to accept connections
- working with transactional data or data that can be locked to a single process

Using Windows PowerShell in Orchestrator

Using Windows PowerShell in Orchestrator script activities vastly expands what can be automated as compared to the functionality available in the various IPs. For example, Virtual Machine Manager (VMM) ships with over 520 cmdlets compared to the 23 IP activities found in the VMM Integration Pack. Windows PowerShell also has the ability to interact with managed code application programming interfaces (APIs), such as the ones that ship with System Center Operations Manager, so even if the product doesn't have cmdlets, scripts can still be created against their managed API.

The one drawback currently is that Orchestrator runs in a 32-bit process and most Windows PowerShell modules today are written for 64-bit only. To get around this, opening a remote session using Windows PowerShell remoting will allow scripts to run in a 64-bit process.

Windows PowerShell remoting

The following pattern should be followed when using Windows PowerShell remoting in the .Net Script activity. The script outline can be broken down into the following sections, each detailed below:

- Subscribe to Published Data
- Set Trace and status variables to defaults
- Validate inputs
- Establish the Windows PowerShell remote session
- Execute script in the remote session
- Use try/catch/finally
- Append useful data to the trace variable
- Add required Windows PowerShell modules
- Use throw for common errors
- Perform core task logic
- Set ErrorState and ErrorMessage
- Return results
- Prep data for Orchestrator Publishing
- Close the remote session

NOTE **All of the scripts and scriptlets in this chapter are available as a zipped archive from** *http://aka.ms/SCrunbook/files***.**

Subscribe to Published Data

The first step is to create variables to hold any required Published Data from the Orchestrator data bus. All Published Data should be captured at the top of the script and not embedded further into the script to aid readability.

```
$SCOMServerFQDN = "{OM Server from "Initialize Data"}"
$MPName = "{Management Pack Name from "Initialize Data"}"
$MPNamespace = "{Management Pack Namespace from "Initialize Data"}"
$GroupName = "{Group Name from "Initialize Data"}"
```

Set trace and status variables to defaults

There are three standard variables that are published from all scripts, which are the ErrorState, ErrorMessage, and Trace variables. ErrorState is an integer that represents success (0), warning

(1), error (2), or critical error (3). ErrorMessage holds the error text when a problem occurs. Trace is a variable where you can append information to help with troubleshooting should an error occur. At the beginning of the script we set these variables to their default values as well as clear the built-in Error variable.

```
$ErrorState = 0
$ErrorMessage = ""
$Trace = ""
$Error.Clear()
```

Validate inputs

All inputs sent to the remote script should be validated to the extent possible, as shown here.

```
        if (($SCOMServerFQDN.length -lt 1) -or ($MPName.length -lt 1) -or
($MPNamespace.length -lt 1)
            -or ($GroupName.length -lt 1))
        {
            Throw "Error: One or more required parameters is Null."
        }
```

Establish PS remote session

Since this script will be making a remote call, establish the remote session with error checking as shown here.

```
$Session = New-PSSession -ComputerName $SCOMServerFQDN
if ($Session -eq $null)
{
    $ErrorMessage =  $Error[0]
    $Trace += "Could not create PSSession on $SCOMServerFQDN"
    $ErrorState = 2
}
```

Execute script in remote session

With a valid session, use the Invoke-Command cmdlet to execute the remote script, passing in any input parameters.

```
else
{
   $ReturnArray = Invoke-Command -Session $Session
   -Argumentlist $SCOMServerFQDN, $MPName, $MPNamespace, $GroupName -ScriptBlock {
    Param ( $SCOMServerFQDN, $MPName, $MPNamespace, $GroupName )
```

Use try/catch/finally

The remote script should be wrapped in a try…catch…finally block to handle any exceptions that occur and to properly close out the Trace variable. A useful variable to create is the Action

variable which holds a string explaining what the script is supposed to be doing. This variable then can easily be referenced as needed when appending information about the execution in Trace. Both examples are shown below.

```
Try
{
    $Action = 'Create OM Group';
    # More Code Here
}
Catch
{
    $Trace += "Exception caught in remote action '$Action'... 'r'n"
    $ErrorState = 2
    $ErrorMessage = $error[0].Exception.tostring()
}
Finally
{
    $Trace += "Exiting remote action '$Action' 'r'n"
    $Trace += "ErrorState:   $ErrorState'r'n"
    $Trace += "ErrorMessage: $ErrorMessage'r'n"
}
```

Append useful data to the Trace variable

As the remote script runs, append information to the Trace variable that will help in debugging issues in the case of failure. At the beginning of the remote script, write out the input variables for easy reference.

```
    $Trace = "Beginning remote action '$Action' 'r'n"
    $Trace += "Parameters: 'r'n"
    $Trace += "  SCOMServerFQDN:  $SCOMServerFQDN 'r'n"
    $Trace += "  MPName:  $MPName 'r'n"
```

Add any required Windows PowerShell modules

The next step is to import any Windows PowerShell modules on the remote server that are necessary to accomplish the desired task.

```
  $Trace += "Importing Operations Manager module 'r'n"
  try
  {
   Import-Module OperationsManager
  }
  catch
  {
     $Trace += "Importing of the OpsMgr module failed as the module was already
present 'r'n"
  }
```

Use throw for common errors

If the script encounters something unexpected that does not throw an error by itself (like getting back a null value from a cmdlet), use the throw keyword to raise exceptions as shown here.

```
$PrimaryMgmtServer = Get-SCOMManagementServer -Name $SCOMServerFQDN

    $Trace += "Checking if primary management server is null or not...`r`n"
    if($PrimaryMgmtServer -eq $null)
    {
       Throw "Can't get the Primary Management Server"
    }
```

Perform core task logic

Write the script to perform the desired task, incorporating Trace best practices as needed.

```
        $group = Get-SCOMGroup -DisplayName $GroupName

        if ($group -eq $null)
        {
            $Trace += "Group '$GroupName' not found. Creating... 'r'n"
            $group = New-Object

Microsoft.EnterpriseManagement.Monitoring.CustomMonitoringObjectGroup($MPNamespace,
                        $GroupName,$GroupName,$formula)
            $mp = Get-SCOMManagementPack -name $MPName
            $mp.InsertCustomMonitoringObjectGroup($group)
        }
```

Set ErrorState and ErrorMessage

At the end of the remote script, if no problems have occurred, set the ErrorState variable to 0. In the case of an exception, inside the catch statement, set the ErrorState to the appropriate value, and then populate the ErrorMessage variable with the exception caught.

```
        $ErrorState = 0   #Return Success
        $Trace += "Completed remote action '$Action'... 'r'n"
    }
    Catch
    {
        $Trace += "Exception caught in remote action '$Action'... 'r'n"
        $ErrorState = 2
        $ErrorMessage = $error[0].Exception.tostring()
    }
```

Return results

The last step in the remote script is to package up the results into the Results variable as an array and return the Results variable to the calling script.

```
    $Results = @($ErrorState, $ErrorMessage, $Trace)
    Return $Results
}
```

Prep data for Orchestrator Publishing

In the local script context, unpackage the results for inclusion on the data bus. If one of the values was an array itself, follow the guidance on handling arrays document in the "Returning arrays" section later in this chapter.

```
$ErrorState = $ReturnArray[0]
$ErrorMessage = $ReturnArray[1]
$Trace = $ReturnArray[2]
```

Close remote session

Lastly, close the remote session to free up resources.

```
Remove-PSSession -Session $Session
```

Putting it all together

The script contained below is the entire script showing the various design elements.

```
$SCOMServerFQDN = "{OM Server from "Initialize Data"}"
$MPName = "{Management Pack Name from "Initialize Data"}"
$MPNamespace = "{Management Pack Namespace from "Initialize Data"}"
$GroupName = "{Group Name from "Initialize Data"}"

$ErrorState = 0
$ErrorMessage = ""
$Trace = ""
$Error.Clear()

if (($SCOMServerFQDN.length -lt 1) -or ($MPName.length -lt 1) -or ($MPNamespace.length -
lt 1)
            -or ($GroupName.length -lt 1))
{
      Throw "Error: One or more required parameters is Null."
}


$Session = New-PSSession -ComputerName $SCOMServerFQDN
if ($Session -eq $null)
{
```

```
    $ErrorMessage =  $Error[0]
    $Trace += "Could not create PSSession on $SCOMServerFQDN"
    $ErrorState = 2
}
else
{
   $ReturnArray = Invoke-Command -Session $Session
   -Argumentlist $SCOMServerFQDN, $MPName, $MPNamespace, $GroupName -ScriptBlock {
        Param ( $SCOMServerFQDN, $MPName, $MPNamespace, $GroupName )

    Try
    {
        $Action = 'Create OM Group';
        $Trace = "Beginning remote action '$Action' 'r'n"
        $Trace += "Parameters:'r'n"
        $Trace += "  SCOMServerFQDN:  $SCOMServerFQDN 'r'n"
        $Trace += "  MPName:  $MPName 'r'n"
        $Trace += "  GroupName:  $GroupName 'r'n"
        $Trace += "'r'n"


           $Trace += "Importing Operations Manager module 'r'n"
           try
           {
           $Path = Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\System Center Operations
Manager\12\Setup"
               $FinalPath = $Path.InstallDirectory +
"\Powershell\OperationsManager\operationsmanager.psd1"
           Import-Module $FinalPath
           }
           catch
           {
                $Trace += "Importing of the OpsMgr module failed as the module was
already present 'r'n"
           }
        $PrimaryMgmtServer = Get-SCOMManagementServer -Name $SCOMServerFQDN

            $Trace += "Checking if primary management server is null or not... 'r'n"
           if($PrimaryMgmtServer -eq $null)
           {
               Throw "Can't get the Primary Management Server"
           }

        $group = Get-SCOMGroup -DisplayName $GroupName
```

```
        if ($group -eq $null)
        {
            $Trace += "Group '$GroupName' not found. Creating... 'r'n"
            $group = New-Object

Microsoft.EnterpriseManagement.Monitoring.CustomMonitoringObjectGroup($MPNamespace,
                        $GroupName,$GroupName,$formula)
            $mp = Get-SCOMManagementPack -name $MPName
            $mp.InsertCustomMonitoringObjectGroup($group)
        }

        $ErrorState = 0   #Return Success
        $Trace += "Completed remote action '$Action'... 'r'n"
    }
    Catch
    {
        $Trace += "Exception caught in remote action '$Action'... 'r'n"
        $ErrorState = 2
        $ErrorMessage = $error[0].Exception.tostring()
    }
    Finally
    {
        $Trace += "Exiting remote action '$Action' 'r'n"
        $Trace += "ErrorState:   $ErrorState'r'n"
        $Trace += "ErrorMessage: $ErrorMessage'r'n"
    }
    $Results = @($ErrorState, $ErrorMessage, $Trace)
    Return $Results
  }
  $ErrorState = $ReturnArray[0]
  $ErrorMessage = $ReturnArray[1]
  $Trace = $ReturnArray[2]

  Remove-PSSession -Session $Session
}
```

Returning arrays

There is a little known fact that you can indeed return more than single-valued variables from a .Net Script activity whose type has been set to Windows PowerShell. The Published Data section of the activity is setup as normal as shown in Figure 5-14, where the variables are defined as needed and there is no option to mark the variable as a collection.

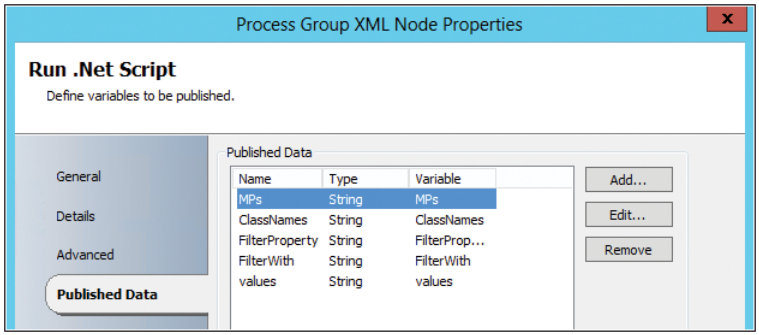


FIGURE 5-14 Published Data in the Run .Net Script activity.

By packaging up the return data into an array, as shown in the following script example, every future activity will be called "n" times, where "n" is the length of the array.

```
[xml]$node = $groupNode

$MPs = @()
$ClassNames = @()
$FilterProperty = @()
$FilterWith = @()

foreach ($classNode in $node.Group.Class)
{
    $MPs += $classNode.MP
    $ClassNames += $classNode.Name
    $FilterProperty += $classNode.FilterProperty
    $FilterWith += $classNode.FilterWith
}
```

In the previous example, four Windows PowerShell arrays were created ($MPs, $ClassNames, $FilterProperty, $FilterWith). These arrays were then populated for each loop using the += operator to add items to each array.

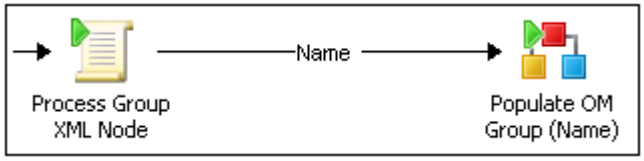


FIGURE 5-15 Multiple activations in a runbook.

If the Process Group XML Node activity shown in Figure 5-15 processed four items in the loop, then the subsequent activity Populate OM Group (Name) would execute four times. If the Windows PowerShell array was generated inside of a remote call, then the array must be repackaged in the local execution context for Orchestrator to correctly handle the array processing, as shown in the following example:

```
$ReturnArray = Invoke-Command -Session $Session -Argumentlist $VMMServerFQDN
-ScriptBlock {
    Param ( $VMMServerFQDN )

    Import-Module VirtualMachineManager

         $VMMServer = Get-SCVMMServer -ComputerName $VMMServerFQDN

    $roleNames = @()
    $roles = Get-SCUserRole
    foreach ($role in $roles)
    {
        $roleNames += $role.Name
    }

    $Results = @($ErrorState, $ErrorMessage, $Trace, $roleNames)
    Return $Results

} #End remote execution

$ErrorState = $ReturnArray[0]
$ErrorMessage = $ReturnArray[1]
$Trace = $ReturnArray[2]
$Names = $ReturnArray[3]

#Reprocessing the array locally for Orchestrator
$RoleNames = @()

foreach ($name in $Names)
{
   $RoleNames += $name
}
```

Runbook patterns

In this section we will discuss three runbook structure patterns, namely component, control, and initiation, as well as design patterns to deal with file-based variables and file-based state for long running processes.

Component runbook pattern

Component runbooks are low-level runbooks that perform a single task without any Orchestrator dependencies other than possibly an IP. In most cases, a component runbook will consist of the following activities:

- Initialize Data activity that sets up all required input parameters.
- .NET Script activity or IP activity that does the task work.
- Two Return Data activities that return the error information or completion information. For more information on setting up runbook data publishing, see the "Publishing data" section in this chapter. For information on what data to return, see the "Error handling" section in this chapter.

The structure for component runbooks is shown in Figure 5-16.

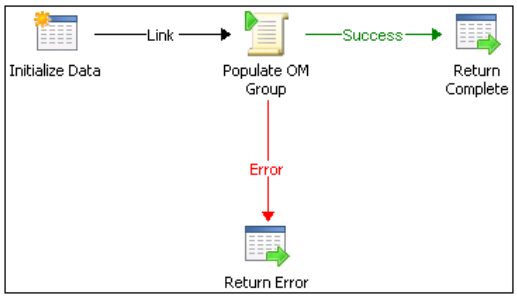
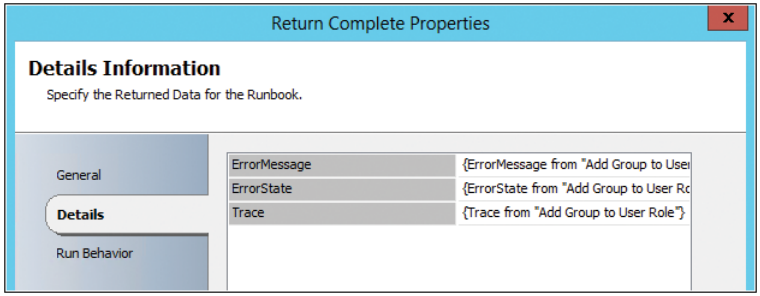


FIGURE 5-16 Component runbook structure.

Rules

The following rules are in place to guide the design.

- The runbook performs a simple task without complex business logic. Business logic is the domain for control runbooks.
- The runbook will declare all inputs with the Initialize Data activity.
- If Windows PowerShell scripts are used, the scripts should follow the Windows PowerShell guidelines on making remote calls, error handling, and error / trace reporting defined in the "Using Windows PowerShell in Orchestrator" section.
- The runbook should publish success / failure by publishing at a minimum, ErrorState, ErrorMessage, Trace Runbook Name, and Activity Name. More information on publishing data can be found in the "Publishing data" section:



- The runbook should have the Log Common Published Data setting active. More information can be found in the "Logging execution data" section.
- The runbook should have Job Concurrency set greater than 1. When it is not possible, it should be noted in documentation for future reference as to the reasons Job Concurrency must be set to 1. More information on job concurrency can be found in the "Setting job concurrency" section.
- The runbook will have no dependencies on Orchestrator global variables. All inputs will be defined on the Initialize Data activity.
- The runbook will have no dependencies on other Orchestrator runbooks.
- The runbook will have no dependencies on additional software being installed on the Runbook server outside of IPs.

Error handling

Error handling in component runbooks consists of publishing data back to the calling runbook using the Return Data activity. The error information returned should consist of, at a minimum, the information listed in Table 5-1.

TABLE 5-1 Summary of Error Handling for Component Runbooks

NAME	DESCRIPTION
Error State	A numeric value indicating success (0), warning (1), error (2), or critical error (3).
Error Message	A string containing the error or warning message, or blank if the execution was successful.
Trace	A string containing the execution flow for a Windows PowerShell script, or any other additional data for other types of activities that can help narrow down the problem.
Runbook Name	A string containing the name of the runbook (common published data).
Activity Name	A string containing the name of the activity that had the problem or blank if successful (common published data).
Activity Start Time	The start time of the activity (common published data). Can be left blank if successful.

Validation of input parameters

All scripts should validate that the required parameters are not null and that the values are valid. Within component runbooks, the validation takes place within the same activity that hosts the main script.

For each script, add the following code snippet and use the variables according to your script requirements:

```
if (($Variable1.length -lt 1) -or ($Variable2.length -lt 1)
     -or ($Variable3.length -lt 1))
```

```
{
            Throw "Error: One or more required parameters is Null."
}
```

For value validations, the following sample scripts can be used.

RANGE VALIDATION (1-12):

```
if (!($variable1 -match "\b(?:1[0-2]|[1-9])\b"))
{
    Throw "Error: parameter is not within range."
}
```

ENUM VALIDATION (BLUE, RED):

```
if (!($variable1 -match "\bblue\b|\bred\b"))
{
    Throw "Error: parameter does not have a valid value."
}
```

EMAIL ADDRESS VALIDATION:

```
if (!($variable1 -match "^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$"))
{
    Throw "Error: parameter is not a valid email address."
}
```

DATE VALIDATION:

```
if (!($variable1 -match "^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12]
[0-9]|3[01])$"))
{
    Throw "Error: parameter is not a valid date."
}
```

IP ADDRESS VALIDATION:

```
if (($test -match "\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4]
[0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4]
[0-9]|[01]?[0-9][0-9]?)\b"))
{
    Throw "Error: parameter is not a valid IP address."
}
```

Control runbook pattern

The control runbook pattern encapsulates the business logic needed and the overall workflow required to perform a complex task. Control runbooks are composed of:

- An Initialize Data activity

- Optionally, one or more component runbooks
- Optionally, one or more control runbooks since control runbooks can be nested
- Error routine runbooks in case of error
- Other Orchestrator activities

In the example shown in Figure 5-17, in addition to calling multiple component runbooks, there are script activities that read business logic for the process from an XML file. Because these scripts are directly tied to the business logic for the control runbook, it doesn't make sense to encapsulate them into component runbooks.

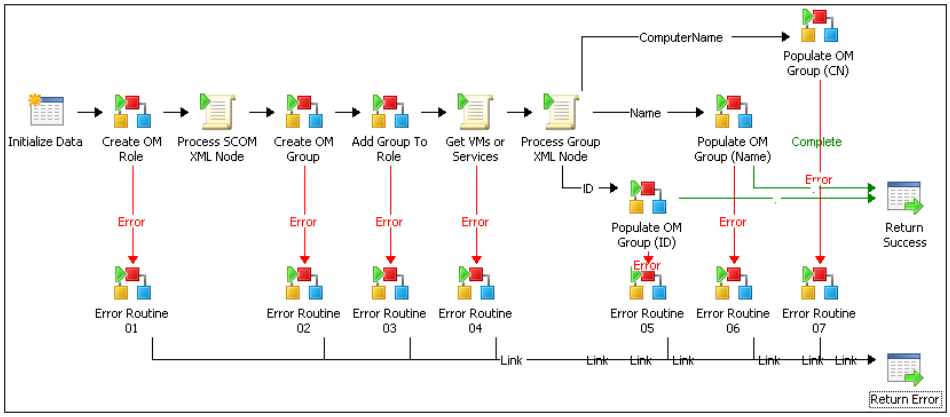


FIGURE 5-17 Control runbook design patternexample.

Component runbooks are executed using the Invoke Runbook activity where the Wait For Completion property is checked, as shown in Figure 5-18. This allows the control runbook to process the output of the child runbook, including handling exceptions.

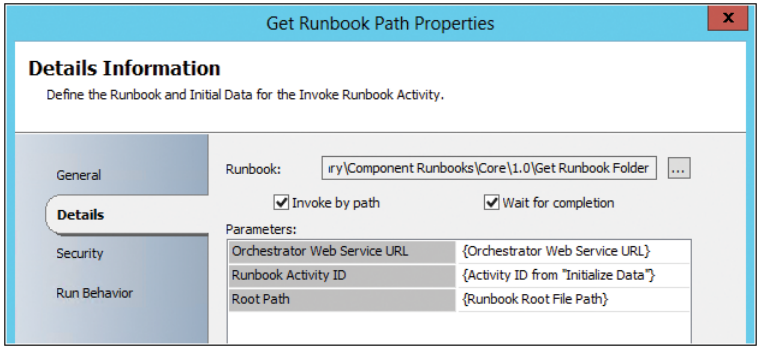


FIGURE 5-18 Setting the Invoke Runbook activity for component runbooks.

Rules

The following rules are in place to guide the design.

- The control runbook stitches one or more component runbooks together to perform a complex task.

- The runbook can have dependencies on variables. For more information on variables, refer to the "File-based runbook variables" section.
- The runbook should have dependencies on component runbooks.
- The runbook will perform error handling by utilizing a component error handling runbook.
- The runbook will publish an overall ErrorState of success / failure.
- The runbook should have no dependencies on additional software being installed on the Runbook server outside of IPs.
- If Windows PowerShell scripts are used, the scripts should follow the Windows PowerShell guidelines on making remote calls, error handling, and error / trace reporting defined in the "Using Windows PowerShell in Orchestrator" section.
- The runbook should start with an Initialize Data activity to handle any required inputs. Runbooks that start via a Monitor activity are handled as Initiation runbooks.

Error handling

Error handling in control runbooks consists of calling an error routine runbook, as shown in Figure 5-19.

FIGURE 5-19 Calling an error routine runbook inside a control runbook.

Optionally, the runbook can publish data back to a calling runbook using the Return Data activity as shown in Figure 5-20.

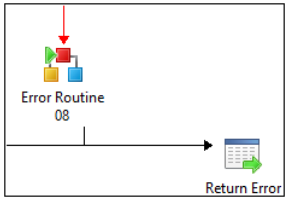


FIGURE 5-20 Returning error information to a calling runbook.

The error information returned, when published, should consist of, at a minimum, the information listed in Table 5-2.

TABLE 5-2 Summary of Error Handling for Control Runbooks

NAME	DESCRIPTION
Error State	A numeric value indicating success (0), warning (1), error (2), or critical error (3).
Runbook Name	A string containing the name of the runbook (common published data).
Activity Name	A string containing the name of the activity that had the problem or blank if successful (common published data).
Activity Start Time	The start time of the activity (common published data). Can be left blank if successful.

Validation of input parameters

At the control runbook level, it is important to check if all required parameters are passed into the runbook are valid and that any process prerequisites are checked. An example is shown in Figure 5-21.

The second activity in each control runbook should be a validation script activity that will check if all required parameters are passed in and that the values are not null. Additionally, a range check can be used where appropriate.

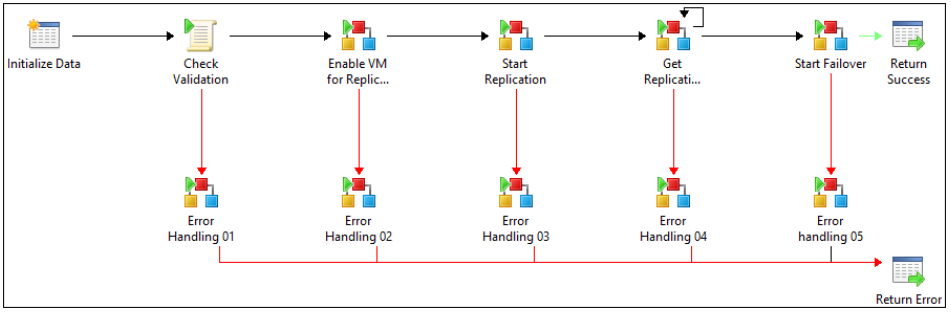


FIGURE 5-21 Validation inside of a control runbook.

The next activity is a "Check Prerequisites" component runbook or script. This is optional, and is required only if prerequisites are defined. Depending on the logic behind the prerequisites check, it might be a component runbook (if complex) or be part of the main control runbook. This depends on the complexity of the validation and if it can be reused in other scenarios as well.

Connectivity runbook

It's also important to check whether all hosts that are required for a specific scenario are reachable and able to be connected to. For example, when creating a cluster it's important to be able to connect to all hosts that will form the cluster. For instance, it would be wasteful to

run the entire process for creating the cluster and installing features only to discover that one or more of the required nodes couldn't be reached.

This form of validation is only required when modeling scenarios (control runbooks) that involve interacting with more than one host; therefore, control runbooks involving only a single host do not require this type of validation. For each script, a connectivity check will be implemented and therefore a check on control runbook level is not required.

The implementation is a component runbook that takes a single parameter (list of hosts or a single host) which will be evaluated. The script will publish a couple of variables that help to define what to do depending on the output.

The variables used in the script are:

- **$isReachable** A Boolean that is set to True if all hosts are reachable. If a single host is not available this variable is set to False
- **$serverResultList** An array of servers with a boolean set to True if available, otherwise set to False. (Output is server1,true;server2,false;server3,true)
- **$unreachableServerCount** The number of servers unreachable (so if you pass in five servers to the component runbook, and one out of five is not reachable, then the value is set to 1)
- **$reachableServerCount** The same as for $unreachable (using the same example this value would be set to 4).

The script itself is as follows:

```
#Inputs from published data
$Hosts= vmm01;sco01;sco02"

$HostList = @()
$HostList = $Hosts.split(",",[stringsplitoptions]::RemoveEmptyEntries)

$ErrorState = 2
$ErrorMessage = ""
$Trace = ""
$Error.Clear()

# ------------------------------------------------------------
Function TestPSRemoting {
        Param ( [string]$HostName )

       $ErrorState = 0
       $ErrorMessage = ""
       $Trace = ""
       $Error.Clear()

       Try
```

```
        {
            $IPAddressFromDNS = [System.Net.Dns]::GetHostAddresses($HostName)
            $Trace += $IPAddressFromDNS + " 'r'n"
            $Trace += "Attempting Invoke-Command -ComputerName $HostName `r`n"
            $ErrorActionPreference = "Stop"
            $Result = Invoke-Command -ComputerName $HostName { 0 }
            If ($Result -eq 0) {
                $Trace += "Successfully connected to $HostName 'r'n"
                $ErrorState = 0
            }
        }
        Catch
        {
            $ErrorState = 2
            $ErrorMessage = $error[0].Exception.ToString()
        }
        Finally
        {
            $Trace += "ErrorState:     $ErrorState 'r'n"
            $Trace += "ErrorMessage:   $ErrorMessage 'r'n"
               #write-host $ErrorState
        }
        $ResultArray = @($ErrorState, $ErrorMessage, $Trace)
        Return $ResultArray
}
#-------------------------------------------------------------

$Trace = "Beginning BatchTest-PSRemoting... 'r'n"
$Trace += "'r'n"
$Trace += "Host(s):   $Hosts'r'n"
$Trace += "'r'n"

$serverResultList = @()
$ResultList = @{}
$unreachableServerCount=0
$reachableServerCount=0
$isReachable = $true

foreach($x in $HostList)
{
    $ResultList = TestPSRemoting($x)
    if($ResultList[0] -ne '0')
    {
        $isReachable = $false
        $unreachableServerCount++
```

```
        $serverResultList+="$x,false"
    }
    else
    {
        $reachableServerCount++
        $serverResultList += "$x,true"
    }
}
```

Initiation runbooks

The sole purpose of initiation runbooks is to call control runbooks based on a trigger condition that is initiated by one of the various Orchestrator monitor activities. The execution of the control runbook should not wait for completion and therefore should not provide any error handling except for reporting errors with the Monitor activity itself if required.

The example shown in Figure 5-22 consists of a Monitor Date/Time activity that when triggered (once per hour in this case), calls the Get VMM Roles control runbook, not waiting for completion.

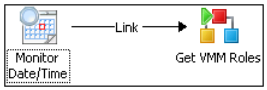
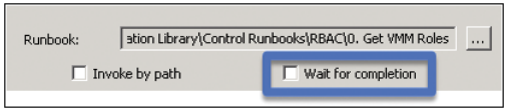


FIGURE 5-22 A simple initiation runbook.

Rules

The following rules are in place to guide the design.

- The runbook should start with one of the monitor activities followed by a call to a control runbook. The exception to this rule is when the initiation is controlled by System Center Service Manager service requests. This is detailed in the "Service requests initiation runbooks" section later in this chapter.
- The runbook should execute the control runbook without waiting for completion.



- The runbook should only provide error handling if the Monitor activity needs to report on errors.

Error handling

The only time error handling should be added to an initiation runbook is when the Monitor activity is not based on date/time and itself could throw an exception. An example would be the Monitor Object activity from the System Center Service Manager IP.

Service requests initiation runbooks

To properly implement service requests in Service Manager, the request needs to publish the runbook activity ID to a property on the runbook's Initialize Data activity. This way, if an error occurs, the runbook can set the runbook automation activity to a failed status so that the default error handling will not break the success / failure logic inherent in Service Manager. Figure 5-23 shows an example of this.

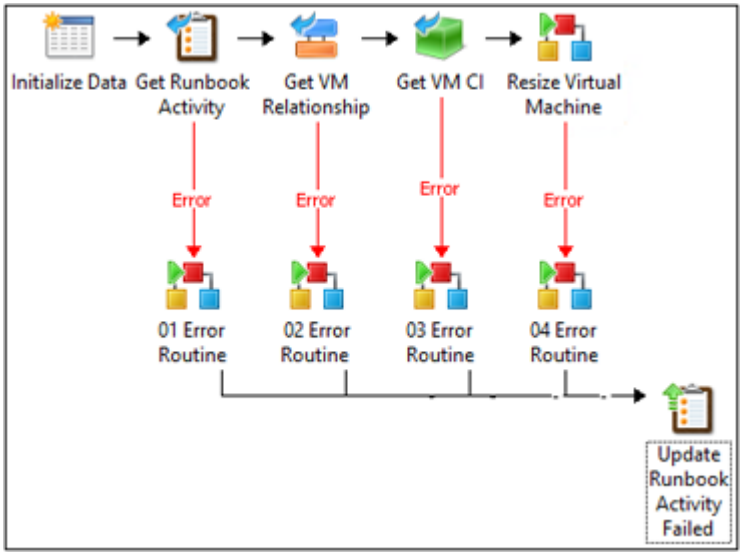
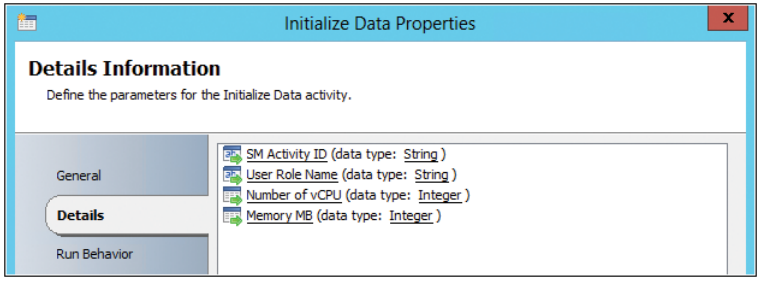


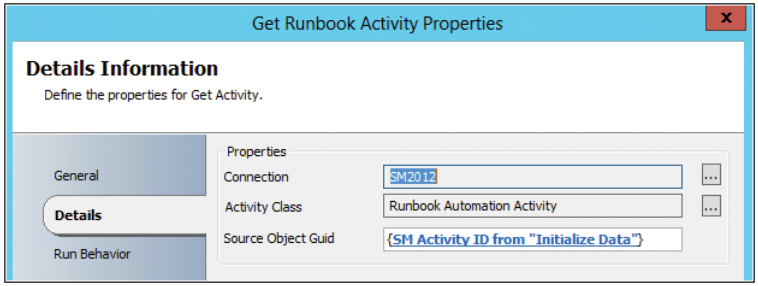
FIGURE 5-23 An example of an initiation runbook to handle a Service Manager service request.

The example in Figure 5-23 is broken down into the following parts:

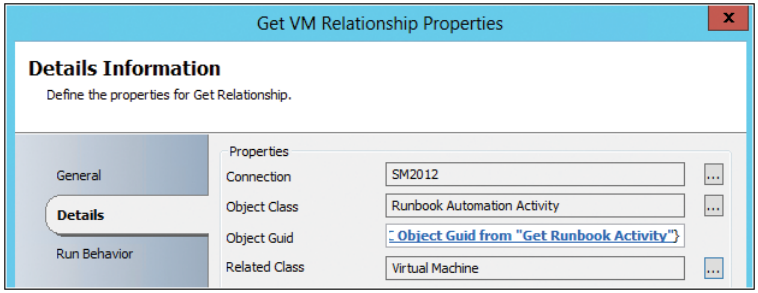
1. The first activity is the Initialize Data activity, where we have defined the SM Activity ID property as well as any other required inputs.

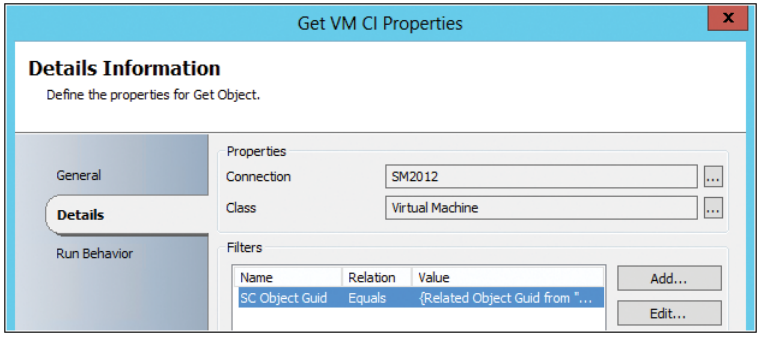


2. The second activity retrieves the Service Manager runbook automation activity that initiated this call using the passed in SM Activity ID.

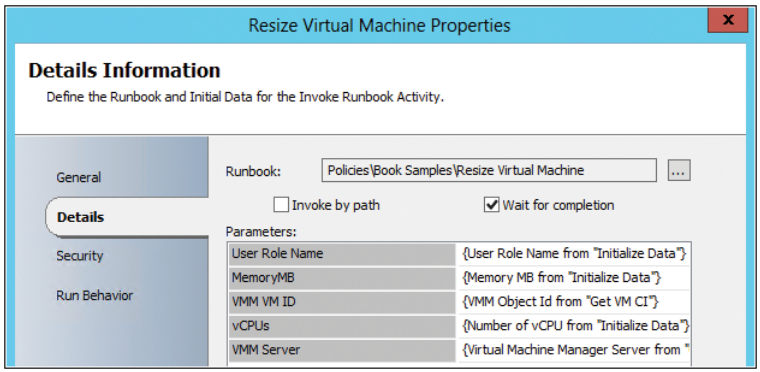


3. The next two activities in this case are getting any associated configuration items that have been added to the Service Manager runbook automation activity. In the previous example, we are getting an associated virtual machine configuration item by first getting relationships of type virtual machine and then getting the actual virtual machine item.

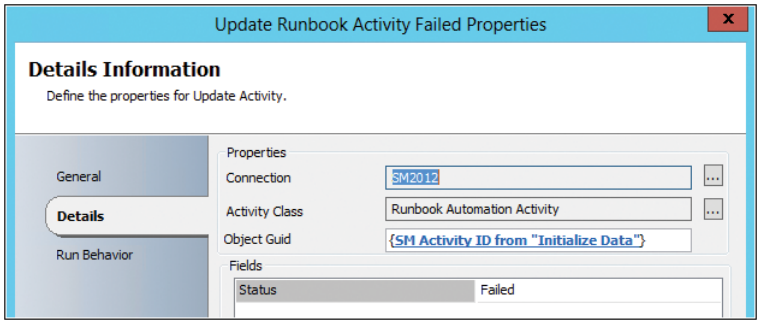




4. Finally, we are calling our control runbook which in the previous example, resizes the virtual machine. In this case we set our Wait For Completion checkbox to checked.



5. If any of the activities fail, we are calling our component error runbook and then we are setting the Service Manager runbook automation activity to a failed status.



File-based runbook variables

Orchestrator global variables present a number of challenges, especially when dealing with a runbook library concept as opposed to a one-off development implementation. These challenges include:

- Lack of visibility into runbook dependencies. There is no way to know if a variable value is changed, what runbooks are affected by that change.
- Runbook exports always export all global variables, used or not for the scope of the export. There are CodePlex solutions to strip the unused variables out of export files but this is not supported by Microsoft.
- Multivalued variables are not available. CSV values can be created but are not user-friendly to read or edit.
- In practice, most "global" variables truly only apply to one or a select few runbooks.

Your runbook automation library (Automation Library) should define a root path for all files in the Orchestrator global variable Runbook Root File Path. For single server deployments, this

variable can be set to a local directory. In multiserver deployments, this variable should be set to a network file share that is accessible to all runbook servers. A suggested location would be placing a share on the Orchestrator database server but is by no means a hard requirement. The Automation Library should also define the URL to the Orchestrator REST API in the Orchestrator global variable Orchestrator Web Service URL. These two values are Orchestrator global variables because they will be used by all runbooks that need access to their file-based runbook variables, as well as any runbook that implements the file-based state pattern later in this chapter.

The Automaton Library will also contain two component runbooks that are used in the patterns. Details of how to build these runbooks is included in the Appendix. The first is the Get Runbook Folder runbook which determines the file path for a runbook based on its location in the Orchestrator folder hierarchy. The second runbook created is called Get Relative Folder and it takes in a starting directory and, with the use of some additional parameters, can return a relative path as needed. The suggested locations of these items are listed in Table 5-3.

TABLE 5-3 Suggested Folder Locations

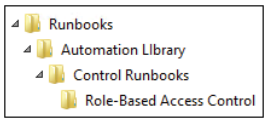
ITEM	TYPE	LOCATION
Runbook Root File Path	Variable	Global Settings\Variables\Automation Library
Orchestrator Web Service URL	Variable	Global Settings\Variables\Automation Library
Get Runbook Folder	Runbook	Runbooks\Automation Library\Component Runbooks\Core
Get Relative Folder	Runbook	Runbooks\Automation Library\Component Runbooks\Core

Local runbook variables

In an attempt to address these issues, the Automation Library can use a pattern of file-based, runbook local variables. To share variables for a complex task across multiple runbooks, see the "Shared runbook variables" section. If the variable in question is truly global in nature, you can utilize Orchestrator global variables.

To use the pattern, create a text file to hold your control runbook settings, preferably in the XML format. Place the file in the logical place on the file system as follows:

1. Start with a root path, that is, C:\Runbooks
2. Create the folder structure using the Automation Library \ Control Runbooks tree. The leaf folders of the folder structure will be the runbooks themselves.
3. Only runbooks that need to read from file-based global variables, or perform file-based state operations need to have a folder on the file system.
4. Create your XML file in the proper runbook file folder.



In the control runbook, after the initialize data activity, place an Invoke Runbook activity on the design surface and use the Get Runbook Folder runbook as shown in Figure 5-24.

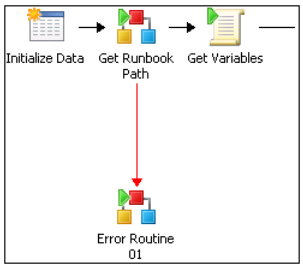


FIGURE 5-24 Using the Get Runbook Path runbook.

This runbook has three inputs, as shown in Figure 5-25 and described in Table 5-4.

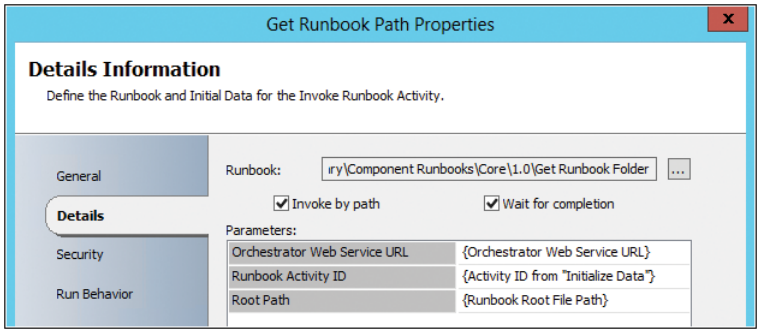


FIGURE 5-25 The settings for the Get Runbook Path runbook activity.

TABLE 5-4 Inputs for the Get Runbook Path Runbook Activity

INPUT	VALUE
Orchestrator Web Service URL	Global Settings\Variables\Automation Library\ Orchestrator Web Service URL
Runbook Activity ID	Published Data: Activity ID from Initialize Data
Root Path	Global Settings\Variables\Automation Library\ Runbook Root File Path

This runbook will use the inputs to retrieve the logical file location for a runbook, publishing the data to the Runbook Path data bus variable. Next, add a Run .Net Script activity to read in a file at the Runbook Path location and publish the results. In the following example, the script activity reads from a Runbook Variables.XML file and outputs the values to the data bus. The XML file was set up as follows:

```
<xml>
   <XMLRulesEngine>myEngine.xml</XMLRulesEngine>
</xml>
```

The Run .Net Script activity is set up as shown in Figures 5-26 and 5-27.

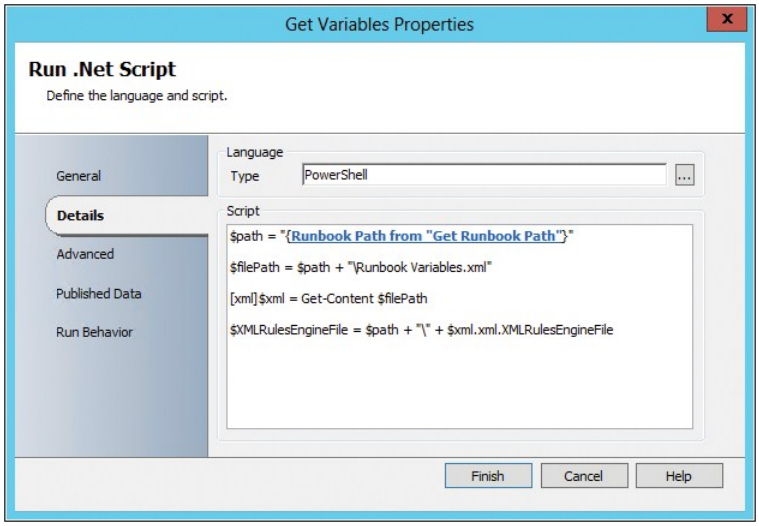


FIGURE 5-26 The Get Variables .Net Script Activity script pane.

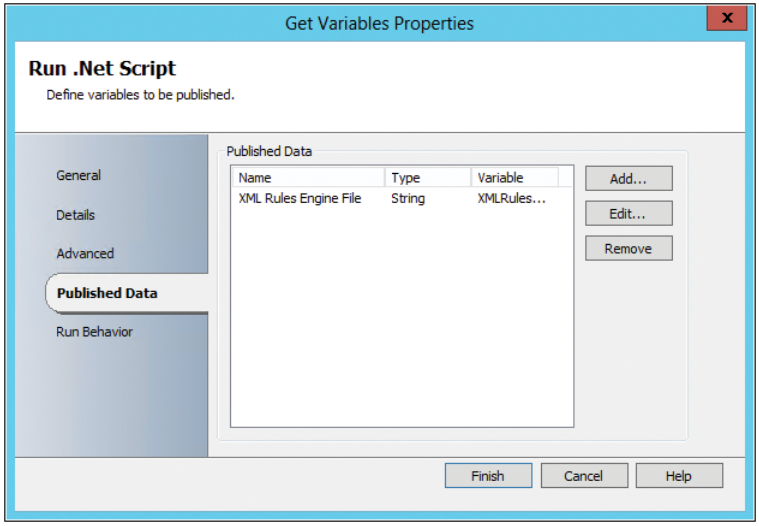


FIGURE 5-27 An example of theThe Get Variables .Net Script Activity published data pane.

Now subsequent activities can use these values as needed.

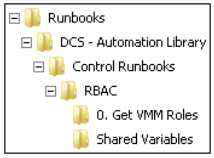
Shared runbook variables

The simplest way to share variables for a complex task is to create an outer control runbook that reads in all the variables from its file and then passes in those variables to all the child control runbooks. Use the Local runbook variables pattern above and have the outer parent runbook contain all the shared variables in its local variable file.

There are also times where a group of runbooks needs to share variables where it may not be convenient for a parent runbook to read them in and pass them on to the child runbooks. In this case, shared file-based variables can be used. This pattern extends the local file-based variables pattern by introducing a second core component runbook that will allow relative-position directory navigation so that a shared location can be found and accessed.

To use the pattern, create a text file to hold your control runbook settings, preferably in the XML format. Place the file in the logical place on the file system as follows:

1. Start with a root path, that is, C:\Runbooks
2. Create the folder structure using the Automation Library \ Control Runbooks tree. The leaf folders of the folder structure will be the runbooks themselves.
3. Only runbooks that need to read from file-based global variables, or perform file-based state operations, need to have a folder on the file system.
4. Create a directory to hold your shared variable file. The directory should be created within the scope of the control runbook theme, that is, Shared Variables under RBAC.



5. Create your XML file in the folder.

In the control runbook, after the Initialize Data activity, place an Invoke Runbook activity on the design surface and use the Get Runbook Folder runbook setup as in the previous section. Next, add another Invoke Runbook activity and point it to the Get Relative Folder runbook, as shown in Figure 5-28.

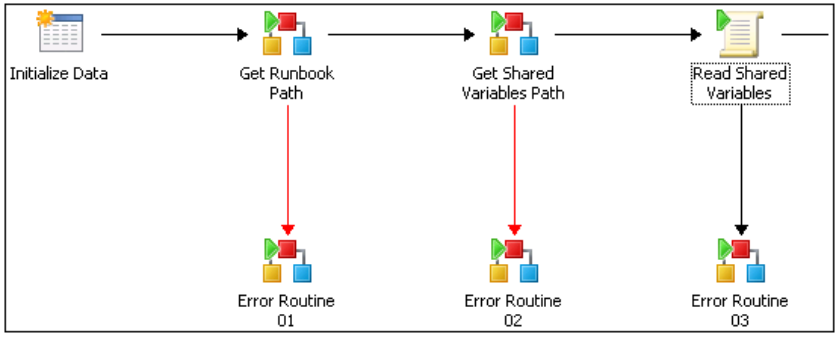


FIGURE 5-28 Adding the Get Relative Folder runbook for shared variable access.

This runbook has three inputs, as shown in Figure 5-29 and described in Table 5-5.

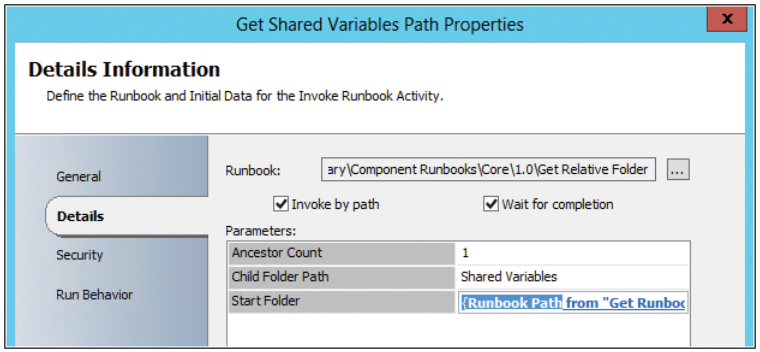


FIGURE 5-29 Setting the parameters for the Get Relative Folder runbook.

TABLE 5-5 Parameters for the Get Relative Folder Runbook

INPUT	VALUE
Start Path	The runbook path as returned by Get Runbook Path
Ancestor Count	The number of parent directories to traverse
Child Path	The path to navigate down to after traversing the parent directories

This runbook will uses the inputs to retrieve the shared file location for a runbook, publishing the data to the Directory Path dat abus variable. Next, add a Run .Net Script activity to read in a file at the Directory Path location and publish the results as in the preceding section.

File-based state pattern

The state patterns are very similar to the file-based variables patterns as there are two patterns to consider. The first pattern deals with local state, meaning state that is tracked within a single runbook. The other state pattern deals with shared state, where multiple runbooks share the state information to complete a complex task. Both patterns use the same core components that are described in the "File-based runbook variables" section. The Get Runbook Folder component runbook will return the path that should be used for the local state pattern and the combination of that runbook plus the Get Relative Folder component runbook will allow the state to be saved in a shared location relative to where the control runbook is located.

File-based state relies on being able to create, read, and update information in a file. While any file schema can be used for state, this guide will focus on using XML files as described in the "Working with XML in Windows PowerShell" section that follows.

Working with XML in Windows PowerShell

Windows PowerShell makes scripting with XML simple and straightforward. This section will review some simple patterns for working with XML including creating an XML file, both exclusive and non-exclusive reading and writing, how to wait for a locked file and how to delete an XML file.

CREATING THE FILE

The following script example shows how to create an XML file using Windows PowerShell by creating a string template, loading in as XML, and then setting the values.

```
$path = "{Directory Path from "Get Shared State Path"}"
$filePath = $path + "\SharedState.xml"
$template = "<xml><Request ID='' Status='' StatusText=''/></xml>"
[xml]$xml = $template
$xml.xml.Request.ID = 'SR12345'
$xml.xml.Request.Status = '0'
$xml.xml.Request.StatusText = 'Submitted'
$xml.Save($filePath)
```

NON-EXCLUSIVE READ/WRITE

In situations where file read/writes happen sequentially or are limited to a single process, the following script example can be followed.

```
$path = "{Directory Path from "Get Shared State Path"}"
$filePath = $path + "\SharedState.xml"
[xml]$xml = Get-Content '$filePath'
$xml.xml.Request.Status = '1'
$xml.xml.Request.StatusText = 'Approved'
$xml.Save('$filePath')
```

EXCLUSIVE READ/WRITE

In situations where there is the possibility for concurrent file writes, all write operations should be done with the file locked for exclusive access. The following script uses the .NET System.IO classes to accomplish this.

```
$path = "{Directory Path from "Get Shared State Path"}"
$filePath = $path + "\SharedState.xml"
[System.IO.FileStream]$file = [System.io.File]::Open($filepath, 'Open', 'ReadWrite',
'None')
[xml]$xml3 = New-Object System.Xml.XmlDocument
$xml3.Load($file)
$xml3.xml.Request.Status = '6'
Read-Host
#rewind the stream the beginning to overwrite
$file.Position = 0
$xml3.Save($file)
$file.Close()
```

FILE OPEN PATTERN FOR FILE LOCKS

In situations where exclusive write access to a file is needed, reading the XML and handling possible lock issues is a bit more complicated. The following example shows how this can be accomplished with looping and a try/catch block.

```
$path = "{Directory Path from "Get Shared State Path"}"
$filePath = $path + "\SharedState.xml"
$accessed = $false
$tries = 0
while (!$accessed -and $tries -lt 10)
{
  try
  {
    [System.IO.FileStream]$file = [System.io.File]::Open($filePath, 'Open', 'ReadWrite',
'ReadWrite')
    [xml]$xml3 = New-Object System.Xml.XmlDocument
    $xml3.Load($file)
    $accessed = $true
    #Do Work Here
    $file.Close()
  }
  catch
  {
    $tries++
    sleep(5)
  }
}
```

DELETE FILE

In situations where exclusive write access to a file is needed, deleting the XML and handling possible lock issues is a bit more complicated. The following example shows how this can be accomplished with looping and a try/catch block.

```
$path = "{Directory Path from "Get Shared State Path"}"
$filePath = $path + "\SharedState.xml"
$accessed = $false
$tries = 0
while (!$accessed -and $tries -lt 10)
{
  try
  {
    [System.IO.FileStream]$file = [System.io.File]::Open($filePath, 'Open', 'Read',
'None')
    [xml]$xml3 = New-Object System.Xml.XmlDocument
    $xml3.Load($file)
```

```
    $accessed = $true
    #Do Work Here
    $file.Close()
    [System.io.File]::Delete($filePath)
  }
  catch
  {
    $tries++
    sleep(5)
  }
}
```

Local state pattern

In the local state pattern, all interaction with the state file(s) happens within a single control runbook execution. An example would be a process where the control runbook asynchronously starts a series of child runbooks and then waits for notification (via state files) that the process has completed. This is shown in Figure 5-30. Although there are multiple runbooks, the calling runbook can share the state file location with the child runbooks.

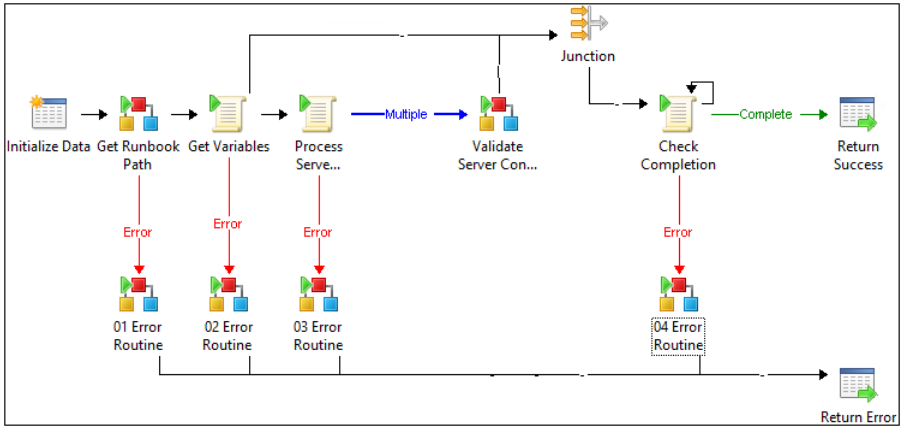


FIGURE 5-30 A local state pattern example.

Shared state and routing/engine patterns

In the shared state pattern, multiple disconnected runbooks need to access the state of a long-running process that does not have an overarching control runbook that executes for the entire duration of the process. This pattern is important when the design requires easy restart ability from a state which is not a rerun from the beginning scenario. An example would be a series of runbooks to handle virtual machine provisioning workflows where the following occur:

- A process creates the request (state).

- Another process monitors for requests that are in particular states and hands them off to the scheduling engine.
- The scheduling engine looks at the current state and hands off the request to the proper child process. Example child processes include create virtual machine, install SCOM agent, add local administrators, and so on.
- The child process updates the state during execution.
- The process monitor then repeats until the request is in the finished or failed state.

The routing / engine pattern uses the shared state pattern to work on a long-running process and make it easier for the process to be restarted at the point of failure instead of having to restart at the beginning of the process. Virtual machine (VM) provisioning is a good example of where this pattern can be applied since there may be many steps involved beyond the interaction with VMM to kick off the VM provisioning for activities such as Active Directory computer account set up, System Center agent installs, etc.

In the diagram in Figure 5-31, there are five folders that make up the folder structure for the routing/engine pattern. The VM Provisioning Request folder is the parent shared state folder in this example and contains four other folders that will house the request XML files as they are processed. The Active folder is for requests that need to be processed through the engine. The Completed folder is for requests that have completed the overall process. The Failed folder is for requests that could not be completed and the In Process folder is for requests that are actively being worked on by a runbook.

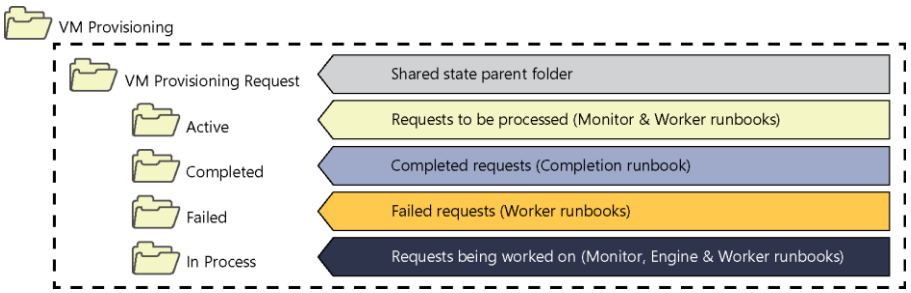


FIGURE 5-31 A Shared state folder example for VM provisioning.

In this pattern there are four types of runbooks, namely a monitor runbook, an engine runbook, multiple worker runbooks, and lastly a completion runbook. The monitor runbook looks for requests from the Active folder, moves the request to the In Process folder, and hands off each request to the engine runbook. This is shown in Figure 5-32.

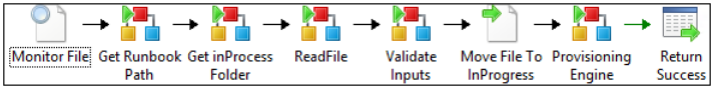


FIGURE 5-32 A monitor runbook example.

The engine runbook examines the status in the file, updates the status to Pending for the next scheduled task, and hands the request off to the worker runbook to process. This is shown in Figure 5-33.

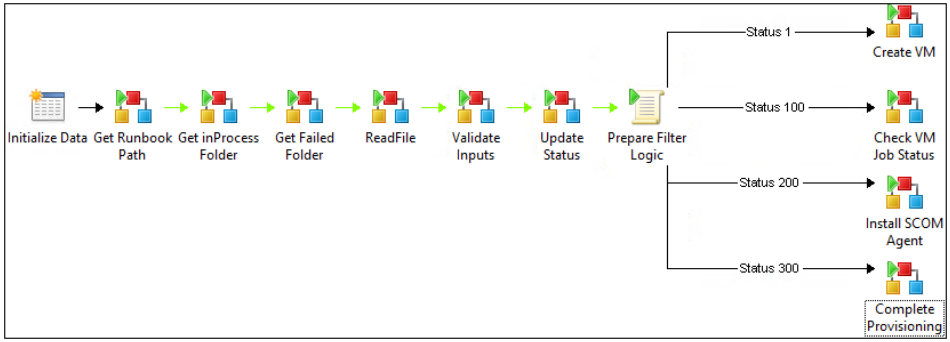


FIGURE 5-33 An engine runbook example.

The worker runbook processes the request and updates the status to either Step Completed or Step Failed. If the step completed, the worker process moves the request back to the Active folder to start the process again. If the worker runbook failed the request, it moves the request to the Failed folder for operator triage. This is shown in Figure 5-34

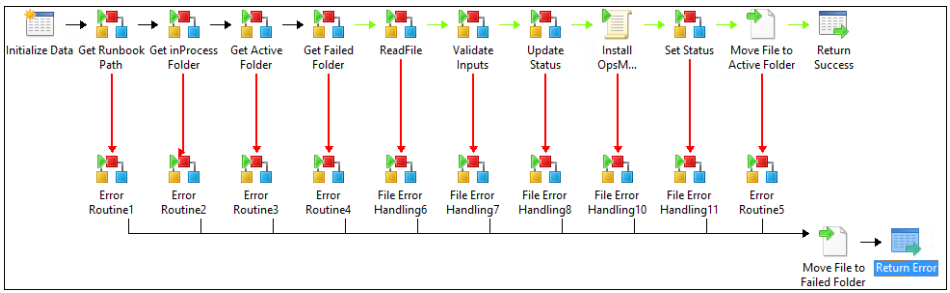


FIGURE 5-34 An example of a worker runbook.

If the problem can be fixed, the operator edits the request to reset the status to the last known good status and moves the file back into the Active folder for further processing. When all work has been completed, the engine runbook hands the request off to the completion runbook which in turn updates the status to the final completion status and moves the request to the Completed folder. This is shown in Figure 5-35.

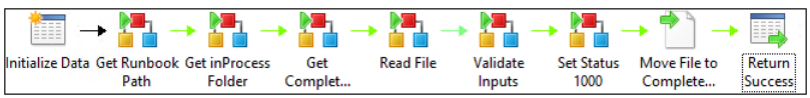


FIGURE 5-35 An example of a completion runbook.

When the monitor runbook hands requests off to the engine runbook, the runbook activity is setup with the wait for completion flag off so that multiple requests can be serviced simultaneously. Likewise, when the engine runbook hands off the request to a worker runbook, the runbook activity is setup with the same setting, that is, fire and forget so that multiple requests can be worked on in parallel. In both cases, the degree of parallelism is controlled by the job concurrency setting for each runbook. Once the concurrency limit has been reached, subsequent executions will be queued until a run slot opens up. For more information on setting the job concurrency, see the "Setting job concurrency: section.

CHAPTER 6

Modular runbook example

This chapter presents an example of process automation using the framework described in previous chapters. It illustrates how the modularity prescribed in the runbook framework enables you to build an increasingly valuable library of runbooks and how the time you invest now will help you to do more in the future with less work.

This chapter builds on the previous chapters, using various patterns and scripts to illustrate a complete virtual machine provisioning process. Using this example, you will learn the basic information you need to apply the runbook framework to whatever automation solution you need to build.

Requirements

The provisioning of a new virtual machine (VM) is a time-consuming process that requires the routing pattern for implementation. The length of time involved will, of course, be determined by your requirements but even just a single instance of this process can take up to 25 minutes. It is not a good idea to have individual Microsoft System Center Orchestrator 2012 runbooks execute for an extended period of time because each runbook server has a fixed amount of runbooks that it can execute (50 by default, which can be increased but will typically be between 50 and 200).

For VM provisioning, some of the more time-consuming tasks, such as copying the VM image to the target host server, can take an extended amount of time. If a runbook is running and waiting for a task to complete, it is unnecessarily consuming Orchestrator resources. A better approach for long-running operations is to have a runbook trigger the operation, but then exit the runbook and have a separate runbook (that is, the routing pattern) periodically monitor that progress and continue executing the overall process once the long running steps have completed. The routing pattern enables a long-running process to be subdivided into component runbooks with the routing runbook managing the state and execution of the process.

As outlined in previous chapters, the process to be orchestrated and the required actions, operations, or approvals should all be optimized and mapped prior to building Orchestrator runbooks.

For the example outlined in this chapter, the following virtual machine provisioning process will be orchestrated:

- The process will be initiated by a file monitor looking for new VM provisioning request files in a particular location

- Input variables will be read from the file and validated
- A routing engine runbook will control state and execution flow through several sequential steps
- Operations—such as creating a VM, monitoring status of the VM deployment process, and post-installation tasks—are automated using runbooks
- On completion, the request is completed and success status is returned

A System Center Virtual Machine Manager (VMM) template is used to create a new VM. The VM template is either created dynamically at run time by the runbook, called a *dynamic template*, or an existing VMM template from the VMM library can be utilized, called a *static template*.

Using the routing pattern described in earlier chapters, all required data for the initiation and state management of the VM provisioning request is stored in an XML-formatted file. This file contains all the related data such as the name of the new virtual machine, the template name, the name of the domain, and a process status field. The status field is updated as each step of the automation is performed so that the status represents the current state of the process execution. This file will be read up by a monitor runbook and process flow and execution will be determined based on the value of the status. This enables decisions to be made, such as whether to proceed or fail with an error message, based on the status written to the file by each step. This is a method of state management for the process. If a step in the process should fail, the status field will indicate the last successful step so that after troubleshooting, the automation can resume from the previous successful step. Some of the process steps include creating a template, receiving the host rating from VMM, creating a VM, and provisioning a System Center Operations Manager agent to the new VM. With each change in the status, an integer number is incremented representing the progress of the overall process.

If you use this approach, you will be able to extend the process later on to include additional functions such as provisioning storage upfront or installing more software to the new VM.

VM provisioning input XML file

An XML-formatted file is used to trigger a new instance of provisioning a new VM. This file contains all the relevant and required data for the process. A simple component runbook will read the XML file and publish all data to the data bus.

The following example of an XML-formatted file uses the RequestID GUID as the filename and the Status element is used to track progress and make sure the steps are called in the right sequence.

```
<xml>
  <RequestID>b7c87df1-4e93-4e7b-9a20-8309bb63bc16</RequestID>
  <Status>0</Status>
```

```
  <VMMServerName>VMM01</VMMServerName>
  <VMName>newVirtualMachine01</VMName>
  <VMDomain>contoso.com</VMDomain>
  <UserRoleName>Administrator</UserRoleName>
  <VMMAdministratorRoleName>Administrator</VMMAdministratorRoleName>
  <ProvisioningType>Windows</ProvisioningType>
  <CloudName>testcloud</CloudName>
  <IsWorkgroup>false</IsWorkgroup>
  <LocalAdminUser>Administrator</LocalAdminUser>
  <VMMTemplateType>Dynamic</VMMTemplateType>
  <HardwareProfileID>e4a4de67-fe1e-4fa9-b22e-abb5babd9c5f</HardwareProfileID>
  <OSProfileID>12313691-6847-45cf-8469-766f5183025b</OSProfileID>
  <OSVHDID>519d2a02-f79b-42fe-ad0d-d591db388870</OSVHDID>
  <NetworkName>Public</NetworkName>
  <VLANNumber>1067</VLANNumber>
  <ExistingTemplateID>773fc727-c4c5-498a-a1fe-030cb912bbc4</ExistingTemplateID>
  <SCOMMgmtServerFQDN>OpsMgr01.contoso.com</SCOMMgmtServerFQDN>
  <Error>
    <ErrorState></ErrorState>
    <ErrorMessage></ErrorMessage>
    <ActivityName></ActivityName>
    <RunbookName></RunbookName>
    <Trace></Trace>
    <RequestID></RequestID>
  </Error>
  <DataVHDs>
    <DataDisk Quantity="2" ID="4f7e5061-5e6b-4068-8103-5b558a829fdb" />
    <DataDisk Quantity="2" ID="2e70e3d8-45ce-470b-8d42-4d121053c978" />
  </DataVHDs>
</xml>
```

Component runbooks

One of the primary benefits of mapping out the optimized, end-to-end process on paper is that it serves as an input to the functional specifications of the runbooks that must be created. Using the framework described in this book, the first step in runbook authoring after the target process to automate has been defined is to create component runbooks for the primary functionality required. For the sample VM provisioning process in this chapter, the following component runbooks are required:

- **Read XML File** This component runbook reads the XML file and publishes the variables and values from the XML file to the Orchestrator data bus. While this is a component runbook, we recommend it be placed in the control runbook folder because it is specific to this process and unlikely that it will be reused by any other scenario.

- **Validate Inputs** This component runbook validates all data gathered from the XMLfile based on ranges, formats, and values. The validation runbook should be called in each step of the control runbook to ensure that the most current version of the file is used. Similar to the Read XML File runbook, this runbook is very specific to this scenario so we recommend placing it in the control runbook because it is specific to this process and unlikely that it will be reused by any other scenario.
- **Update Status and Set Status** These two component runbooks maintain the status field in the XML file. The Update Status runbook updates the status field while the Set Status runbook increments the status integer. These two runbooks ensure that the overall process status is up to date and the provisioning steps occur in the correct sequence.
- **Get VM Host Rating** This component runbook gets the host rating from VMM to make sure that there is a host available for the new VM. Windows PowerShell cmdlets will be utilized within an Orchestrator Run .Net Script activity to get the VM host rating required. If a host is not available, an error is returned and the control runbook will evaluate the returned result and call the error routine if the error cannot be handled by the control runbook.
- **Create New VM** This component runbook creates a new virtual machine. This runbook is called only if Get VM Host Rating is successful and a host is available for placement of the new VM. Windows PowerShell cmdlets will be utilized within an Orchestrator Run .Net Script activity to create the VM.
- **Get VMM Job Status** This component runbook is called to verify that the VMM job, in this case the creation of the new VM, was successful. This runs periodically in a loop using the XML file until the status returned is successful.

Component runbook detail: GetVMHostRating

The runbook shown in Figure 6-1 is one of the required component runbooks for provisioning a new VM.

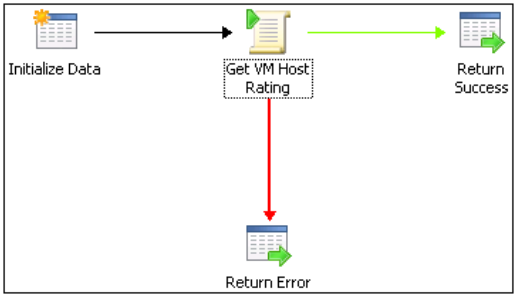


FIGURE 6-1 An example of the GetVMHostRating component runbook.

The runbook performs the following activities:

- Initializing data
- Running the .NET script
- Returning data (such as a success or error status)

An example of where this runbook would be placed is shown in Figure 6-2. Because this is a component runbook that uses System Center Virtual Machine Manager through PowerShell, the runbook is placed in the appropriate folder.

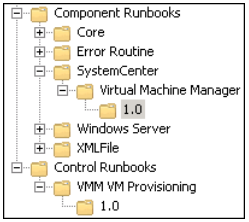


FIGURE 6-2 A view of the Orchestrator folder structure.

Initialize Data

In the first step, the Initialize Data activity defines the following input parameters (each with a data type: string):

- **VMName** The name of the VM
- **VMMServerName** The name of the VMM server
- **RequestID** A GUID that is generated and included in the XML file to give each VM provisioning request a unique identifier
- **CloudName** The name of the VMM cloud the VM should be provisioned to
- **UserRoleName** The name of the user role the VM should be provisioned for
- **ExistingVMTemplateName** The name of the existing template in VMM that will be used

Get VM Host Rating

In the second step, the Run .Net Script activity executes a Windows PowerShell script which takes the input data and parameters that the Initialize Data activity published to the data bus and executes a remote call to System Center VMM to execute the Get-SCVMHostRating cmdlet. This cmdlet will query the VMM server using all the parameters supplied (such as cloud and user role) to find the optimal Hyper-V host to provision the virtual machine to. This step will get the name of the recommended host and publish the name to the data bus.

In the Details tab, the following Windows PowerShell code is utilized. In this case, the Windows PowerShell script code contains the main functionality of the component runbook which determines the recommended host to place the VM being provisioned on.

```
# Get Input parameters
$RequestID = ""
$VMName = ""
$VMMServerName = ""
$ExistingVMTemplateName = ""
$UserRoleName= ""
$CloudName = ""

$ErrorState = 0
$ErrorMessage = ""
$Trace = ""
$Error.Clear()
try
{

   $Trace +=" Validating the VMMServer 'r'n"
          if($VMMServerName -eq "")
         {
            $ErrorState=2
            Throw "Error: Invalid Input Parameters."
         }
         if(!(Test-Connection $VMMServerName))
         {
               $ErrorState=2
               Throw "Error:Unable To Reach VMMServer .."
         }
     $Session = New-PSSession -ComputerName $VMMServerName
     If ($Session -eq $null) {
           $ErrorMessage =  $Error[0]
           $Trace += "Could not create PSSession on $VMMServerName"
           $ErrorState = 2
     }
     else
   {
        $ReturnArray = Invoke-Command -Session $Session -Argumentlist $RequestID,
$VMMServerName,$UserRoleName,$CloudName,$ExistingVMTemplateName,$VMName   -scriptblock {
Param ($RequestID, $VMMServerName, $UserRoleName, $CloudName, $ExistingVMTemplateName,
$VMName)

        $ErrorState = 0
        $ErrorMessage = ""
        $Trace = ""
        $Error.Clear()
```

```
        try    {
             $Trace += "Validating The Input Parameters.. 'r'n"
             if(($RequestID.length -lt 1)-or ($VMMServerName.length -lt 1) -or
($UserRoleName.length -lt 1)     -or ($CloudName.length -lt 1) -or
($ExistingVMTemplateName.length -lt 1) -or ($VMName.length -lt 1))
             {
                   Throw "Error: One or more required parameters is NULL"
             }
              if($VMName.Length -gt 15)
             {
                    $ErrorState=2
                    Throw "Error:VMName Length Should Not Increase More Than 15
Characters.."
             }
             $Trace += "Imporing the VMM Modules `r`n"
             try
             {
                    $Path = Get-ItemProperty "HKLM:\Software\Microsoft\Microsoft System
Center Virtual    Machine Manager Server\Setup"
                    $FinalPath = $Path.InstallPath +
"\bin\psModules\VirtualMachineManager\virtualmachinemanager.psd1"
                    Import-Module $FinalPath
              }
              catch
              {
                    $Trace += "Importing of the VMM module failed as the module was
already present `r`n"
              }

                 $Trace += "Getting connection with VMM Server $VMMServerName 'r'n"
                 Get-SCVMMServer -ComputerName $VMMServerName -UserRoleName
$UserRoleName | Out-Null

                 $Trace += "Getting the cloud object for $CloudName cloud 'r'n"
                 $Cloud = Get-SCCloud -Name $CloudName

                 $Trace += "Getting the VM Template object for template
$ExistingVMTemplateID 'r'n"
                 $VMTemplate = Get-SCVMTemplate  | where {$_.Name -eq
$ExistingVMTemplateName}

                          $Trace += "Checking if Cloud and Template were successfully
fetched 'r'n"
                 if(($Cloud -ne $null) -and ($VMTemplate -ne $null))
```

```
                {
                        $Trace += "Trying to get the Cloud Ratings 'r'n"
                        $HostRatings = @(Get-SCVMHostRating -Cloud $Cloud -VMTemplate
$VMTemplate -DiskSpaceGB 20 -VMName $VMName | sort -Descending Rating)

                        $Trace += "Checking if Host Rating is non null and has valid
data 'r'n"
                        If (($HostRatings -ne $null) -and ($HostRatings.Count -gt 0))
                        {
                                $HostRating = $HostRatings[0].Rating
                                $Trace += "Host Rating is: $HostRating. 'r'n"
                                if($HostRating -eq 0)
                                {
                                        $Trace += "Host Rating is zero. Getting the
reason for Zero Rating. 'r'n"
                                        $Trace +=
$HostRatings[0].ZeroRatingReasonList.Get_Item(0).Description
                                        Throw "Error: No non-zero host rating found
for cloud $Cloud."
                                }
                        }
                        else
                        {
                                Throw "Error: No non-zero host rating found for cloud
$Cloud and VM Template: $ExistingVMTemplateName."
                        }
                }
                else
                {
                        Throw "Error: Not able to get the cloud:$Cloud or VM
Template:$ExistingVMTemplateName."
                }
    }
    Catch
    {
        $ErrorState = 2
        $ErrorMessage = $error[0].Exception.ToString()
    }
    Finally
    {
        $Trace += "'r'n"
        $Trace += "Exiting Script 'r'n"
        $Trace += "ErrorState:  $ErrorState 'r'n"
        $Trace += "ErrorMessage: $ErrorMessage 'r'n"
    }
```

```
    # end finally
    $Results = @($ErrorState, $ErrorMessage, $Trace, $HostRating)
    Return $Results
}
$ErrorState = $ReturnArray[0]
$ErrorMessage = $ReturnArray[1]
$Trace = $ReturnArray[2]
$HostRating = $ReturnArray[3]

Remove-PSSession -Session $Session
}
}
catch
{
      $ErrorState = 2
      $ErrorMessage = $error[0].Exception.ToString()


}
finally
{
         $Trace += "'r'n"
         $Trace += "Exiting New VM 'r'n"
         $Trace += "ErrorState:   $ErrorState 'r'n"
         $Trace += "ErrorMessage: $ErrorMessage 'r'n"
}
```

The above code leverages the Remote PowerShell session patterns from Chapter 5 where we described how to make remote PowerShell calls from a runbook to a management server and executing script code in the remote session. In this example, we are creating a remote Windows PowerShell session to the VMM server. There it imports the VMM Windows PowerShell module and cmdlets then connects to the VMM server by calling Get-SCVMMServer. Then the script gets the VMM cloud and the VMM template using the values from the data bus which were supplied by the process calling this component runbook and supplied to the Initialize Data starting point activity. The script must return data other than null. In a successful case it will return ErrorState with a value of zero.

Once the script code is entered, the Windows PowerShell variables in the script requiring values from previous steps in the runbook must be subscribed to from the data bus. This is one of the most powerful features of Orchestrator, the ability for one step to consume data from previous steps. Right-click within the brackets of each variable assignment, in the context menu that appears, go to Subscribe - Published Data, select the Initialize Data activity and then choose the appropriate variable from the data bus (see Figure 6-3).

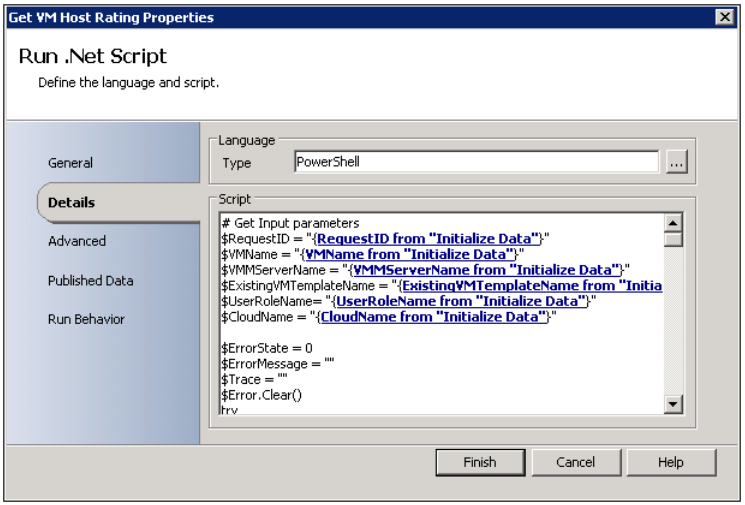


FIGURE 6-3 An example of mapping published data to script.

With the script code entered and the variables linked to published data, the script output—consisting of additional published data—can be configured using the Published Data tab as shown in Figure 6-4. On this page, you define the variables to be published from the Run .Net Script activity. As described previously, any Windows PowerShell script for runbooks using the framework described in this book should always return at least three published data values including the ErrorState, ErrorMessage, and Trace values. Additionally, in an example such as this where the purpose of the runbook is to determine a specific output (such as the name of the Hyper-V host to place a VM) that output should also be published to the data bus. In this example, in addition to the standard three values returned, there is also a HostRating variable that will be published as well for subsequent steps to consume.

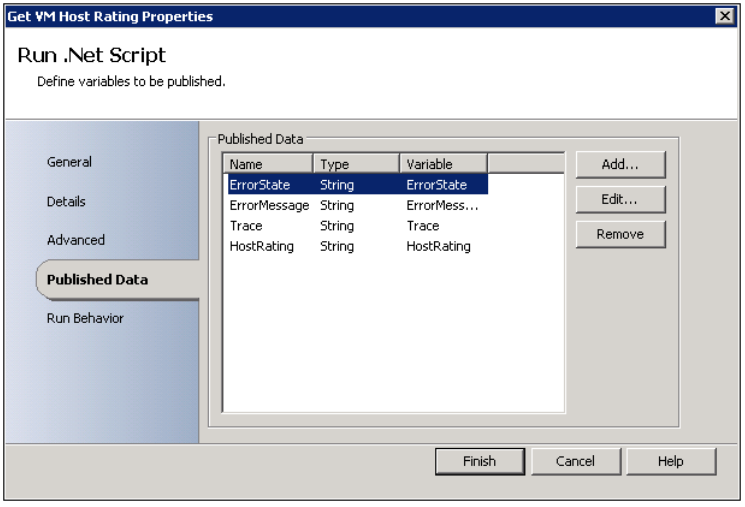


FIGURE 6-4 You can see the defined variable in the Publish Data tab.

Return Data

The third step in the GetVMHostRating component runbook is to define the return data from the runbook. This is a two-step process. First the data to be returned must be defined on the runbook properties level and then the runbook itself must include one or more Return Data activities to map the data appropriately.

To define the return data on the runbook level, right-click the runbook tab at the top of the design console for the current runbook and choose Properties. On the Returned Data tab, you define the settings shown in Figure 6-5.

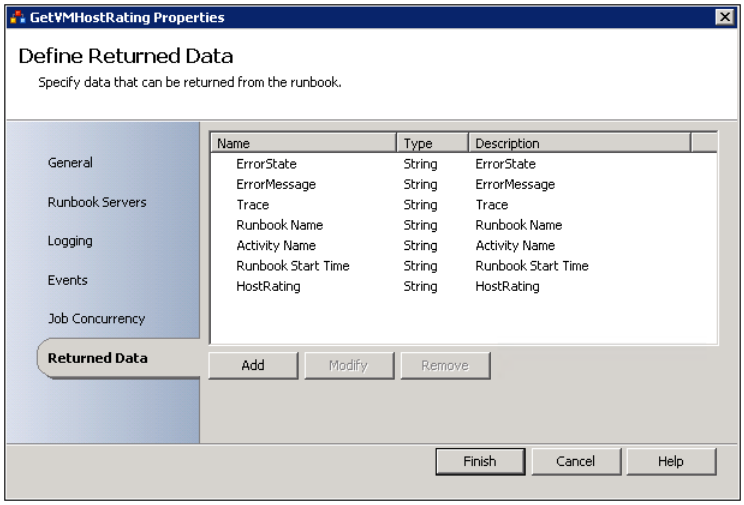


FIGURE 6-5 You can specify the data in the Returned Data tab.

With the runbook level returned data configured, now the published data to return it to the calling runbook must be configured. For all component runbooks, there should be a Return Success and Return Error path. Both should be configured with the data shown in Figure 6-6. The parameter list is populated from the runbook level configured previously, then each parameter should be mapped to published data to populate the values by right-clicking the field and subscribing to published data from the data bus.

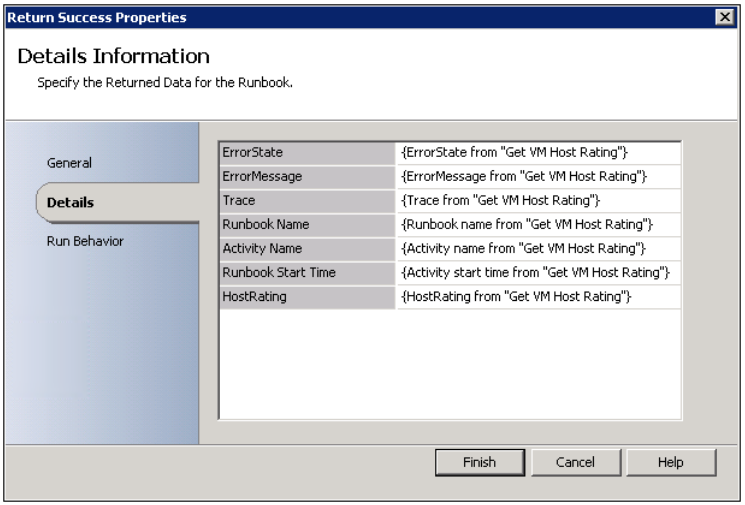


FIGURE 6-6 Mappping published data to the script

Link and Conditional Logic

Once the activities in the runbook are configured, the links and conditional logic between the activities can be configured. The Initialize Data activity should be linked to the Get VM Host Rating activity and pass if Initialize Data results in success. From the Get VM Host Rating activity there will be two links, one for success conditions linked to the Return Success activity and another for error conditions linked to the Return Error activity. For the link to the Return Error activity, the include filters required are shown in Figure 6-7. These filters cover several scenarios. The first is if the Windows PowerShell script exits with a terminating error, in that case the Run .Net Script activity will return a warning or failed status. This is how unhandled errors are captured. The next filters cover logical errors that are handled by the script when the script exits with a status of two or three.

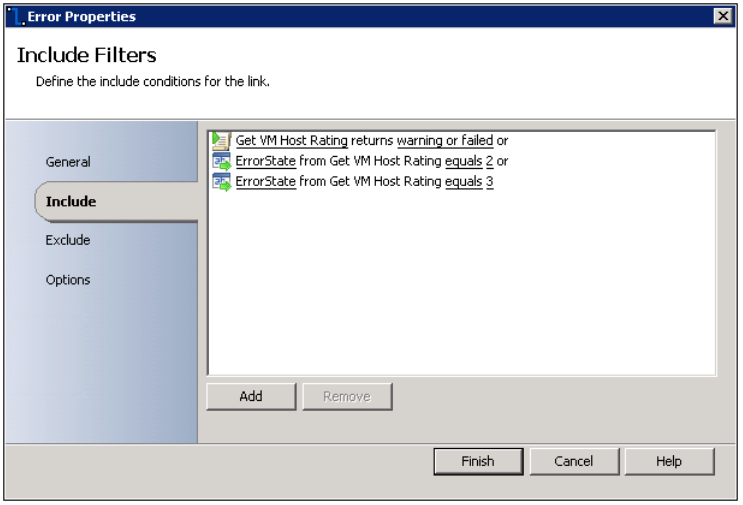


FIGURE 6-7 Include filters for the Return Error path.

For the link to the Return Success activity, the include filters should look like Figure 6-8. In this framework, a status of zero or one indicates that the process should proceed along the success path.

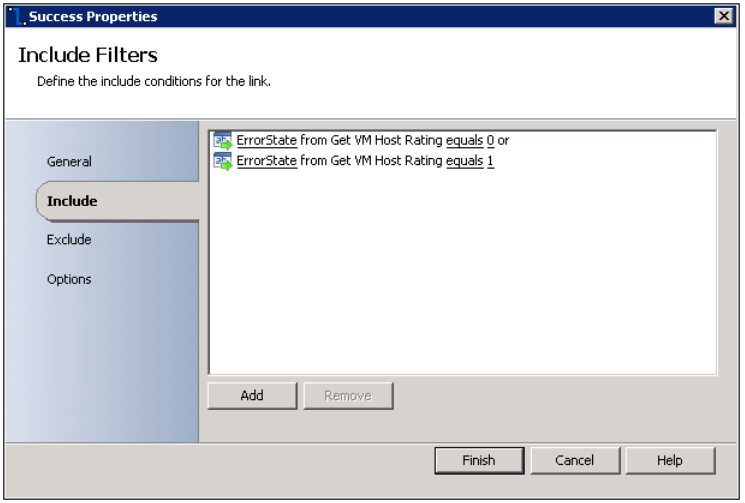


FIGURE 6-8 Include filters for the Return Success path.

Component runbook summary

This section outlined a completed component runbook. This runbook can be tested using the Runbook Tester (from within Orchestrator Runbook Designer) or by creating another runbook that will have an Initialize Data activity (with no parameters defined) and an Invoke Runbook activity (that will call your component runbook and specify all required test data). For all component runbooks created it is highly recommended to test as many possible initial conditions and as much data as possible since component runbooks are intended to be highly reusable and must be very robust. Test cases should include valid data, invalid data, missing data, and so on. The expected result is that all conditions should either result in success or in handled errors with trace data.

Component runbook detail: CreateNewVM

The CreateNewVM component runbook uses a very similar structure to the GetVMHostRating runbook, as shown in Figure 6-9. For brevity, only the differences and the underlying Windows PowerShell script will be described in this section.

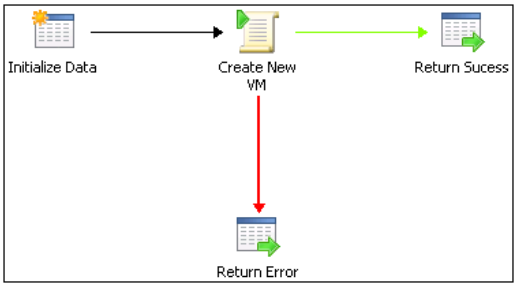


FIGURE 6-9 The CreateNewVM component runbook.

Initialize Data

In the first step, the Initialize Data activity defines the following input parameters (each with a data type: string):

- **RequestID** A GUID that is generated and included in the XML file to give each VM provisioning request a unique identifier
- **IsWorkgroup** Indicates whether the VM will be a workgoup VM
- **UserRoleName** The name of the user role the VM should be provisioned for
- **certificateFindType** Utilized for securing credentials and passwords for local administration
- **certificateFindValue** Utilized for securing credentials and passwords for local administration
- **opalisHelper** Utilized for executing command line or console programs
- **IsLocalAdminPasswordEncrypted** Utilized for securing credentials and passwords for local administration
- **certificateStore** Utilized for securing credentials and passwords for local administration
- **LocalAdminPassword** Decrypted password for the local administrator
- **CertificateLocation** Utilized for securing credentials and passwords for local administration
- **ExistingVMTemplateName** The name of the existing template in VMM that will be used
- **CloudName** The name of the VMM cloud the VM should be provisioned to
- **ProvisioningType** Type of VM provisioning to be performed
- **VMMServerName** The name of the VMM server
- **WorkgroupName** The name of the workgroup if the VM will not be domain-joined
- **VMFQDN** The fully qualified domain name of the virtual machine

Create New VM

This component runbook utilizes a Run .Net Script activity. In the Details tab, the following Windows PowerShell code is utilized. In this case, the Windows PowerShell script code contains the main functionality of the component runbook which creates a new VM using the data from the Initialize Data activity.

```
#Inputs from published data
$RequestID = ""
$VMMServerName = ""
$ExistingVMTemplateID = ""
$VMFQDN = ""
$ProvisioningType = ""
$IsWorkgroup = ""
$LocalAdminPassword = ""
$WorkgroupName = ""
$LocalAdminUser = "administrator"
$CloudName = ""
$UserRoleName = ""

#encryption related Variables
$opalisHelper = ""
$IsLocalAdminPasswordEncrypted = ""
$executeCommand = "DecryptMessage"
$certificateLocation = ""
$certificateStore = ""
$certificateFindValue = ""
$certificateFindType = ""

$ErrorState = 2
$ErrorMessage = ""
$Trace = ""
$Error.Clear()

$ErrorInDecryption = 0
#Checking and Decrypting Local Admin Password
$Trace += "Checking if the password is encrypted 'r'n"
$Trace += "IsLocalAdminPasswordEncrypted: $IsLocalAdminPasswordEncrypted 'r'n"
if($IsLocalAdminPasswordEncrypted -eq "True")
{
    $Trace += "Password is encrypted. Trying to decrypt it. 'r'n"
    try
    {
       $Trace += "Creating a script block for decryption. 'r'n"
       $command = [ScriptBlock]::Create("$opalisHelper $executeCommand
```

```
'$LocalAdminPassword' $certificateLocation $certificateStore $certificateFindValue
$certificateFindType")

       $Trace += "Invoking the script block to decrypt the encrypted password. 'r'n"
       $output = Invoke-Command -ScriptBlock $command

       $Trace += "Checking if the decryption was successful. 'r'n"
       if($output.StartsWith("SUCCESS:") -eq $true)
       {
                       $Trace += "Decription successful. Getting the decrypted password.
'r'n"
                       $LocalAdminPassword = $output.TrimStart("SUCCESS:");
       }
       else
       {
                       Throw "Decryption process did not succeeded."
       }
    }
    catch
    {
         $Trace += "Not able to decrypt password. 'r'n"
         $ErrorState = 2
         $ErrorMessage = $error[0].Exception.ToString()
         #$Trace += "Exiting script. Please make sure that the certificates are properly
installed and then retry. 'r'n"
         $ErrorInDecryption = 1
    }
}
else
{
    $Trace += "Password is not encrypted. 'r'n"
}

if($ErrorInDecryption -eq 1)
{
    $ErrorState = 2
    $Trace += "Exiting Script as there was error in the Decryption process. 'r'n"
}
else
{
     $Session=""
     try
     {
     $Trace=" Validating The VMMServer 'r'n"
         if($VMMServerName -eq "")
```

```
        {
                        $ErrorState=2
                        Throw "Error: Invalid Input Parameters."
        }
                if(!(Test-Connection $VMMServerName))
                    {
                        $ErrorState=2
                        Throw "Error:Unable To Reach VMMServer .."
                    }

    $Session = New-PSSession -ComputerName $VMMServerName

   If ($Session -eq $null -or $Session -eq "") {
        $ErrorMessage =  $Error[0]
        $Trace += "Could not create PSSession on $VMMServerName or Invalid VMMSever
'r'n"
        $ErrorState = 2

    }
         else
         {
             $ReturnArray = Invoke-Command -Session $Session -Argumentlist $RequestID,
$VMMServerName,$ExistingVMTemplateID, $VMFQDN, $ProvisioningType, $IsWorkgroup,
$LocalAdminUser, $LocalAdminPassword, $WorkgroupName, $CloudName, $UserRoleName  -
scriptblock {
                 Param ($RequestID, $VMMServerName,$ExistingVMTemplateID, $VMFQDN,
$ProvisioningType, $IsWorkgroup, $LocalAdminUser, $LocalAdminPassword, $WorkgroupName,
$CloudName, $UserRoleName)

             $ErrorState = 0
             $ErrorMessage = "No Error"
             $Trace = ""
             $Error.Clear()

       Try
       {
                                    $Trace = "Beginning Create Virtual Machine Script
'r'n"
            $Trace += "'r'n"
            $Trace += "Request ID:                                          $RequestID 'r'n"
            $Trace += "VMMServerName:                                 $VMMServerName 'r'n"
            $Trace += "ExistingVMTemplateName:               $ExistingVMTemplateID 'r'n"
            $Trace += "VMFQDN:                                                $VMFQDN 'r'n"
            $Trace += "ProvisioningType:                            $ProvisioningType 'r'n"
            $Trace += "IsWorkgroup:                                      $IsWorkgroup 'r'n"
```

```
            $Trace += "LocalAdminUser:                                $LocalAdminUser 'r'n"
            $Trace += "WorkgroupName:                                  $WorkgroupName 'r'n"
            $Trace += "CloudName:                                         $CloudName `r`n"
            $Trace += "UserRoleName:                                   $UserRoleName 'r'n"
            #Validating The Input Parameters
            $Trace += "Validating The Input Parameters.. 'r'n"
                if(($RequestID -eq "" -or $RequestID -eq $null) -or ($VMMServerName -eq
"" -or $VMMServerName -eq $null)   -or ($VMFQDN -eq "" -or $VMFQDN -eq $null) -or
($UserRoleName -eq "" -or $UserRoleName -eq $null)  -or ($ProvisioningType -eq "" -or
$ProvisioningType -eq $null) -or ($CloudName -eq "" -or $CloudName -eq $null) -or
($IsWorkgroup -eq "" -or $IsWorkgroup -eq $null) -or ($ExistingVMTemplateID -eq "" -or
$ExistingVMTemplateID -eq $null) )
                {
                    $ErrorState=2
                    Throw "Error: Invalid Input Parameters."
                }
                 if(!($IsWorkgroup.ToLower() -eq "true") -and !($IsWorkgroup.ToLower()
-eq "false"))
                    {
                         $ErrorState=2
                         Throw "Error: Invalid IsWorkgroup."
                    }
                     if(($LocalAdminUser -eq "" -or $LocalAdminUser -eq $null) -and
$IsWorkgroup.ToLower() -eq "true")
                    {
                        $ErrorState=2
                        Throw "Error: Invalid LocalAdmin."
                    }
                     if(($ProvisioningType.ToLower() -ne "windows") -and
($ProvisioningType.ToLower() -ne "linux"))
                    {
                        $ErrorState=2
                        Throw "Error: Invalid ProvisioningType."
                    }

                #Convert VMFQDN to Netbios Name
                $VMName = $VMFQDN.Split('.')[0]
           if($VMName.Length -gt 15)
            {
                 $ErrorState=2
                Throw "Error:VMName Length Should Not Increase More Than 15
Characters.."
            }

                 $JobGroup = [System.Guid]::NewGuid()
```

```
                $Trace += "Importing VMM module 'r'n"
                try
                {
               $Path = Get-ItemProperty "HKLM:\Software\Microsoft\Microsoft System
Center Virtual Machine Manager Server\Setup"
                $FinalPath = $Path.InstallPath +
"\bin\psModules\VirtualMachineManager\virtualmachinemanager.psd1"
                ipmo $FinalPath
            }
                 Catch
                 {
                $Trace += "Importing of the VMM module failed as the module was already
present `r`n"
                 }
           # $Trace += "Getting VM Template with id $ExistingVMTemplateID... 'r'n"
           # $existingVMTemplate = Get-SCVMTemplate | where { $_.Name -eq
$ExistingVMTemplateID }
            #If ($existingVMTemplate -eq $null)
          #  {
             #   Throw "Error: VM Template with id $ExistingVMTemplateID not found."
        #    }
        Get-SCVMMServer -ComputerName $VMMServerName -UserRoleName $UserRoleName| Out-
Null

                $Trace += "Checking if the VM is Linux VM 'r'n"
                if (($ProvisioningType -eq "Hyper-V Linux") -or ($ProvisioningType -eq
"VMWare Linux"))
                {
                                                        $Trace += "Setting Virtual COM
Port for Linux VM `r`n"
                    Set-VirtualCOMPort -NamedPipe "\\.\pipe\$VMName" -GuestPort 1
-JobGroup $JobGroup
                }

            $Trace += "Checking if the VM is Windows Workgroup VM 'r'n"
            if (($IsWorkgroup -eq "true") -and ($ProvisioningType -like "*Windows*"))
            {
                 #Creating local administrator credentials when the Provisioning type is
windows and it is a workgroup VM
                $LocalAdminCred = New-Object -TypeName
System.Management.Automation.PSCredential -argumentlist $LocalAdminUser,(ConvertTo-
SecureString -AsPlainText -Force -String $LocalAdminPassword)
            }

           $Trace += "Getting UserRole $UserRoleName 'r'n"
```

```
           $UserRole = Get-SCUserRole -Name $UserRoleName

            $Trace += "Getting VM Template $ExistingVMTemplateID ... 'r'n"
            ($VMTemplate = Get-SCVMTemplate | where {$_.Name -eq $ExistingVMTemplateID})
| Out-Null
            If ($VMTemplate -ne $null)
            {
               $Trace += "Creating the VM Configurations based on the VMTemplate ...
'r'n"
               $virtualMachineConfiguration = New-SCVMConfiguration -VMTemplate
$VMTemplate -Name $VMName

                $Trace += "Getting the cloud object on which to deploy the VM ... 'r'n"
                $Cloud = Get-SCCloud -Name $CloudName

                $Trace += "Checking if Cloud object is null... 'r'n"
                if($Cloud -eq $null)
                {
                    $ErrorState = 2
                    Throw "Error: Cannot find the specified Cloud : $CloudName"
                }
                If ($Cloud -ne $null)
                {
                 $Trace += "Checking Provisioning type and Creating VM $VMName ... 'r'n"
                    $Trace += "Checking if Provisioning type is valid ... 'r'n"
                    if( ($ProvisioningType -eq $null) -or ($ProvisioningType -eq ""))
                    {
                          $Trace += "Provisioning type is not valid. 'r'n"
                        Throw "Error: Invalid Provisioning Type $ProvisioningType"
                    }
                    elseif (($ProvisioningType -eq "Hyper-V Linux") -or
($ProvisioningType -eq "VMWare Linux"))
                    {
                          $Trace += "Creating the Linux VM ... 'r'n"
                        $VM = New-SCVirtualMachine -Name $VMName -UserRole $UserRole -
VMConfiguration $virtualMachineConfiguration -Cloud $Cloud -RunAsynchronously -JobGroup
$JobGroup -Description "VM Provisioning Request ID: $RequestID" -
SkipInstallVirtualizationGuestServices -StartVM | Out-Null
                    }
                    elseif ($IsWorkgroup -eq "true")
                    {
                          $Trace += "Creating Workgroup VM ... 'r'n"
                        $Trace += "Checking if LocalAdministratorCredential is not null
... 'r'n"
                        if($LocalAdminCred -eq $null)
```

```
                        {
                          Throw "Error: Local Admin Credential is null"

}
                            $Trace += "Credentials have valid data. Creating the VM. ...
'r'n"
                            $VM = New-SCVirtualMachine -Name $VMName -UserRole $UserRole
-VMConfiguration $virtualMachineConfiguration  -Cloud $Cloud -
LocalAdministratorCredential $LocalAdminCred -Workgroup $WorkgroupName -
RunAsynchronously -JobGroup $JobGroup -Description "VM Provisioning Request ID:
$RequestID" -ComputerName $VMName -SkipInstallVirtualizationGuestServices -StartVM |
Out-Null
                        }
                        elseif ($IsWorkgroup -eq "false")
                        {
                                        $Trace += "Creating Domain joined VM ... 'r'n"
                                       $VM = New-SCVirtualMachine -Name $VMName -
UserRole $UserRole -VMConfiguration $virtualMachineConfiguration  -Cloud $Cloud -
RunAsynchronously -JobGroup $JobGroup -Description "VM Provisioning Request ID:
$RequestID" -ComputerName $VMName -SkipInstallVirtualizationGuestServices -StartVM |
Out-Null
                        }
                        If ($? -eq "True")
                        {
                          $ErrorState = 0
                        }
                        Else
                        {
                          Throw $Error[0]
                        }
                    }
                    Else
                    {
                        Throw "VM Host $VMHostName not found."
                    }
                }
                Else
                {
                    Throw "VM Template $VMTemplateName not found."
                }
            }
            Catch
            {
                    $ErrorState = 2
                    $ErrorMessage = $error[0].Exception.ToString()
```

```
            }
            Finally
            {
                $Trace += "`r`n"
                $Trace += "Exiting New VM `r`n"
                $Trace += "ErrorState:   $ErrorState `r`n"
                $Trace += "ErrorMessage: $ErrorMessage `r`n"
            }
            $Results = @($ErrorState, $ErrorMessage, $Trace, $VM.ID)
            Return $Results
        }
            $ErrorState = $ReturnArray[0]
            $ErrorMessage = $ReturnArray[1]
            $Trace += $ReturnArray[2]
            $VMID = $ReturnArray[3]
            Remove-PSSession -Session $Session
        }
}
catch
{
        $ErrorState = 2
         $ErrorMessage = $error[0].Exception.ToString()
}
finally
{
                $Trace += "'r'n"
                $Trace += "Exiting New VM 'r'n"
                $Trace += "ErrorState:   $ErrorState 'r'n"
                $Trace += "ErrorMessage: $ErrorMessage 'r'n"

}
}
```

The above code leverages the Remote PowerShell session patterns from Chapter 5 where we described how to make remote PowerShell calls from a runbook to a management server and executing script code in the remote session. In this example, we are creating a remote Windows PowerShell session to the VMM server. There it imports the VMM Windows PoweShell module and cmdlets then connects to the VMM server by calling Get-SCVMMServer. Then the script gets the VMM cloud and the VMM template using the values from the data bus which were supplied by the process calling this component runbook and supplied to the Initialize Data starting point activity. The script then uses the input data to run the New-SCVirtualMachine cmdlet on the VMM server to create a new virtual machine. The script must return data other than null. In a successful case it will return ErrorState with a value of zero. Critical to the framework outlined in this book, the script and this component runbook do not wait for the VM to be created, the task is run asynchronously, meaning the

component runbook will end as soon as the command is executed but the process will continue to run on the VMM server. This pattern is utilized so that in high volume scenarios there are not dozens or hundreds of long-running runbooks open on the Orchestrator server waiting for VMs to be created.

Return Data

The third step in the CreateNewVM component runbook is to define the return data from the runbook. The process is the same as for the GetVMHostRating runbook described previously so we will not repeat it in this section.

Links and Conditional Logic

Once the activities in the runbook are configured, the links and conditional logic between the activities can be configured. The process is the same as for the GetVMHostRating runbook described previously so we will not repeat it in this section.

Control runbooks

With the component runbooks created, the next step in creating the VM provisioning process is to author the required control runbooks. The control runbooks are where the process logic and state management of the process are contained. For the VM provisioning process in this example, several control runbooks are defined.

Control runbook detail: Monitor VM Provisioning

The first control runbook is called Monitor VM Provisioning, which is shown in Figure 6-10. This runbook is the only runbook in this example that is set to start and stay running in Orchestrator on an ongoing basis. This runbook uses a Monitor File starting point activity and monitors a particular folder for a new instance of an XML input file for VM creation.

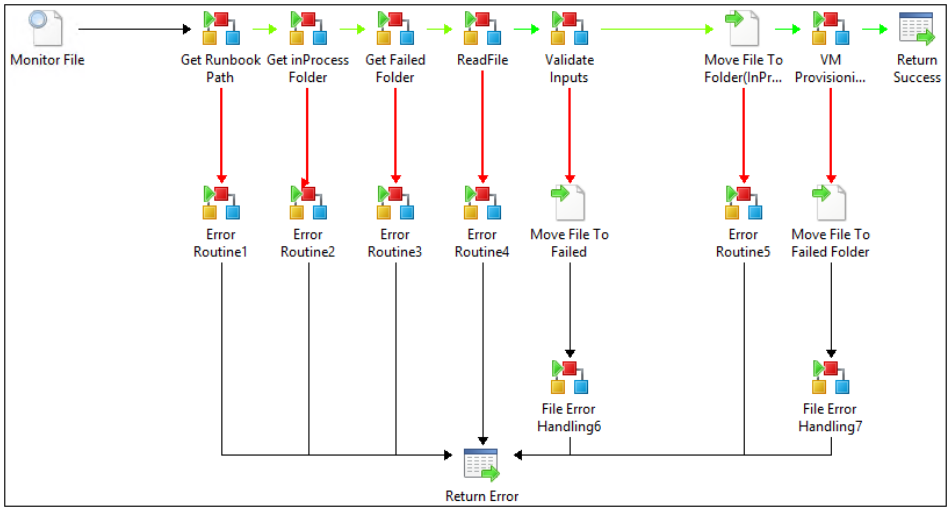


FIGURE 6-10 A diagram of the Monitor VM Provisioning runbook.

This runbook monitors a file share (called the Active folder based on the routing pattern described in the previous chapter) and waits for an XML file to be created in the monitored location. It will then process the file, get all the different paths to the folders that are required for the routing pattern, read the file, validate the input, and move the file to the InProcess folder. It then calls the VM Provisioning Engine control runbook without waiting for completion. After that this instance is completed and it will again wait for another XML file.

Control runbook: VM Provisioning Engine

The next control runbook is the VM Provisioning Engine runbook. This runbook contains the majority of the process flow and state management logic. This control runbook, which is shown in Figure 6-11, is called VM Provisioning Engine and its purpose is to make sure all required steps are invoked in the right order and that management of the state of the overall process is tracked to enable the ability to restart the process from the last successful step should a correctable error be encountered.

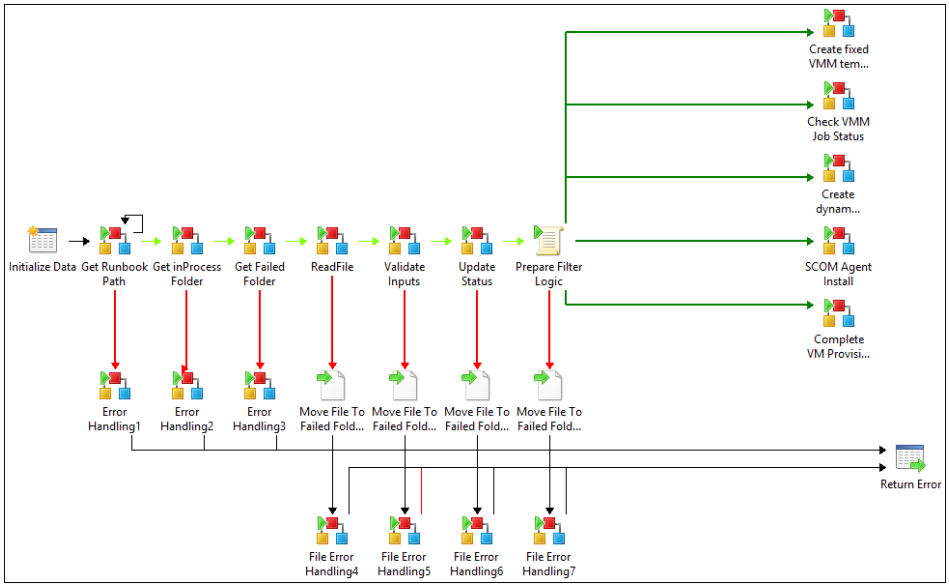


FIGURE 6-11 A diagram of the VM Provisioning Engine runbook.

The VM Provisioning Engine will process the XML file from the previous runbook (Monitor VM Provisioning) from the InProcess folder and continue the process execution. As you can see in Figures 6-10 and 6-11, both control runbooks share the first few component runbook calls, the difference between Monitor VM Provisioning and VM Provisioning Engine starts with updating the status field in the XML file. In this case, it will increase the status value by one to indicate that the next phase of the process has begun.

As mentioned earlier, the status is responsible for the steps (control runbooks) involved and in which order they are executed. Based on the status value, it will call one of the five additional control runbooks, each executing a specific step:

- Create a new VM from a fixed template
- Create a new VM from a dynamic template
- Check VMM job status
- SCOM Agent installation
- Complete VM provisioning

The Prepare Filter Logic activity is a simple script activity that will prepare and analyze the current status information. For some activities, Orchestrator allows only the usage of data from the connected activity, not the activities previous to it. So in this example you are not able to access data from Read File, but only from Update Status. However, the VM Provisioning process needs to use a parameter from the XML file that will identify whether a static or dynamic template should be used to determine which runbook to call next or take a different path through the process flow. So we introduce the Prepare Filter Logic script which can easily

access all published data, such as that from the Read File activity. The output of that script will then be used as the filter criteria for the different paths.

The VM Provisioning Engine control runbook and the other related control runbooks should be created in the folder structure outlined previously and as shown in Figure 6-12.

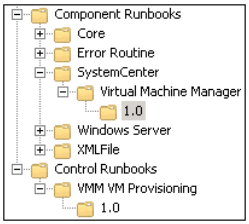


FIGURE 6-12 Control Runbooks folder hierarchy in Orchestrator.

The control runbook is built using the following runbook activities:

- Initialize Data
- Invoke Runbook (multiple times, one for each component runbook call)
- .NET Script
- Return Data

The key is to understand that all Invoke Runbook activities call component runbooks that are either located in the Component Runbooks folder or—if they are specific to the IT process (such as Read XML File)—component runbooks that are located in the Control Runbooks folder itself.

It is important to check the settings such as Invoke By Path (to make sure your runbook works in other environments as well) and Wait For Completion as shown in Figure 6-13. Wait for completion is required in many cases otherwise your control runbook would not wait for the data that will be returned by the component runbook. In other cases, there may be situations where you don't want to wait for completion. It is important to consider the desired outcome each time and use the appropriate setting for the circumstance.

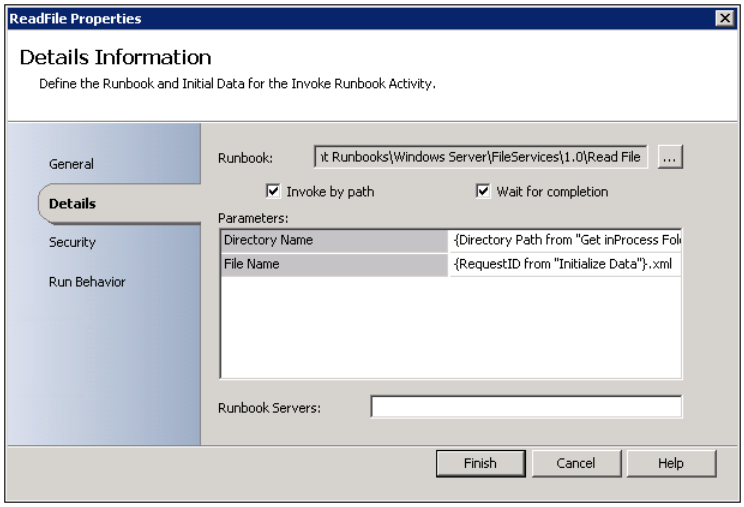


FIGURE 6-13 The Details tab with the Invoke Runbook settingfor ReadFile.

After the ReadFile step, the Update Status step calls a component runbook while supplying input parameters including the FilePath, FileName, and Status value. This value will be used to increase the status in the file. In this case it will be increased by one. It will then publish the new updated value back the data bus. We will use this value to determine the right path for the VM Provisioning Engine runbook to take. The reasoning behind this pattern is that VM provisioning is a multistep process where some steps are long running and where some steps might fail. This pattern and its associated state management using a file and status numbers lets us model a wide range of process flows using a single runbook. This runbook will be called several times through the VM provisioning process and the execution path taken will depend on the status code.

The Prepare Filter Logic activity uses the following script so that it can use data in addition to the Status field alone. It will take the two variables, the VMTemplateType and the updatedStatus to calculate another output value called outStatus.

```
#Inputs
$outVMTemplateType = "{VMMTemplateType from "ReadFile"}"
$inStatus = "{updatedStatus from "Update Status"}"

#Standard Variables
$ErrorState=0
$ErrorMessage=""
$Trace=""
$Error.Clear()

try
{
    $Trace="Preparing Filter logic 'r'n"
```

```
    $Trace +="Input Parameters: 'r'n"
    $Trace +="outVMTemplateType:$outVMTemplateType 'r'n"
    $Trace +="inStatus:$inStatus  'r'n"

    $outStatus=$inStatus
    if($outVMTemplateType -eq "fixed" -and $inStatus -eq 1) { $outStatus=1 }
    if($outVMTemplateType -eq "dynamic" -and $inStatus -eq 1) { $outStatus=2 }
    $Trace+=" Preparing Filter Logic Completed Successfully... 'r'n"
}
catch
{
    $Trace += "Exception caught in remote action '$Action'... 'r'n"
    $ErrorState = 2
    $ErrorMessage = $error[0].Exception.tostring()
}
finally
{
    $Trace += "Exiting remote action $Action.. 'r'n"
    $Trace += "ErrorState: $ErrorState.. 'r'n"
    $Trace += "ErrorMessage: $ErrorMessage'r'n"
}
```

The outStatus will be published as filterStatus to the data bus and then be leveraged to define the path and the next step, as shown in Figure 6-14.

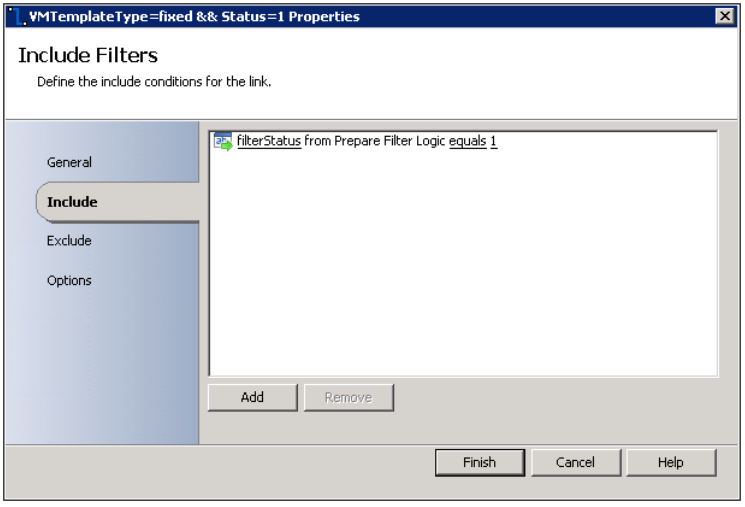


FIGURE 6-14 Filter logic for the VM Provisioning Engine runbook.

This concept becomes powerful when you consider that one activity can have multiple output links and possible execution paths. While this can increase complexity, the use of link labels to indicate process flow can be helpful, as shown in Figure 6-15.

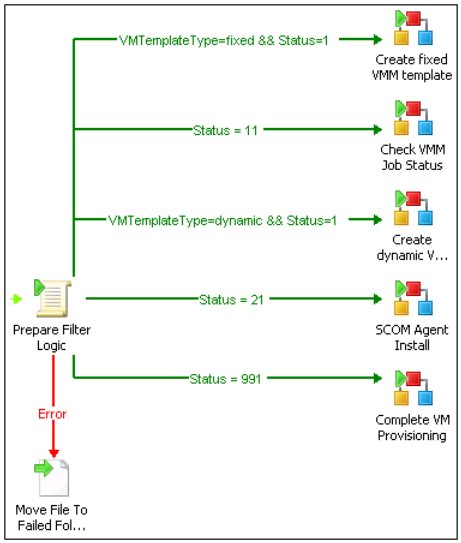


FIGURE 6-15 Link logic for the VM Provisioning Engine control runbook process flow.

All Invoke Runbook calls to the various control runbooks that implement the specific steps, such as Create Fixed VMM Template, are configured without Wait For Completion checked (see Figure 6-16). That will help to avoid long-running runbooks by not holding open the VM Provisioning Engine runbook. The instance of the VM Provisioning Engine is completed within a few seconds since it would not wait until the VM is created. The reason this is possible in such a multistep process is the use of the XML file and status codes to maintain state across multiple runbooks.

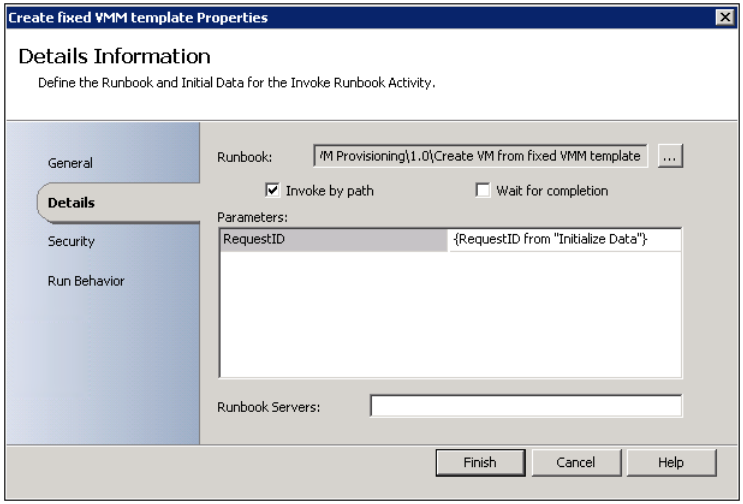


FIGURE 6-16 Invoke Runbook activity without waiting for completion.

Remaining control runbooks

Figure 6-17 illustrates that there are a number of additional runbooks in the VM provisioning process. For brevity, these are not described in detail; they are either very similar to existing runbooks already documented or they are unique to a particular design such as the step to install the System Center Operations Manager agent in the VM after provisioning. They are included in the figures as examples of how the process can be extended with different steps specific to your particular requirements and how the same framework and design patterns can be utilized.

Initiation runbook

Both the component and control runbooks required for the VM provisioning process have been described in this chapter. The final level of framework is the initiation runbook level. As discussed in previous chapters, the reason for this modularity is to support multiple methods of triggering a given process. This example uses the simplest model, which is creating an initiation runbook that creates a file in a particular location to trigger the Monitor VM Provisioning control runbook. In another environment it might be desirable to have a portal or help desk application initiate a VM provisioning request and this could be easily implemented by creating new initiation runbooks without having to modify the control or component runbooks at all, provided the required data is supplied to them.

Initiation runbook: Initiate VM Provisioning

In the example documented here, a runbook will generate an XML file that will trigger a new VM provisioning instance. This runbook is implemented as an initiation runbook that will again validate the input to make sure all values are in a given range or from a specific type. It will get the folder path of the active folder and create the XML file there. With that approach, every file will be picked up automatically by the Monitor VM Provisioning runbook and kick off a new process immediately.

The Initiate VM Provisioning runbook is shown in Figure 6-17.

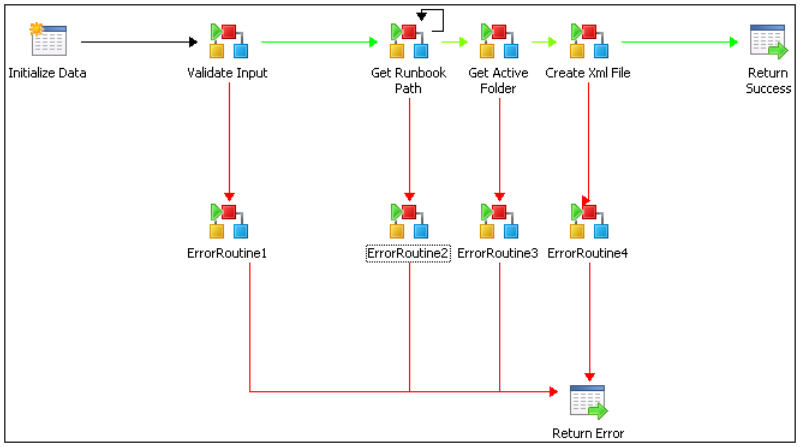


FIGURE 6-17 Initiation runbook for VM provisioning.

The runbook uses the following activities:

- Initialize Data
- Invoke Runbook
- Return Data

Initialize Data now defines all input parameters from the XML file that was shown earlier in this chapter. The same Validation and Get Path component runbooks used in the previous examples to validate the input data and get the appropriate folders are utilized.

Next, another component runbook that creates the XML file will be utilized as shown in Figure 6-18.

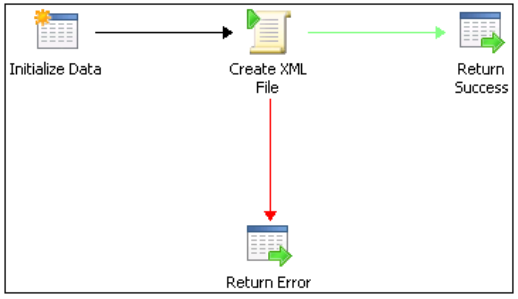


FIGURE 6-18 The Create XML File Runbook.

The Create XML File component runbook called by the Initiate VM Provisioning control runbook takes all the input data and creates an XML file in the given path, as shown in Figure 6-19.

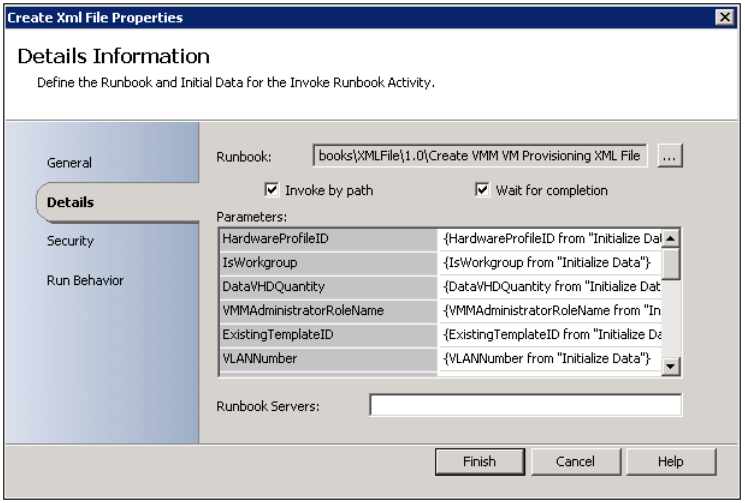


FIGURE 6-19 The results of Invoke Runbook activity of the Create XML File Properties.

CHAPTER 7

Calling and executing Orchestrator runbooks

In this chapter we will discuss some of the ways to execute your System Center Orchestrator 2012 runbooks. The options for execution include using the Orchestration console, programmatically using the Orchestrator REST application programming interfaces (APIs), and executing runbooks though the System Center Service Manager service catalog.

Orchestration console

The Orchestration console is a web application installed by Orchestrator that allows you to view the real-time status of your Orchestrator environment as well as start and stop runbook jobs. To launch the Orchestration console from a browser's address bar, type **http://<computer name>:<port number>** where *computer name* is the name of the server where the web service is installed, and *port* is the port number selected during configuration of the web service. By default, the port is 82.

The web application is broken down into three main areas as shown in Figure 7-1: navigation on the left, details in the middle, and actions on the right. The navigation pane allows you to either view the runbook folder tree (similar to the runbook tree shown in the Runbook Designer), view the status of runbook jobs and instances by runbook sever, or view events raised by the management and runbook services.

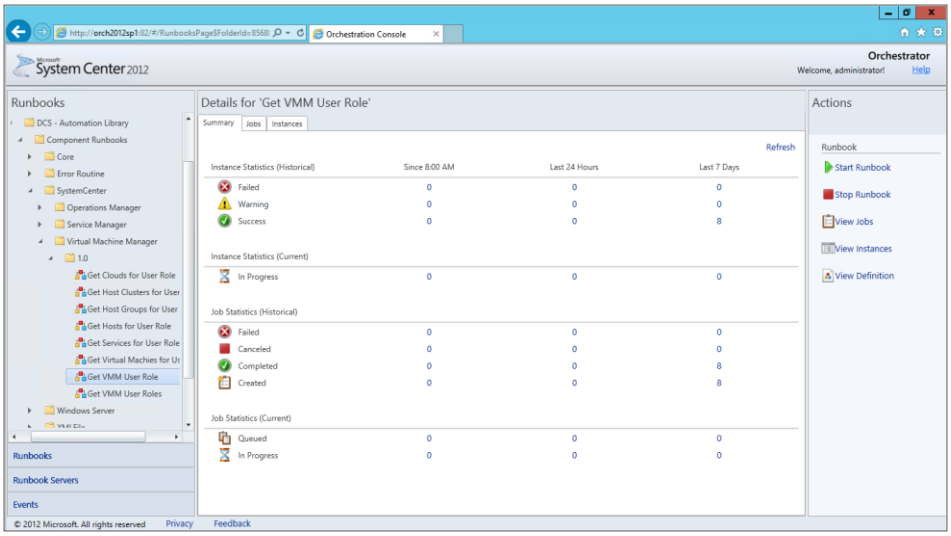


FIGURE 7-1 A view of the Orchestration console.

To start a runbook, from the Runbook navigation view, navigate to the runbook in question using the tree structure. Unlike the Runbook Designer, the actual runbook is part of the navigation structure that can be seen in Figure 7-1. Select the runbook you want to start from the navigation pane and from the Actions pane, select Start Runbook. This opens a Start Runbook dialog box similar to the one shown in Figure 7-2. Enter the values for any runbook parameters. If you want to run the runbook on a server other than its default, click a server in Available Runbook Server(s), and then click the right arrow to add the server to the Selected Runbook Server(s). Click Start to start the runbook.

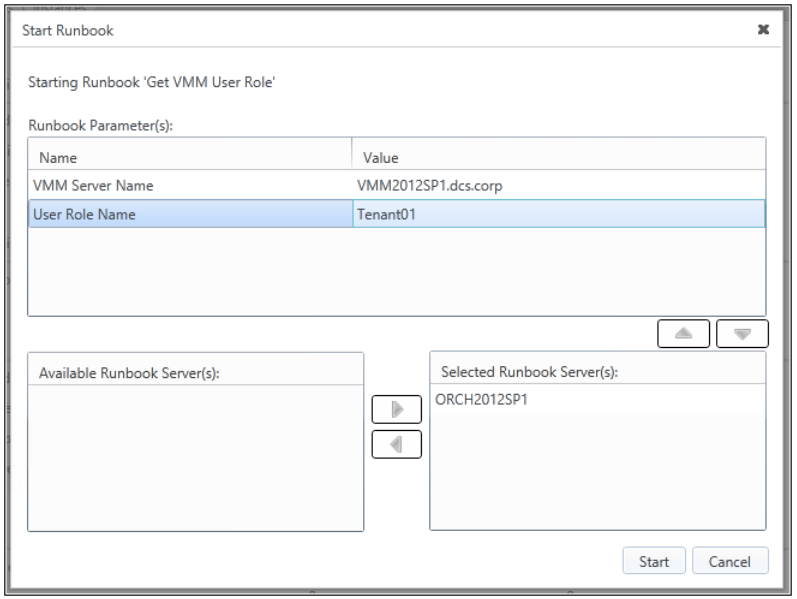


FIGURE 7-2 The Start Runbook dialog box.

You can switch to the Jobs view in the Details pane to see the job you just started. To drill down into the job instance, you can click View Instances under Job in the Actions pane. To view the execution of your runbook instance, click the Instance View Details link in the Action pane. This opens a window similar to the one shown in Figure 7-3. This view is broken down into two main areas, the Instance Summary—where general execution status, inputs, and outputs are available—and the Diagram/Activity Details section—where you can see a graphical view of the execution path as well as activity specific details for every activity that was executed.

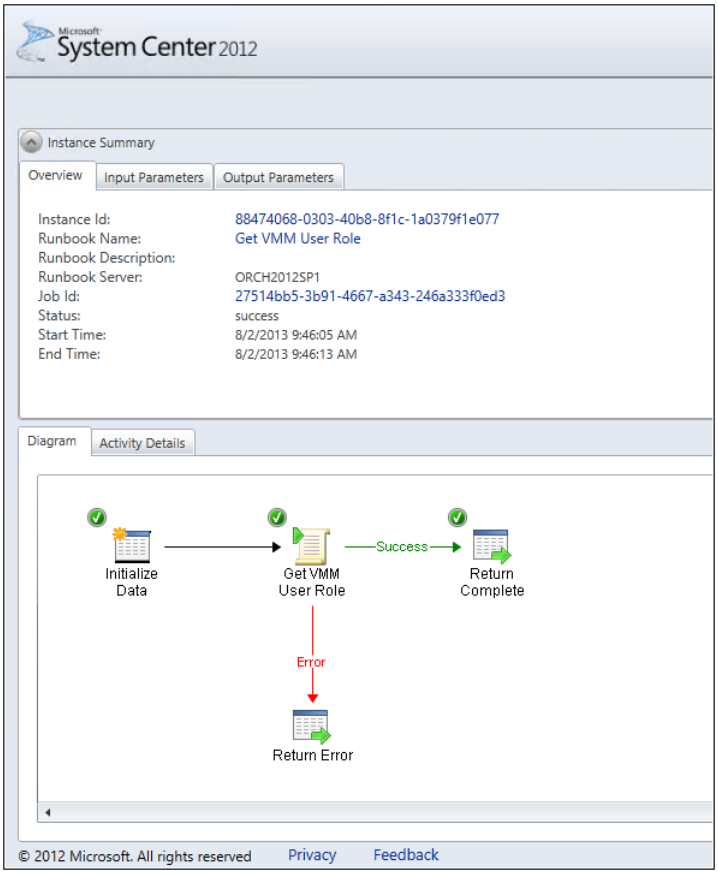


FIGURE 7-3 You can view the instance details of a runbook job.

Orchestrator REST API

The Orchestrator web service is a REST-based web service that exposes Orchestrator resources and relationships through the Open Data Protocol (OData). You can access the web service using programming languages such as C# and Java or scripting languages such as Windows PowerShell.

Microsoft Visual Studio

The Orchestrator REST API relies on GUIDs to launch runbooks, namely the runbook ID and each of the input parameter IDs. These are not easily discoverable so it may help to create some helper functions to do some of the lookups for these GUIDs. In creating the following project, a service reference named SCOService was added pointing to the Orchestrator web service URL, in the form of *http://<server>:<port>/Orchestrator2012/Orchestrator.svc*, where the default port is 81.

NOTE **All of the code, scripts, and scriptlets in this chapter are available as a zipped archive from *http://aka.ms/SCrunbook/files*.**

The first helper function returns a Runbook object based on the path to the runbook.

```
private static Runbook GetRunbookByPath(string serviceURL, string path)
{
    Runbook runbook = null;

    SCOService.OrchestratorContext context =
       new SCOService.OrchestratorContext(new Uri(serviceURL));

    // Set credentials to default or a specific user.
    context.Credentials = System.Net.CredentialCache.DefaultCredentials;

    // Get the runbook based on the entered path
    runbook = context.Runbooks.Where(rb => rb.Path == path).First();
    return runbook;
}
```

The second helper function returns a list of RunbookParameter objects based on the runbook GUID and only returns parameters whose direction is marked as In.

```
private static List<RunbookParameter> GetRunbookParametersById(string serviceURL,
   Guid runbookID)
{
    List<RunbookParameter> parameters = null;
    SCOService.OrchestratorContext context =
       new SCOService.OrchestratorContext(new Uri(serviceURL));

    // Set credentials to default or a specific user.
    context.Credentials = System.Net.CredentialCache.DefaultCredentials;

    // Get the parameters based on the runbook ID
    parameters = context.RunbookParameters.Where(rp => rp.RunbookId == runbookID
       && rp.Direction == "In").ToList();
    return parameters;
}
```

The last helper function actually submits the runbook job based on the values from the above functions plus values for the parameters.

```
private static Job StartRunbookJob(string serviceURL, Guid runbookID,
   List<RunbookParameter> runbookParameters, Hashtable runbookParameterValues)
{
    Job job = new Job();
    // Configure the XML for the parameters
```

```
    StringBuilder parametersXml = new StringBuilder();
    if (runbookParameters != null && runbookParameters.Count() > 0)
    {
        parametersXml.Append("<Data>");
        foreach (var param in runbookParameters)
        {
           parametersXml.AppendFormat(
              "<Parameter><ID>{0}</ID><Value>{1}</Value></Parameter>",
              param.Id.ToString("B"), runbookParameterValues[param.Name]);
        }
        parametersXml.Append("</Data>");
    }

    try
    {
        // Create new job and assign runbook Id and parameters.
        job.RunbookId = runbookID;
        job.Parameters = parametersXml.ToString();

        SCOService.OrchestratorContext context =
           new SCOService.OrchestratorContext(new Uri(serviceURL));

        // Set credentials to default or a specific user.
        context.Credentials = System.Net.CredentialCache.DefaultCredentials;

        // Add newly created job.
        context.AddToJobs(job);
        context.SaveChanges();

        return job;
    }
    catch (DataServiceQueryException ex)
    {
        throw new ApplicationException("Error starting runbook.", ex);
    }
}
```

FInally, here is the code that calls the helper functions to submit a runbook job. The first section of variables should be changed to match your environment as well as the runbook you are targeting.

```
static void Main(string[] args)
{
    // Begin change values
    string serviceURL = "http://orch2012sp1:81/Orchestrator2012/Orchestrator.svc";
    string runbookPath = @"\DCS - Automation Library\Component
```

```
Runbooks\SystemCenter\Virtual Machine Manager\1.0\Get VMM User Role";
    Hashtable parameters = new Hashtable();
    parameters["VMM Server Name"] = "VMM2012SP1.dcs.corp";
    parameters["User Role Name"] = "TenantAdmin1";
    // End change values

    Runbook runbook = GetRunbookByPath(serviceURL, runbookPath);
    List<RunbookParameter> runbookParameters = GetRunbookParametersById(serviceURL,
                                                    runbook.Id);
    Job job = StartRunbookJob(serviceURL, runbook.Id, runbookParameters, parameters);
    Console.WriteLine("Successfully started runbook. Job ID: {0}", job.Id.ToString());
    Console.ReadKey();
}
```

Windows PowerShell

Orchestrator currently does not ship with any built-in Windows PowerShell cmdlets but you can make calls to the Orchestrator REST API in a similar fashion to the way described in the prior section. The same basic flow of getting the runbook by its path, then its parameters, and finally submitting the job still apply.

First get the runbook ID based upon the runbook path.

```
$serviceURL = "http://orch2012sp1:81/Orchestrator2012/Orchestrator.svc";
$runbookPath = "\DCS - Automation Library\Component Runbooks\SystemCenter\Virtual
Machine Manager\1.0\Get VMM User Role";
$parameters = @{"VMM Server Name" = "VMM2012SP1.dcs.corp"; "User Role Name" =
"TenantAdmin1"}

# Get the runbook based on the path
$fullUrl = [String]::Format("{0}/Runbooks?`$filter=Path eq '{1}'", $serviceURL,
$runbookPath);
$request = [System.Net.HttpWebRequest]::Create($fullUrl)

# Build up a nice User Agent
$request.UserAgent = $(
   "{0} (PowerShell {1}; .NET CLR {2}; {3})" -f $UserAgent,
$(if($Host.Version){$Host.Version}else{"1.0"}),
   [Environment]::Version,
   [Environment]::OSVersion.ToString().Replace("Microsoft Windows ", "Win")
)
$request.UseDefaultCredentials = $true

[System.Net.HttpWebResponse] $response = [System.Net.HttpWebResponse]
$request.GetResponse()
$reader = [IO.StreamReader] $response.GetResponseStream()
$responseString = $reader.ReadToEnd()
```

```
$reader.Close()
$response.Close()
$xml = [xml]$responseString

$runbookID = $xml.feed.entry.content.properties.Id.InnerText
```

Then get the runbook parameters based on the runbook ID making sure to only grab parameters whose direction is set to In. In this case, once we have the parameters we are going ahead and setting up the parameters XML for the eventual job request.

```
# Get the runbook parameters based on the runbook ID
$fullUrl = [String]::Format("{0}/Runbooks(guid'{1}')/Parameters", $serviceURL,
$runbookId);
$request = [System.Net.HttpWebRequest]::Create($fullUrl)

# Build up a nice User Agent
$request.UserAgent = $(
   "{0} (PowerShell {1}; .NET CLR {2}; {3})" -f $UserAgent,
$(if($Host.Version){$Host.Version}else{"1.0"}),
   [Environment]::Version,
   [Environment]::OSVersion.ToString().Replace("Microsoft Windows ", "Win")
)
$request.UseDefaultCredentials = $true

[System.Net.HttpWebResponse] $response = [System.Net.HttpWebResponse]
$request.GetResponse()
$reader = [IO.StreamReader] $response.GetResponseStream()
$responseString = $reader.ReadToEnd()
$reader.Close()
$response.Close()
$xml = [xml]$responseString

# Format the param string from the Parameters hashtable
$rbParamString = "<d:Parameters><![CDATA[<Data>"
foreach ($entity in $xml.feed.entry)
{
   if ($entity.content.properties.Direction -eq "In")
   {
      $rbParamString = -join
($rbParamString,"<Parameter><ID>{",$entity.content.properties.Id.InnerText,"}</ID>
<Value>",$parameters[$entity.content.properties.Name],"</Value></Parameter>")
   }
}
$rbParamString += "</Data>]]></d:Parameters>"
```

Finally, submit the runbook job using the runbook ID we discovered as well as the runbook parameter XML we generated.

```
# Create the request object for submitting the job
$fullUrl = [String]::Format("{0}/Jobs", $serviceURL);
$request = [System.Net.HttpWebRequest]::Create($fullUrl)

# Set the credentials to default or prompt for credentials
$request.UseDefaultCredentials = $true

# Build the request header
$request.Method = "POST"
$request.UserAgent = $(
   "{0} (PowerShell {1}; .NET CLR {2}; {3})" -f $UserAgent,
$(if($Host.Version){$Host.Version}else{"1.0"}),
   [Environment]::Version,
   [Environment]::OSVersion.ToString().Replace("Microsoft Windows ", "Win")
)
$request.Accept = "application/atom+xml,application/xml"
$request.ContentType = "application/atom+xml"
$request.KeepAlive = $true
$request.Headers.Add("Accept-Encoding","identity")
$request.Headers.Add("Accept-Language","en-US")
$request.Headers.Add("DataServiceVersion","1.0;NetFx")
$request.Headers.Add("MaxDataServiceVersion","2.0;NetFx")
$request.Headers.Add("Pragma","no-cache")

# Build the request body
$requestBody = @"
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
    <content type="application/xml">
        <m:properties>
            <d:RunbookId m:type="Edm.Guid">$runbookID</d:RunbookId>
            $rbparamstring
        </m:properties>
    </content>
</entry>
"@

# Create a request stream from the request
$requestStream = new-object System.IO.StreamWriter $Request.GetRequestStream()

# Sends the request to the service
$requestStream.Write($RequestBody)
$requestStream.Flush()
```

```
$requestStream.Close()

# Get the response from the request
[System.Net.HttpWebResponse] $response = [System.Net.HttpWebResponse]
$Request.GetResponse()

# Write the HttpWebResponse to String
$responseStream = $Response.GetResponseStream()
$readStream = new-object System.IO.StreamReader $responseStream
$responseString = $readStream.ReadToEnd()

# Close the streams
$readStream.Close()
$responseStream.Close()

# Get the ID of the resulting job
if ($response.StatusCode -eq 'Created')
{
    $xmlDoc = [xml]$responseString
    $jobId = $xmlDoc.entry.content.properties.Id.InnerText
    Write-Host "Successfully started runbook. Job ID: " $jobId
}
else
{
    Write-Host "Could not start runbook. Status: " $response.StatusCode
}
```

System Center Service Manager service catalog

Delivering a self-service experience within your organization begins with defining what the standardized service offerings will be. For each proposed service, line-of-business (LOB) application owners and IT agree on what information is required to fulfill the requests, where this information will come from, who needs to approve such requests, and establish the expected performance levels that are required for each service.

Service Manager and Orchestrator together allow the fabric administrator to establish these standardized offerings by storing the above information and using it to drive the service delivery and automation process. The primary interface to the above processes is through the Service Manager service catalog.

From the service catalog, customers identify and request services offered by IT. Users start by selecting a service offering. These provide the high-level list of all of the things that can be requested; for example, the top-level offerings might be Cloud Services, Desktop Services, and so forth.

Within each service offering that a fabric administrator defines, a customer can select one or more request offerings. Request offerings contain details of a specific request that IT offers

to the organization. Each request offering contains information such as cost, SLA details, knowledge articles, and specific input requirements in the form of user prompts that a requestor, such as the application owner, completes as part of the request process.

The steps for a fabric administrator to enable this functionality are:

- Create runbooks to fulfill service requests in Orchestrator.
- Create an Orchestrator connector to synchronize runbook metadata from Orchestrator into Service Manager.
- Create a runbook automation activity template in Service Manager that binds to a particular runbook.
- Create a service request template in Service Manager that maps the workflow steps to fulfill a service request.
- Create a request offering in Service Manager that uses the service request template and creates customer prompts to collect data and maps those values to the service request.
- Lastly, create a service offering in Service Manager that houses one or more related request offerings. A service offering for Virtual Machines may contain request offerings for new virtual machine, resize virtual machine, and delete virtual machine among others.

The remainder of this section will walk you through the above steps to create a request offering based on resizing a virtual machine.

Create an initiation runbook

As discussed in Chapter 5, "Orchestrator runbook best practices and patterns," you can create an initiation runbook that can be used by System Center Service Manager to execute a runbook. Please refer to the "Service requests initiation runbooks" section in Chapter 5 for more information. For this chapter, we will use the same initiation runbook that will resize a virtual machine's number of virtual processors and/or the amount of memory given to the VM as shown in Figure 7-4.

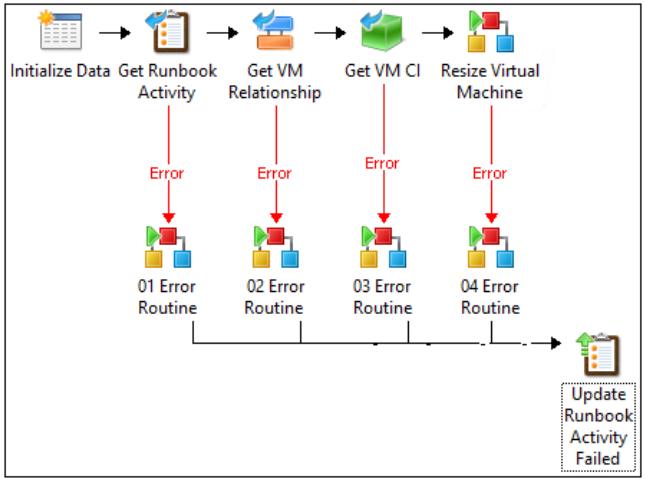


FIGURE 7-4 An example of a Service Manager initiation runbook.

The Initialize Data activity defines the following inputs to support the scenario shown in Table 7-1 below.

TABLE 7-1 Inputs for the Initialize Data Activity.

NAME	DESCRIPTION
SM Activity ID	The Service Manager GUID that uniquely identifies this runbook automation activity. This is needed for error control and for accessing any Service Manager configuration items related to this service request.
User Role Name	The name of the user role this VM belongs to.
Number of vCPU	The integer number of vCPUs the VM should have.
Memory MB	The amount of memory the VM should have in MB.

If you look at the inputs you will notice that the name of the virtual machine is not listed. The virtual machine will come across as a related configuration item so that additional properties of the virtual machine can be accessed in your runbook. The properties we care about include the VMM VM ID which uniquely identifies the virtual machine as well as the VMM server that this VM attached to. We could also look at the existing vCPU and memory to detect which properties are changing.

For more information on the rest of the activities in this runbook, refer back to the "Service requests initiation runbooks" section in Chapter 5. For more information on how to sync VMM data into Service Manager, see Appendix B.

Create an Orchestrator connector

On your Service Manager server or where you have the Service Manager console installed, launch the console and then navigate to the Administration section. In the Administration pane, expand Administration, and then click Connectors. In the Tasks pane, under Connectors, click Create Connector, and then click Orchestrator Connector. This launches the Orchestrator Connector Wizard, as shown in Figure 7-5. In the Orchestrator Connector Wizard dialog box, click Next.

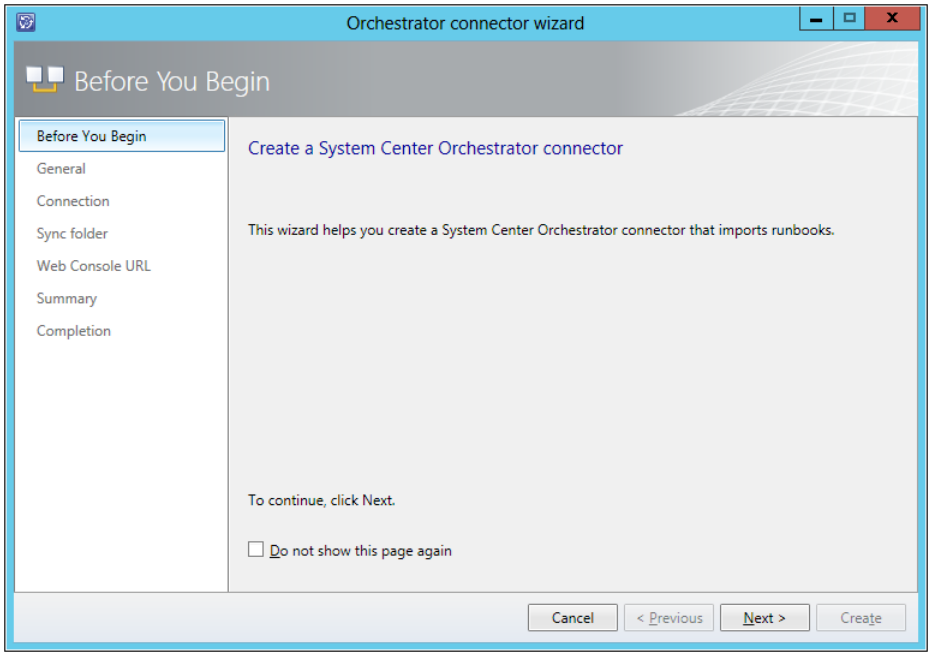


FIGURE 7-5 The Orchestrator Connector Wizard.

Give the connector a name and, optionally, a description before clicking Next as shown in Figure 7-6.

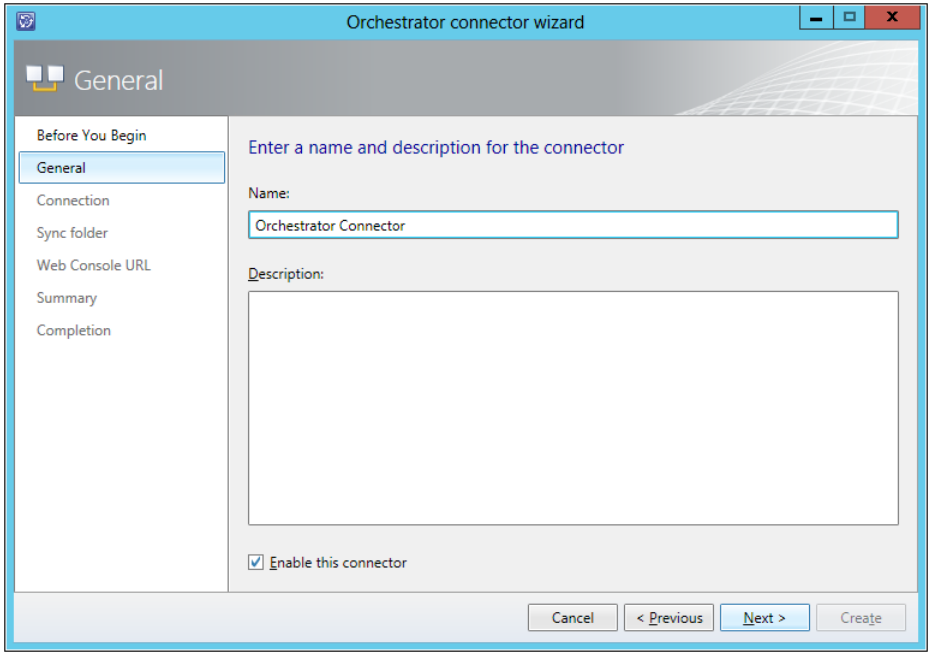


FIGURE 7-6 The General page of the Orchestrator Connector Wizard.

Enter in the URL for the Orchestrator web service in the form *http://<server>:<port>/Orchestrator2012/Orchestrator.svc* where *<server>* is the server where the web service is installed and *<port>* is the port where the web service is installed which by default is port 81. Add credentials that have full access to Orchestrator in the Credentials section and click Test Connection to verify, as shown in Figure 7-7. Then click Next.

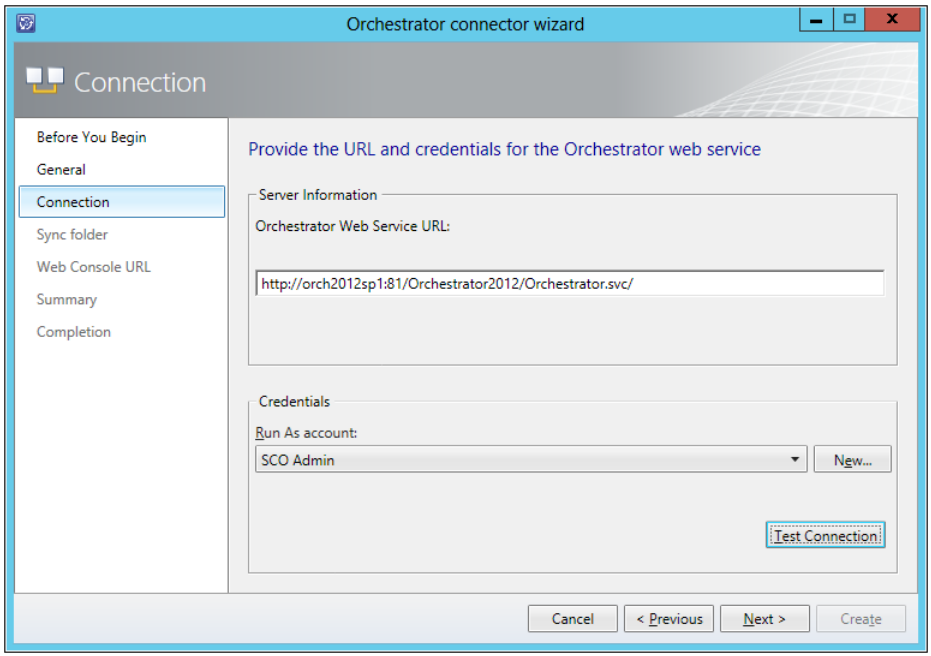


FIGURE 7-7 The Connection page of the Orchestrator Connector Wizard.

Click Next on the Sync Folder page to accept the root path for syncing all runbooks on this server, as shown in Figure 7-8.

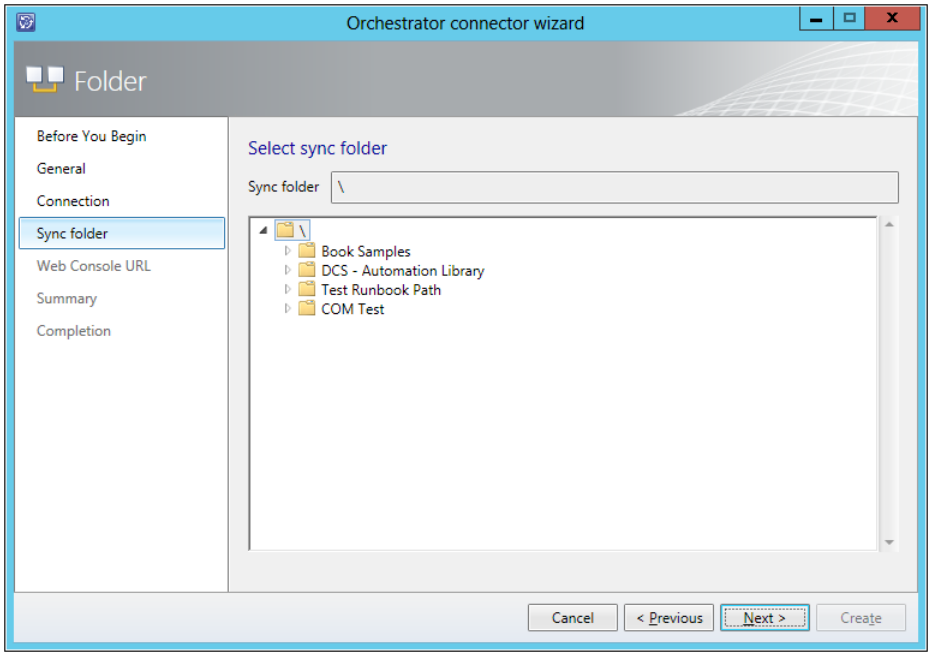


FIGURE 7-8 The Sync Folder page of the Orchestrator Connector Wizard.

On the Web Console URL page, enter in the address for the Orchestration console in the form of http://<*server*>:<*port*>/ where <*server*> is the server where the web console is installed and <*port*> is the port where the web console is installed which by default is port 82. This is shown in Figure 7-9. Then click Next.

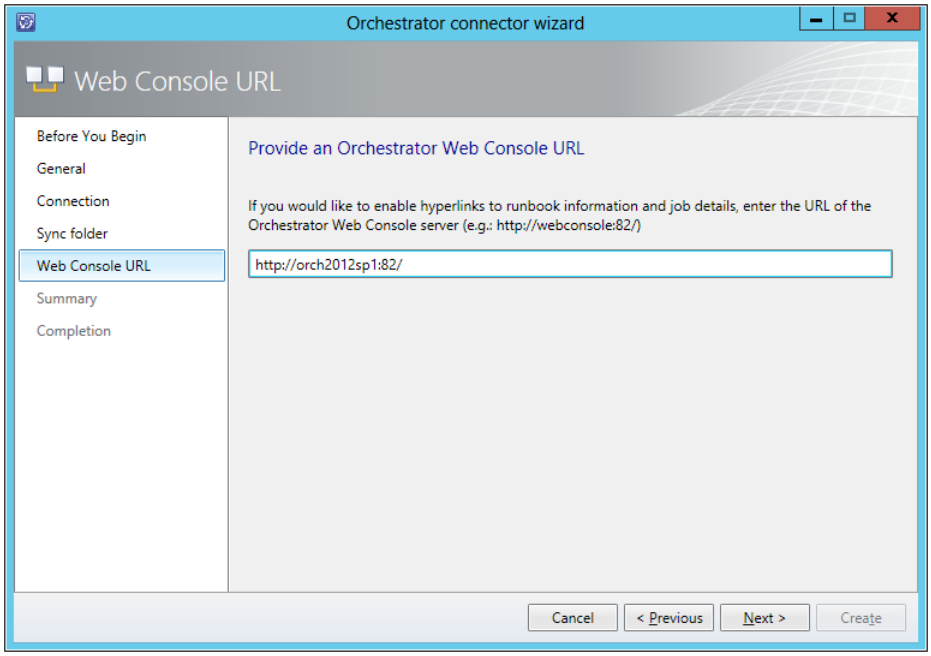


FIGURE 7-9 The Web Console URL page of the Orchestrator Connector Wizard.

On the Summary page, click Create to create the connector and when completed, click Close to close the wizard. This is shown in Figure 7-10.

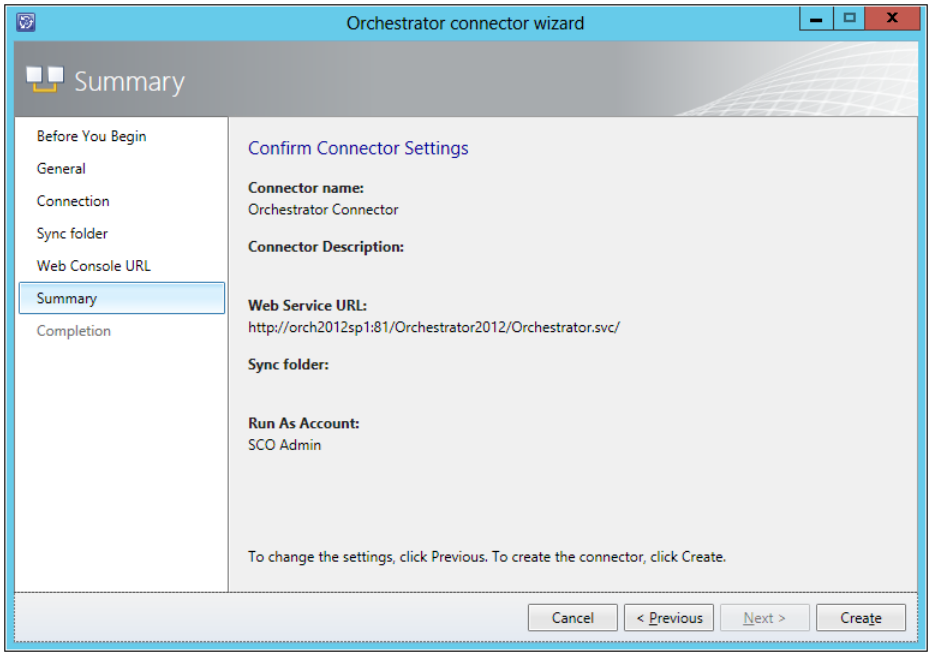


FIGURE 7-10 The Summary page in the Orchestrator Connector Wizard.

Next, highlight the newly created connector and from the Task pane, click Synchronize Now to kick off a manual synchronization. After the synchronization completes you can navigate to Library | Runbooks to see all the runbook metadata that was synchronized. An example of this is shown in Figure 7-11.

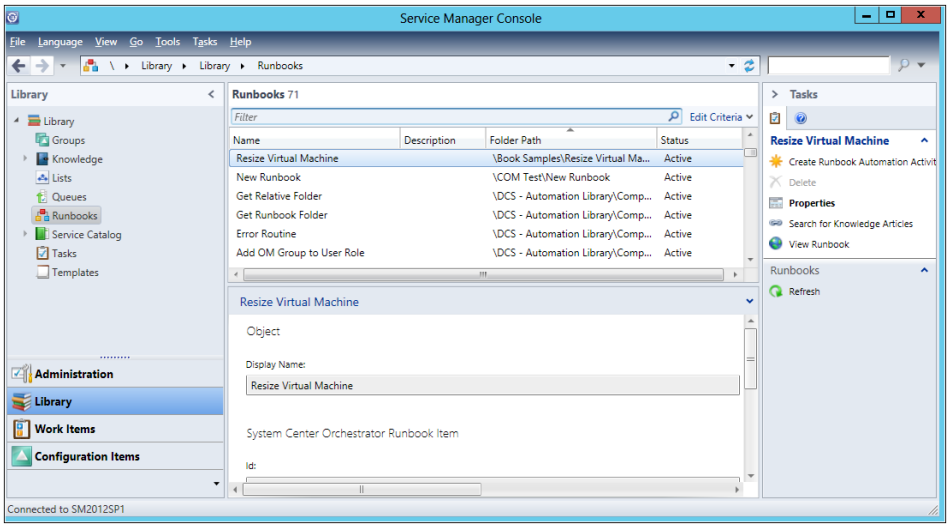


FIGURE 7-11 A listing of runbooks synchronized from Orchestrator.

Create a runbook automation activity template

In the Service Manager Console, select the Library tab from the navigation pane. Select Templates from the tree control and from the Tasks pane, click Create Template. In the Create Template dialog box shown in Figure 7-12, give your template a name and description. Then choose Runbook Automation Activity for the class. Lastly, create a new management pack by clicking New and filling in the information required. Click OK to open the Runbook Activity Template window.

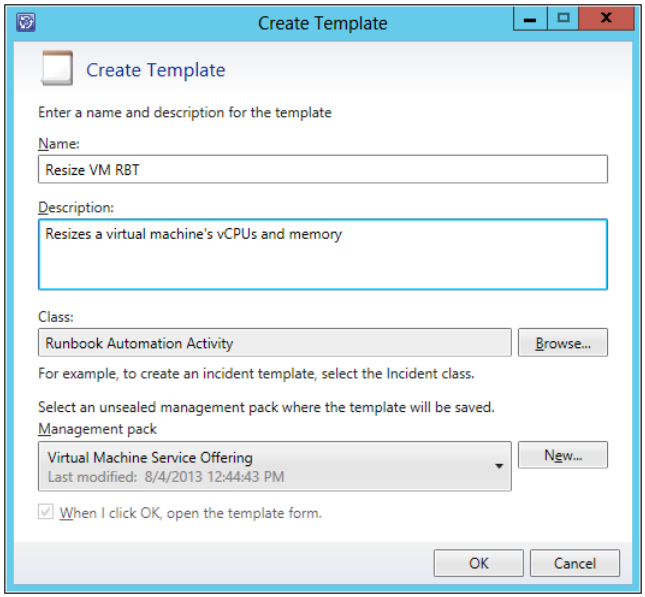


FIGURE 7-12 The Create Template dialog box.

On the General tab, fill in the information as needed, making sure to select the Is Ready For Automation check box. If this is not selected, the runbook will not execute. This is shown in Figure 7-13. When this page is complete, click the Runbook tab.

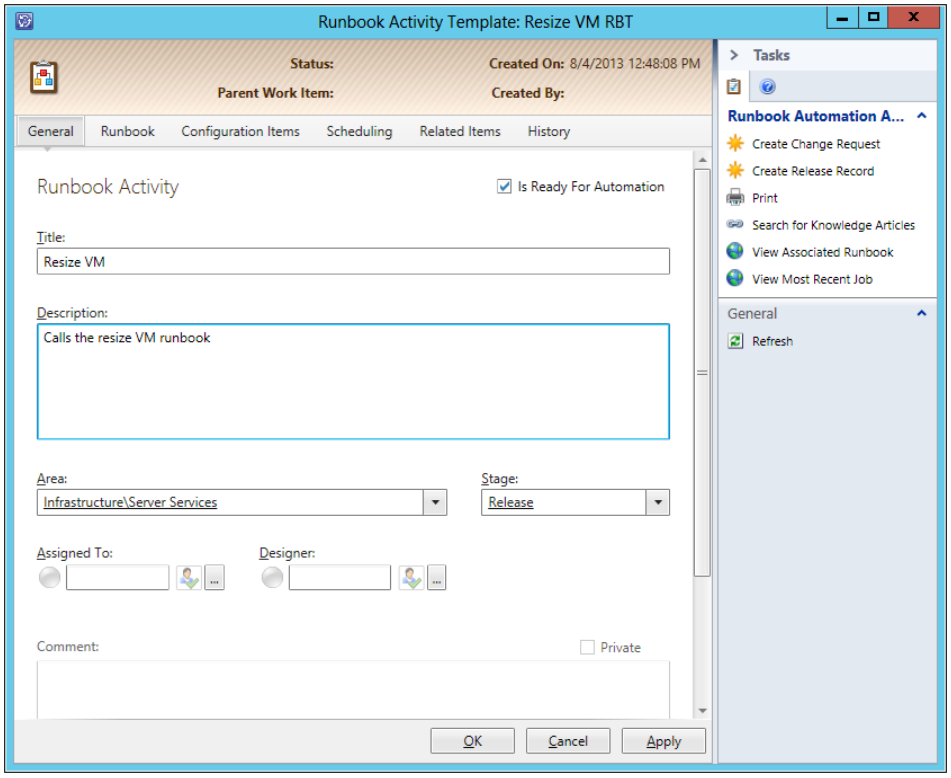


FIGURE 7-13 General tab of the runbook activity template.

Select the runbook by clicking the Select button and choosing the runbook from the list supplied. In the Parameter Mapping section, click the Edit Mapping button for the SM Activity ID. Expand the Object node, select Id, and click Close to save. This is shown in Figure 7-14. Click OK to save the runbook template.

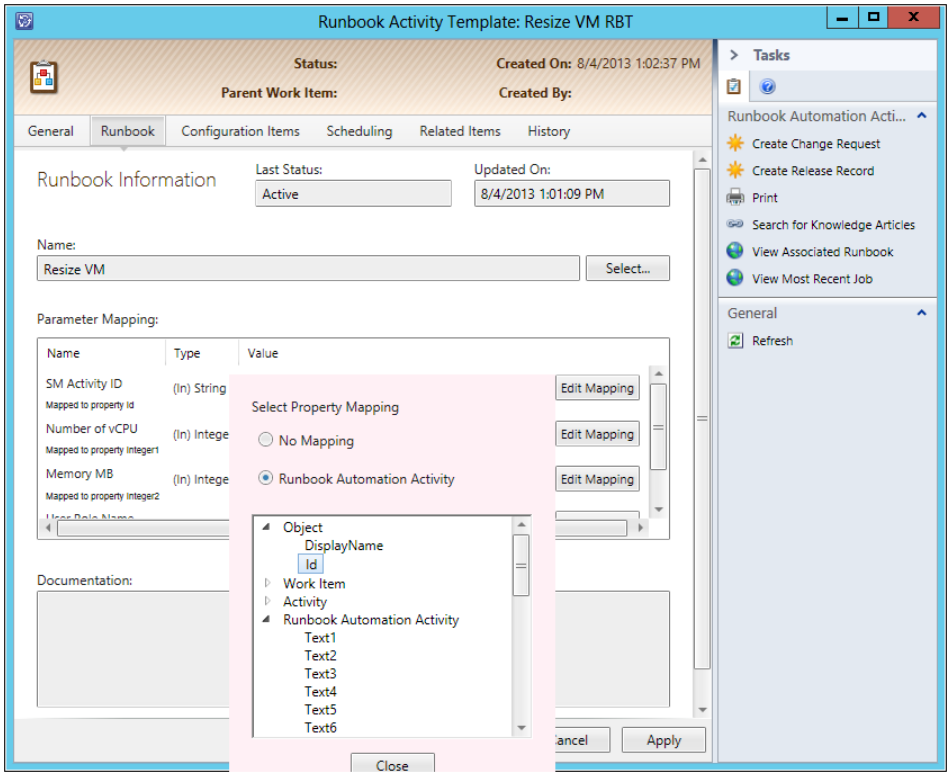


FIGURE 7-14 The Runbook tab of the runbook activity template.

Create a service request template

Once again, click Create Template from the Task pane to create a service request template. On the Create Template dialog, give your template a name and description. Then choose Service Request for the Class by clicking Browse and selecting the proper class. Lastly, select the management pack you created in the previous section. This is shown in Figure 7-15. Click OK to bring up the Service Request Template window.

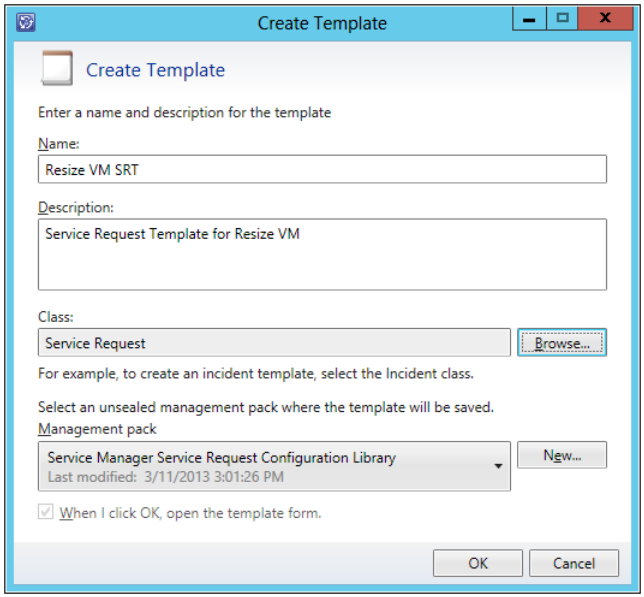


FIGURE 7-15 A view of the Create Template dialog box for creating a service request template.

Give the template a title and fill out the other fields on the General tab as needed. This is shown in Figure 7-16.

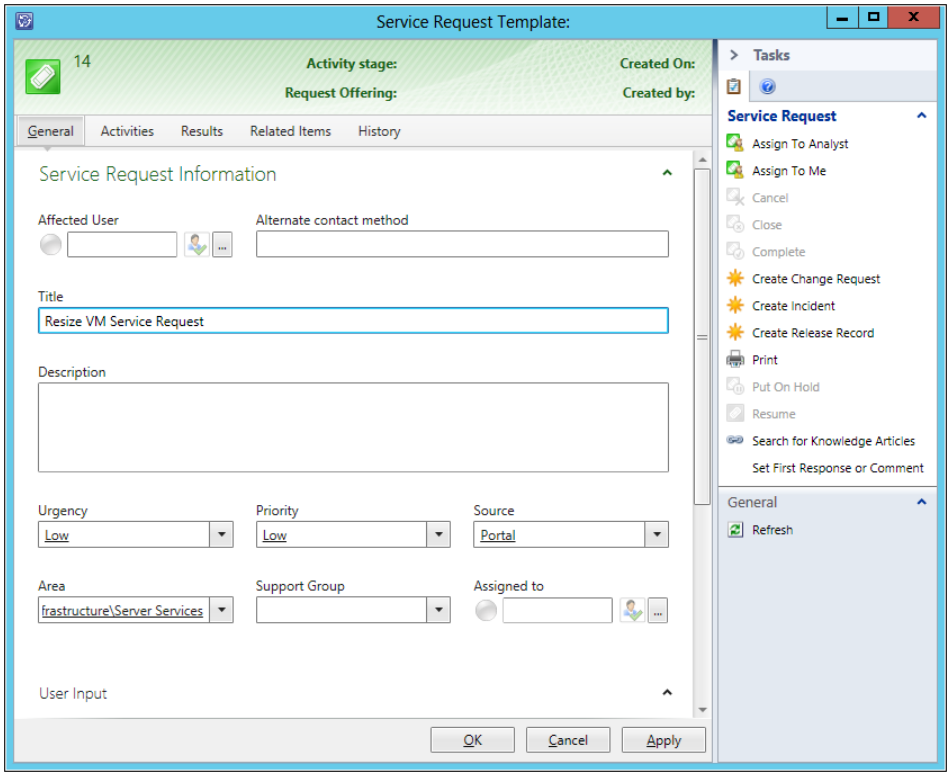


FIGURE 7-16 The General tab of the service request template.

Next, click the Activities tab. Click the green plus (+) icon to add an activity. Select the Resize VM template you created earlier. Click OK when the template window opens. This is shown in Figure 7-17. For this example, this will be the only activity in the request but understand that other activities such as review activities could have been added to allow for human review before allowing the runbook to run. Click OK to close the Service Request Template window.

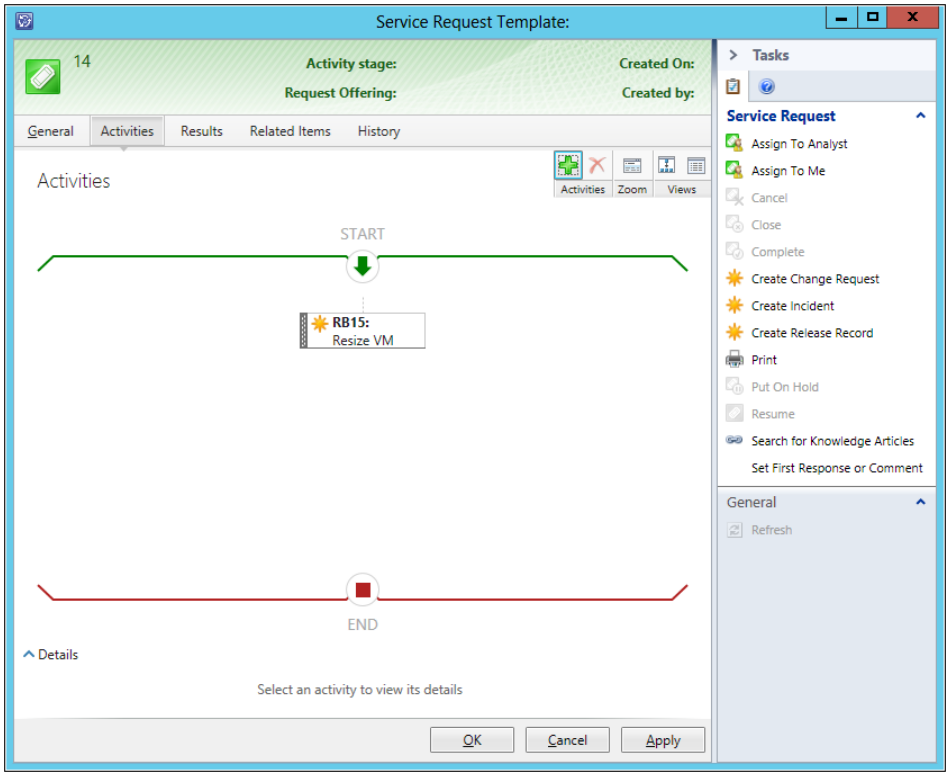


FIGURE 7-17 The Activities tab of the service request template

Create a request offering

Now that both of the templates have been created, it is now time to create the request offering. To do this, from the Service Manager Console, in the Library view, expand the service catalog and click Request Offerings. Next, from the Tasks pane, click Create Request Offering to open the Create Request Offering Wizard. Click Next to continue.

On the General page, enter in a title and description for your request offering. Select the Resize VM service request template by clicking Select Template and choosing the appropriate template, as shown in Figure 7-18. Click Next to continue on to the User Prompts page.

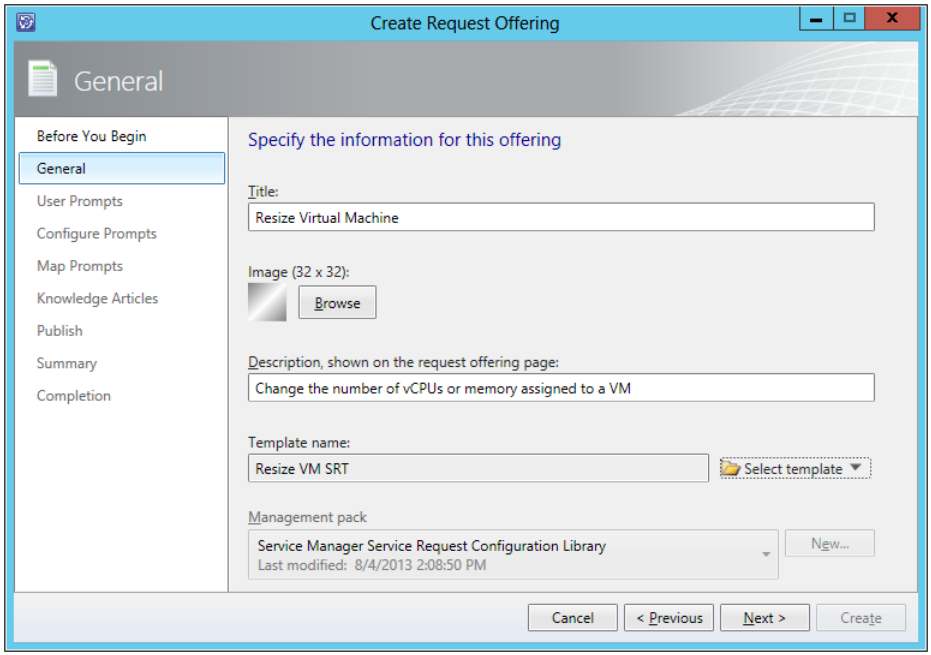


FIGURE 7-18 The General page of the Create Request Offering Wizard.

On the User Prompts page, enter in any form instructions and then enter in the prompts as shown in Figure 7-19. Click Next to continue on to the Configure Prompts page.

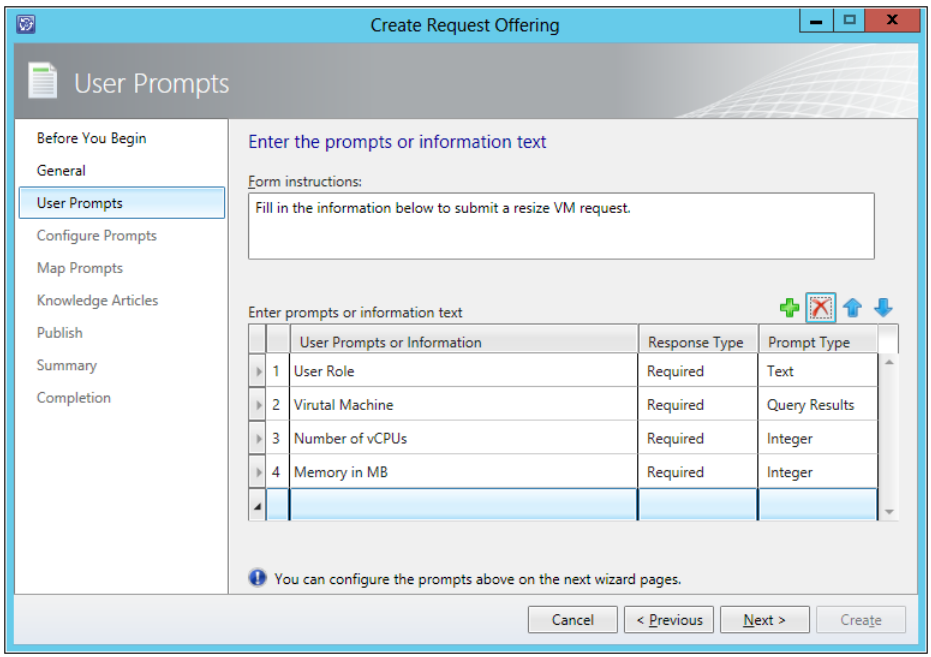


FIGURE 7-19 TheUser Prompts page in the Create Request Offering Wizard.

Click Virtual Machine and click the Configure button. This launches the Configure Query Results Wizard. Select the Virtual Machine class by changing the scope to All Basic Classes as shown in Figure 7-20. Click Configure Criteria (optional) to move onto the next tab.

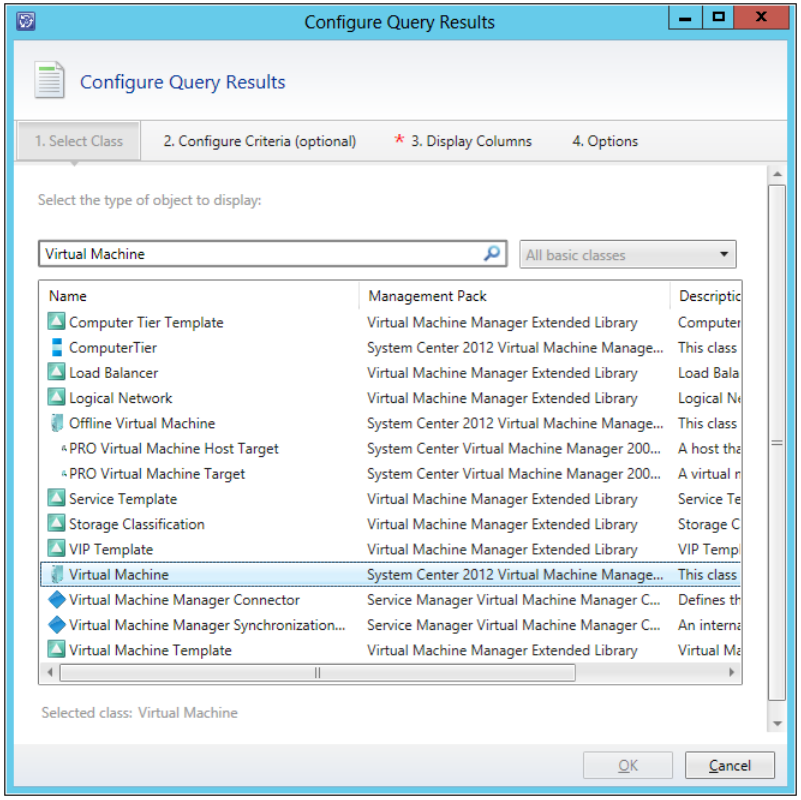


FIGURE 7-20 The Select Class tab in the Configure Query Results Wizard.

Find the Owner User Role field in the list of fields and click Add Constraint to add the field to the Criteria section. Next, change the operator to equals and then use the Set Token to select the User Role prompt. This allows the request form to filter the list of displayed virtual machines to only those VMs owned by the entered user role. This is shown in Figure 7-21. Click Display Columns to move to the next page.

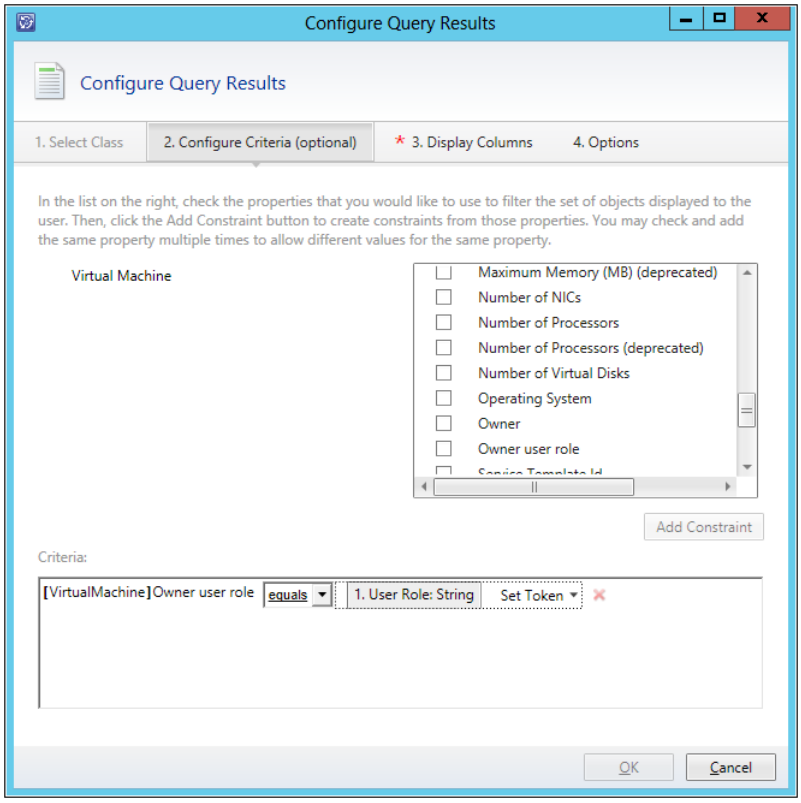


FIGURE 7-21 The Configure Criteria tab of the Configure Query Results Wizard.

Next, choose the properties you want displayed on the form. As shown in Figure 7-22, choose Display Name, Number Of Processors, and Total RAM (MB). You can rename Display Name to Virtual Machine Name to aid requestors in understanding the data returned. Click Options to go to the last page of the wizard.

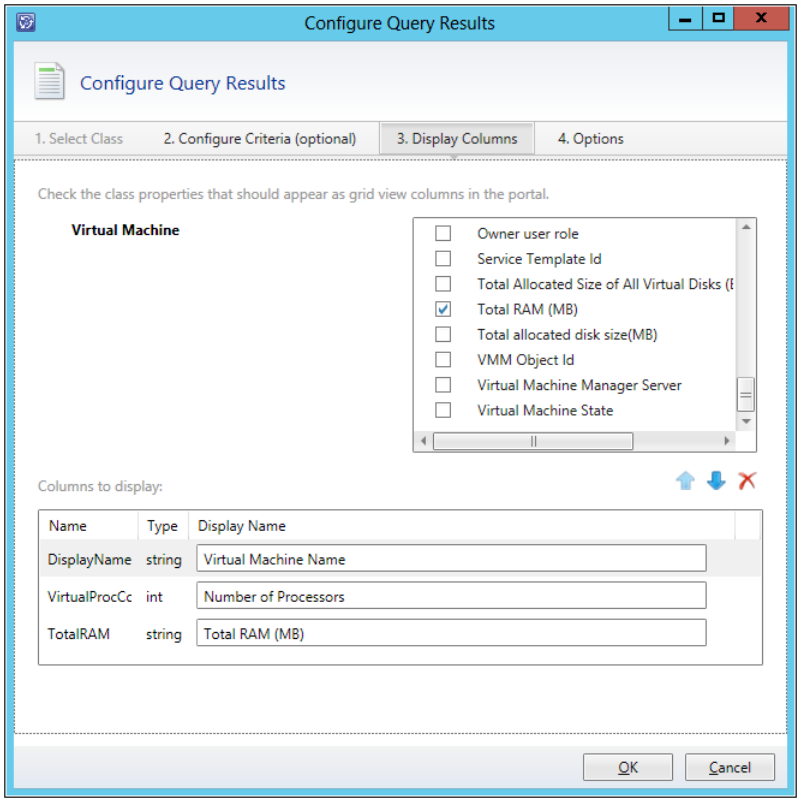


FIGURE 7-22 The Display Columns tab of the Configure Query Results Wizard.

On the Options page, select the check box for Add User-Selected Objects To Template Object As Related Items and select the Resize VM Runbook Automation Activity. This is shown in Figure 7-23. Click OK to close the wizard and return to the Create Request Offering wizard. You can optionally configure appropriate range values for the vCPUs and memory prompts by clicking Configure for each of the prompts. Click Next to continue to the Map Prompts page.

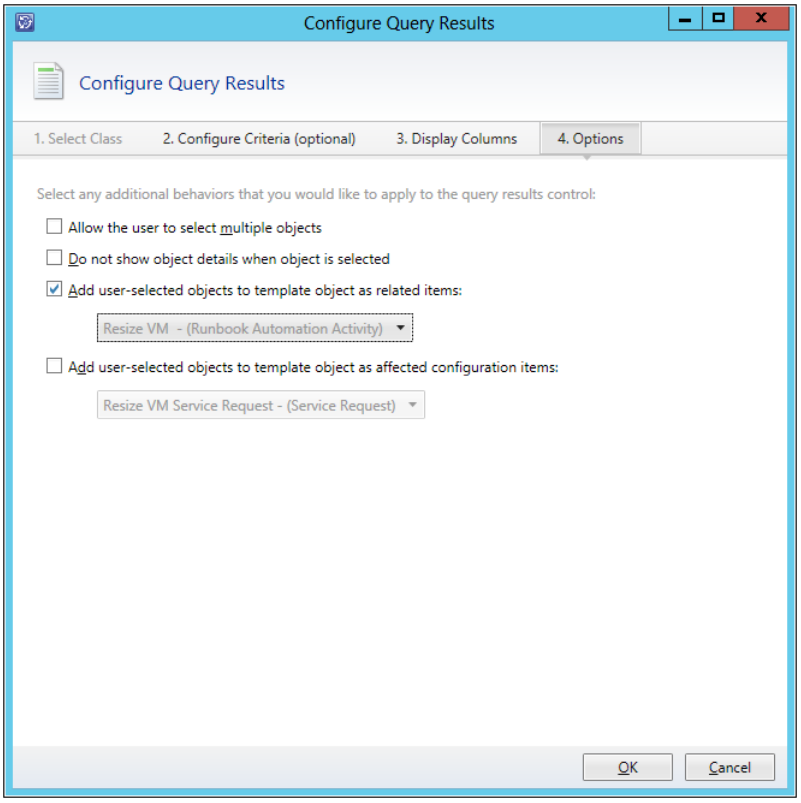


FIGURE 7-23 The Options tab of the Configure Query Results Wizard.

On the Map Prompts page, select the Resize VM Runbook Automation Activity and then map the prompts as shown in Figure 7-24. This maps the user input directly to the Initialize Data activity inputs. Click Next twice to move to the Publish page.

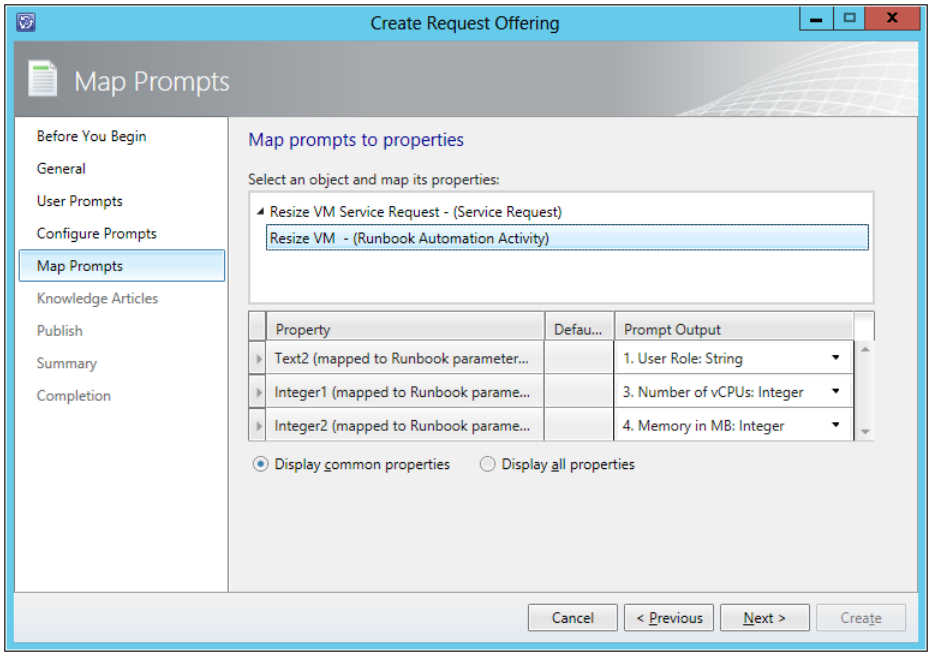


FIGURE 7-24 The Map Prompts page of the Create Request Offering Wizard.

On the Publish page, change the Offering status to published and then click Next to move on to the Summary page. Click Create to create the request offering and then click Close to close the wizard.

Create a service offering

The last step in the process of creating a service request customers can initiate on the Service Manager Portal is to create the containing service offering. To do this, from the Service Manager Console, in the Library view, expand the service catalog and click Service Offerings. Next, from the Tasks pane, click Create Service Offering to open the Create Service Offering Wizard. Click Next to continue.

On the General page give your service offering a title, category, language, overview, and description. Choose the management pack you created earlier and click Next to continue. This is shown in Figure 7-25. Continue clicking Next until you get to the Request Offering page.

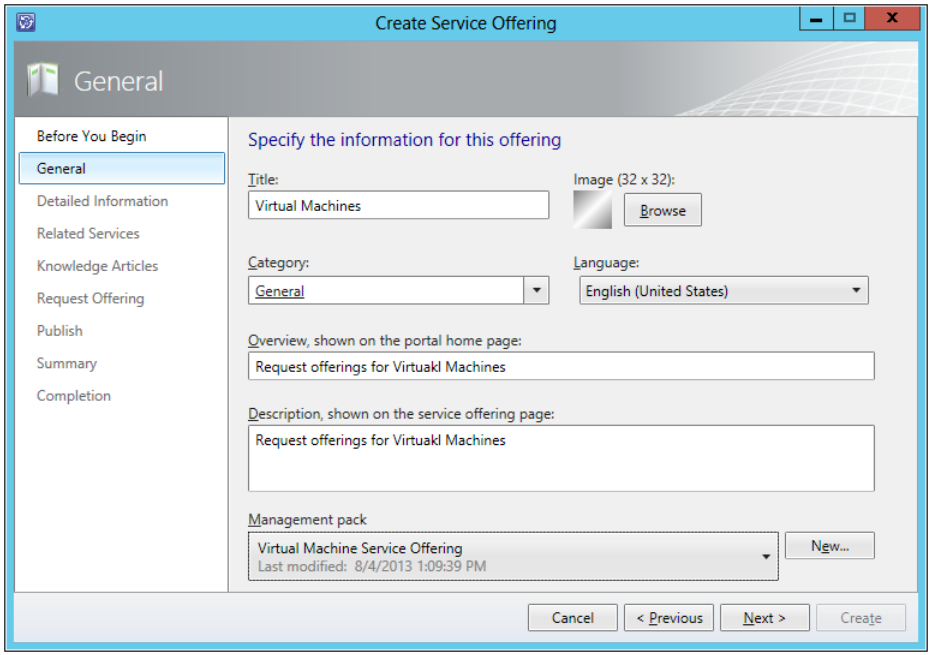


FIGURE 7-25 The General page of the Create Service Offering Wizard.

On the Request Offering page, add your newly created request offering by clicking Add and selecting the appropriate request offering. This is shown in Figure 7-26. Click Next to continue to the Publish page.

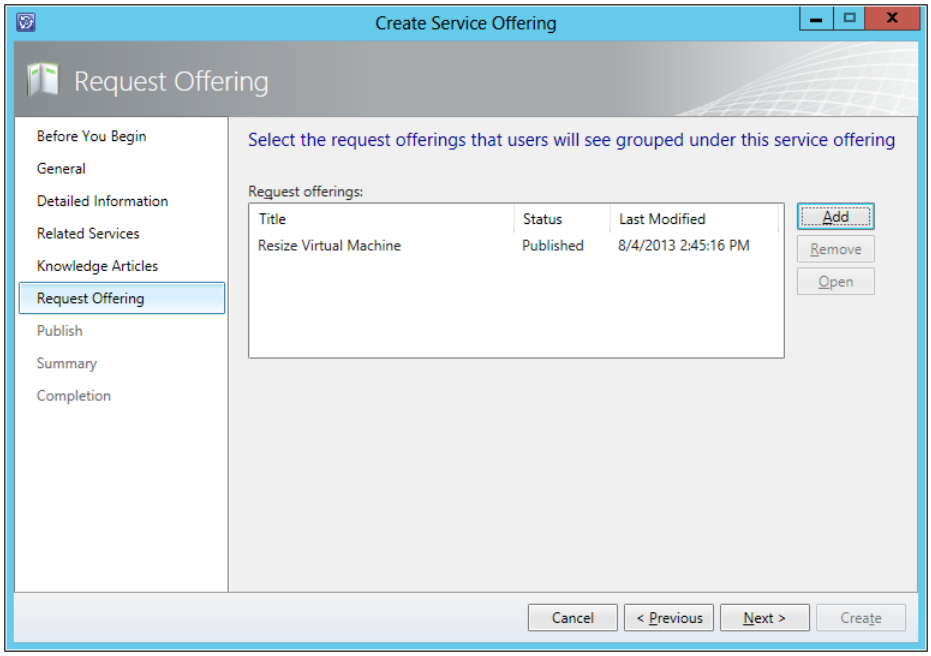


FIGURE 7-26 The Request Offering page of the Create Service Offering Wizard.

On the Publish page, change the offering status to publish and click Next to continue. Click Create to create the service offering and click Close to close the wizard. The service offering and the associated request offering should now be live on the Service Manager Portal.

APPENDIX A

Windows PowerShell source code for core component runbooks

Get Runbook Path

```
$Url = "{Orchestrator Web Service URL from "Initialize Data"}"
$ActivityID = "{Runbook Activity ID from "Initialize Data"}"
$RootPath = "{Root Path from "Initialize Data"}"

$ErrorState = 0
$ErrorMessage = ""
$Action = "Get Runbook Path"
$Trace = "Begin '$Action' `r`n"

try
{
    $Trace += "Calling Get-ActivityRunbookPath with URL '$Url' and ActivityID
'$ActivityID'. `r`n"
    if(!($Url.EndsWith('/') -Or $Url.EndsWith('\'))) {
        $Url += "/"
    }
    $Url = $Url + "Activities(guid'$ActivityID')/Runbook"
    $Trace += "Completed URL: $Url `r`n"
    $URI = New-Object System.Uri($Url,$true)

    $counter = 0
    $max = 6
    $completed = $false
    while ($counter -le $max -and !$completed)
    {
        try
        {
            $counter ++;
            #Create a request object using the URI
            $request = [System.Net.HttpWebRequest]::Create($URI)
```

```
            #Build up a nice User Agent
            $request.UserAgent = $(
                "{0} (PowerShell {1}; .NET CLR {2}; {3})" -f $UserAgent,
$(if($Host.Version){$Host.Version}else{"1.0"}),
                [Environment]::Version,
                [Environment]::OSVersion.ToString().Replace("Microsoft Windows ", "Win")
                )

            $request.UseDefaultCredentials = $true

            [System.Net.HttpWebResponse] $response = [System.Net.HttpWebResponse]
$request.GetResponse()
            $completed = $true
        }
        catch
        {
            if ($counter -eq $max)
            {
                Throw "$($_.Exception.Message)"
            }
            else
            {
                $Trace += [DateTime]::Now.ToString() + " $($_.Exception.Message)...
Trying again.`r`n"
                sleep 10
            }
        }
    }

    $reader = [IO.StreamReader] $response.GetResponseStream()

    [xml]$output = $reader.ReadToEnd()
    $reader.Close()

    $response.Close()
    $path = $RootPath + $output.Entry.Content.properties.Path
    $Trace += "Path to return: $path `r`n"
    $Trace += "Completed $Action `r`n"
}
catch
{
   $ErrorState = 2;
   $ErrorMessage = $error[0].Exception.tostring()
   $Trace += "Error running '$Action'. `r`n"
}
```

```
finally
{
   $Trace += "Exiting  '$Action' `r`n"
   $Trace += "ErrorState:   $ErrorState`r`n"
   $Trace += "ErrorMessage: $ErrorMessage`r`n"
}
```

Get Relative Folder

```
$startPath = "{Start Folder from "Initialize Data"}"
$ancestors = {Ancestor Count from "Initialize Data"}
$childPath = "{Child Folder Path from "Initialize Data"}"

$Action = "Get Relative Folder"
$ErrorState = 0
$ErrorMessage = ""
$Trace = "Starting $Action`n`n"

try
{
    if (!$childPath.StartsWith("\"))
    {
       $childPath = "\" + $childPath
    }

    $Trace += "Validating $startPath exists `r`n"
    $directory = [System.IO.DirectoryInfo]$startPath

    $Trace += "Getting $ancestors directory parents`r`n"
    while ($ancestors -gt 0)
    {
        $directory = $directory.Parent
        $ancestors--
    }

    $Trace += "Appending $childPath to new path and validating existance`r`n"
    $newPath = $directory.FullName + $childPath
    $newDirectory = [System.IO.DirectoryInfo]$newPath
    if (!$newDirectory.Exists)
    {
        throw "New path not found! $newPath"
    }

    $Trace += "Completing $Action`r`n"
```

```
    $DirectoryPath = $newDirectory.FullName
}
catch
{
    $ErrorState = 2
    $ErrorMessage = $error[0].Exception.tostring()
    $Trace += "Error caught in $Action`r`n"
}
finally
{
    $Trace += "Exiting $Action `r`n"
    $Trace += "ErrorState:   $ErrorState`r`n"
    $Trace += "ErrorMessage: $ErrorMessage`r`n"
}
```

APPENDIX B

Steps to set up VMM to SM integration

This appendix provides information that will help you set up Microsoft System Center Service Manager to sync System Center Virtual Machine Manager (VMM) PRO data. This synchronization brings data about hosts, virtual machines, and relationships from VMM in to the Service Manager Configuration Management Database (CMDB). This is useful when creating runbooks that will combine both Service Manager actions as well as VMM actions. The VMM data is synchronized to Service Manager through System Center Operations Manager, meaning there is native integration between VMM and Operations Manager, then a connector is configured between Operations Manager and Service Manager. The end result is data synchronization from VMM to Operations Manager to Service Manager.

Management packs

On your Service Manager server, import:

- The following management pack: *http://blogs.technet.com/b/servicemanager/archive/2012/02/09/faq-installing-all-the-prerequisite-mps-for-the-cloud-services-management-pack.aspx*
- All of the management packs found on the Service Manager server under \Program Files\Microsoft System Center 2012\Service Manager\Operations Manager 2012 SP1 Management Packs
- The System.NetworkManagement.Library.mp found on the System Center Operations Manager install media under Management Packs
- The management packs from the Virtual Machine Manager server under \Program Files\Microsoft System Center 2012\Virtual Machine Manager\ManagementPacks.

Create an Operations Manager CI Connector

In the Service Manager console, click Administration. In the Administration pane, expand Administration, and then click Connectors. In the Tasks pane, under Connectors, click Create Connector, and then click Operations Manager CI Connector as shown in Figure B-1.

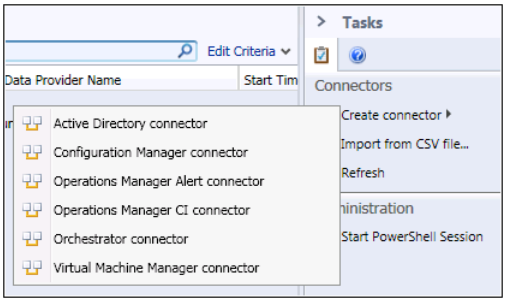


FIGURE B-1 Creating the Operations Manager CI Connector.

On the Before You Begin page, click Next. On the General page, in the Name box, type the name such as OpsMgrCon. Make sure that the Enable check box is selected, and then click Next as shown in Figure B-2.

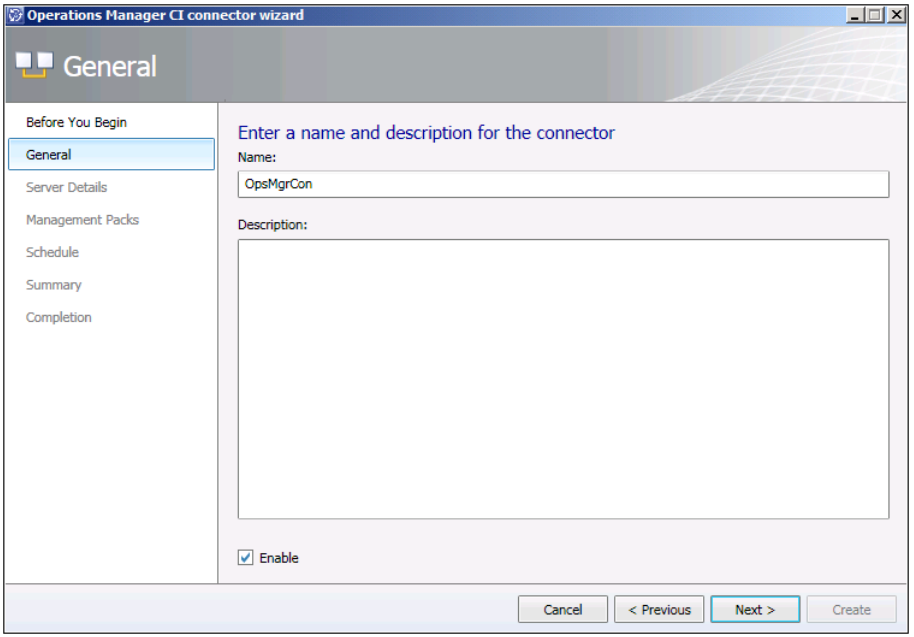


FIGURE B-2 The General page of the Operations Manager CI Connector Wizard.

On the Server Details page, in the Server Name box, type the name of the server that is hosting the Operations Manager root management server. Under Credentials, click New, as shown in Figure B-3.

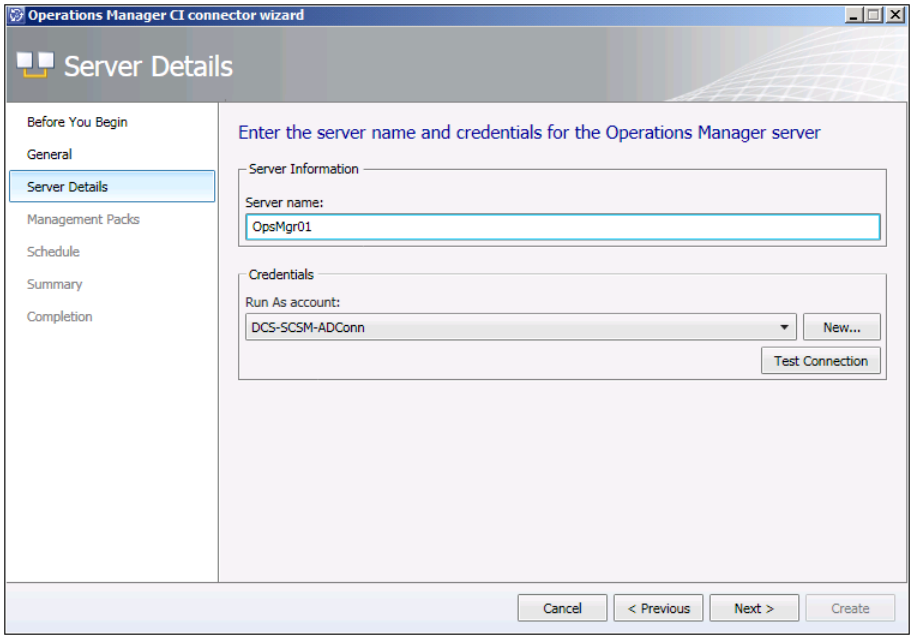


FIGURE B-3 The Server Details page of the Operations Manager CI Connector Wizard.

In the User Name, Password, and Domain boxes, type the credentials for the Operations Manager Connector CI Account (for example, DCS-SCSM-OMCI), and then click OK as shown in Figure B-4.

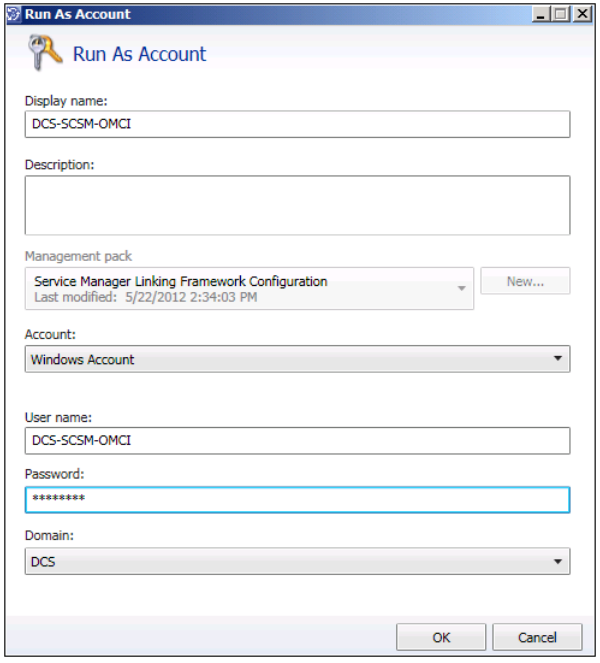


FIGURE B-4 The Run As Account for the Operations Manager CI Connector Wizard.

On the Server Details page, click Test Connection. You will receive the following confirmation message: "The connection to the server was successful." This is shown in Figure B-5. Click OK, and then click Next.

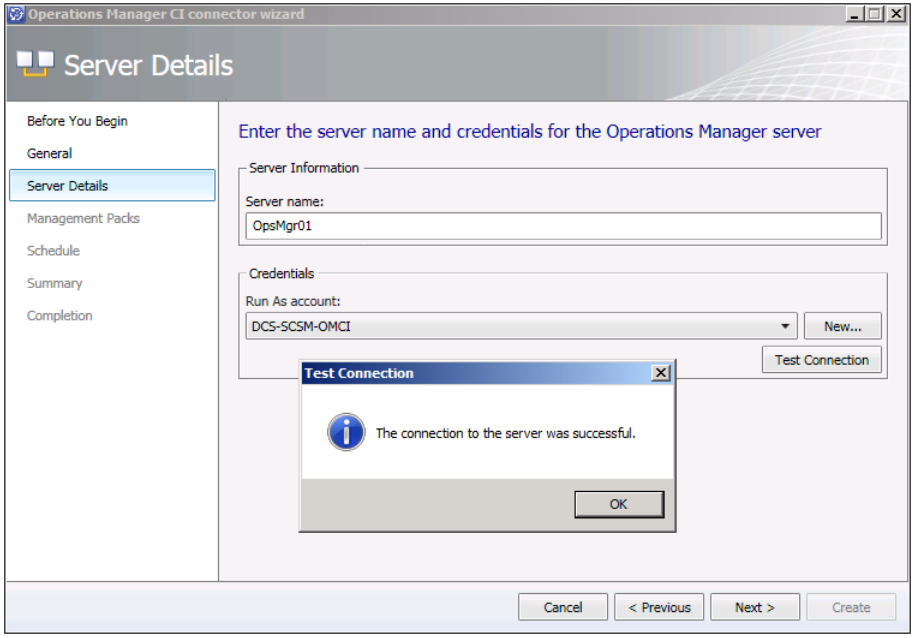


FIGURE B-5 Testing the connection in the wizard.

On the MP Selection page, click Select only Microsoft.SystemCenter.VirtualMachineManager.2012.Discovery and then click Next as shown in Figure B-6.

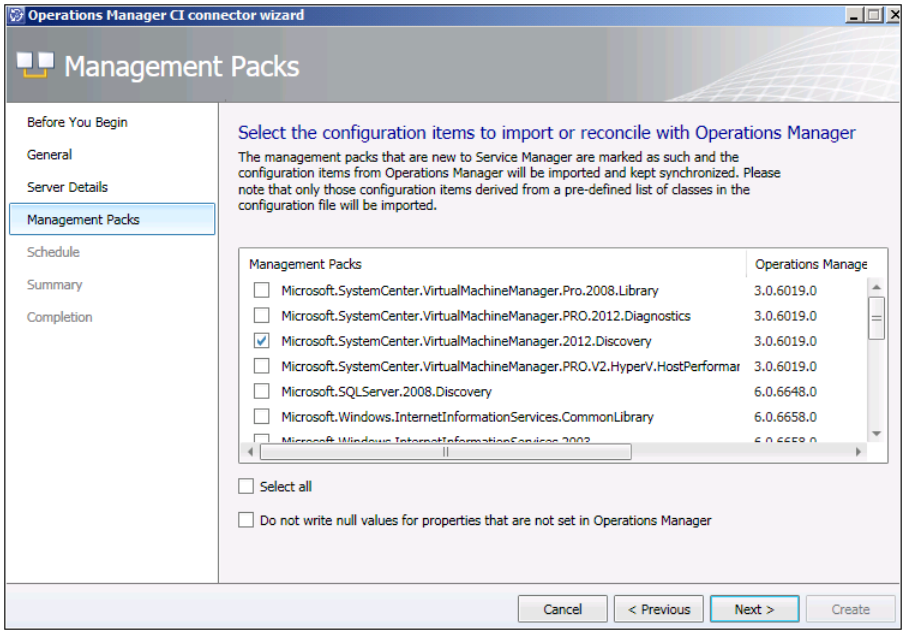


FIGURE B-6 The Management Pack page of the Operations Manager CI Connector Wizard.

On the Schedule page, click Next, as shown in Figure B-7.

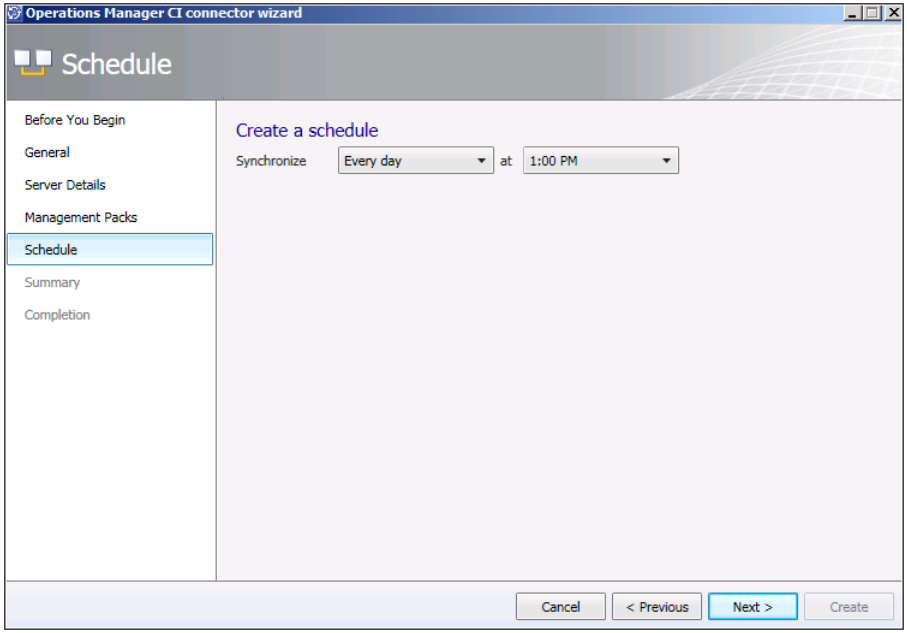


FIGURE B-7 The Schedule page of the Operations Manager CI Connector Wizard.

On the Summary page, click Create and on the Completion page, click Close. In the Connectors pane, select the Operations Manager connector OpsMgrCon. In the Tasks pane, under the connector name, click Synchronize Now. In the Synchronize Now dialog box, click OK. Now VMM data should be syncing from VMM to Service Manager.

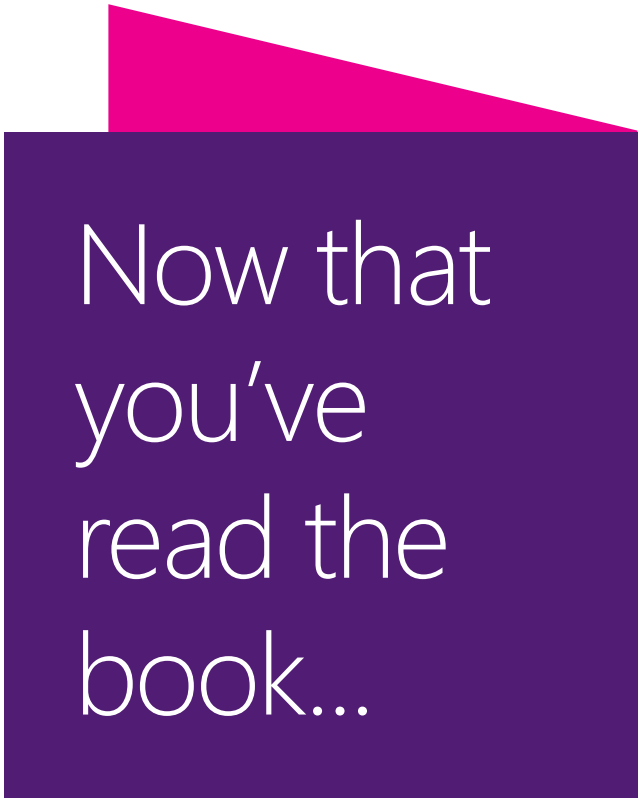
About the authors

DAVID ZIEMBICKI is a Senior Architect in Microsoft Services' Americas Office of the CTO. David's areas of expertise include private & hybrid cloud, virtualization, and datacenter automation. He has been a leading infrastructure architect across hundreds of strategic projects with public sector and Fortune 500 customers in multiple industries throughout his IT career. David is a lead architect for Microsoft's Datacenter Services Portfolio and the Microsoft Private Cloud Fast Track program. He is a course instructor, published author, and regular speaker on Microsoft Cloud, Datacenter, and Infrastructure solutions.

David's blog can be found at *http://davidzi.com/blog* and he is on Twitter at *http://www.twitter.com/davidzi*.

AARON CUSHNER is a Lead Architect in the Datacenter Program Team, Microsoft Services HQ. Aaron's nearly 14-year experience at Microsoft has spanned database and business intelligence consulting, .Net development, and for the past 5 years has been focused on datacenter automation and self-service for private and hybrid cloud solutions. He led the development of a private cloud reference implementation as part of Microsoft's Datacenter Services Portfolio that has helped Microsoft Services deliver Infrastructure as a Service private clouds to our global customers. Aaron is a course instructor and regular speaker at internal conferences focusing on datacenter self-service and automation.

ANDREAS RYNES is a Lead Architect in the Datacenter Program Team, Microsoft Services HQ. He's working on datacenter solutions focusing on automation and management. Andreas has a background of more than 10 years in software engineering and architecture before he started working on infrastructure, virtualization and management solutions over 5 years ago. Andreas supports designing and building datacenter solutions and helps defining architectures for successful implementations of private, public, and hybrid clouds for customers worldwide. Andreas is also a course instructor, and he is a regular speaker at public and internal conferences.



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