You can find the most up-to-date technical documentation on the VMware Web site at:
http://www.vmware.com/support/
The VMware Web site also provides the latest product updates.
If you have comments about this documentation, submit your feedback to:
docfeedback@vmware.com

© 2009 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at http://www.vmware.com/go/patents.
VMware, the VMware “boxes” logo and design, Virtual SMP, and VMotion are registered trademarks or trademarks of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.
Contents

Updated Information  5

About This Book    7

1 Introduction to VMware vCenter Orchestrator    9
   Key Features of the Orchestrator Platform       9
   Orchestrator User Roles and Related Tasks      10
   Orchestrator Architecture                     11

2 Developing Workflows    13
   Principal Phases in the Workflow Development Process    14
   Testing Workflows During Development     14
   Workflow Workbench                         15
   Provide General Workflow Information       16
   Defining Attributes and Parameters         17
   Workflow Schema                            19
   Obtaining Input Parameters from Users When a Workflow Starts  35
   Requesting User Interactions While a Workflow Runs  40
   Calling Workflows Within Workflows          42
   Developing Long-Running Workflow Elements   47
   Configuration Elements                     52
   Workflow User Permissions                   53
   Running Workflows                          54
   Develop a Simple Example Workflow           57
   Develop a Complex Workflow                  80

3 Developing Actions  101
   Reusing Actions                            101
   Access the Actions View                    101
   Components of the Actions View             102
   Creating Actions                           102

4 Scripting  105
   Orchestrator Elements that Require Scripting 105
   Using the Orchestrator API                  106
   Exception Handling Guidelines              111
   Orchestrator JavaScript Examples           112

5 Creating Packages  119
   Create a Package                           119
   Set User Permissions on a Package          120
Updated Information

This vCenter Orchestrator Developer’s Guide is updated with each release of the product or when necessary.

This table provides the update history of the VMware vCenter Orchestrator Developer’s Guide.

<table>
<thead>
<tr>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
</table>
| EN-000129-01 |  - Changed the order of the information in the sections “Web View Overview,” on page 201 and “Web View Components,” on page 203.  
  - Updated the filenames and paths of the Refactor Tutorial example throughout the sections “Refactoring Packages with the Basic Refactoring Workflow,” on page 216 and “Refactoring Packages with the Advanced Refactoring Workflows,” on page 220 to reflect changes in the bundle of sample applications.  
  - Updated the filenames and paths of the Solar System example throughout the section “Create an Orchestrator Plug-In,” on page 142 to reflect changes in the bundle of sample applications. |

| EN-000129-00 | Initial release. |
About This Book

The VMware vCenter Orchestrator Developer’s Guide provides information and instructions about how to use the VMware vCenter Orchestrator platform to develop process-automation applications for virtual environments.

Intended Audience

This document is intended for developers who want to develop applications using the Orchestrator platform. Specifically, this document is intended for the following types of developer.

- Application developers who want to create new extensions to the Orchestrator platform.
- Scripting developers who want to create new building blocks to automate certain processes.
- Web service application developers who want to access these processes across a network, through technologies such as simple object access protocol (SOAP) and Web services definition language (WSDL).
- Web designers who want to create or customize Web front ends for these processes, using the Web 2.0 technologies.
- IT staff who want to automate processes to save time, to reduce risk and cost, and to comply with regulations or standard practices.

Example Applications

The examples applications which this document describes are available to download. You can download a ZIP file of examples from the Orchestrator documentation home page.

Document Feedback

VMware welcomes your suggestions for improving our documentation. If you have comments, send your feedback to docfeedback@vmware.com.
## Technical Support and Education Resources

The following technical support resources are available to you. To access the current version of this book and other books, go to [http://www.vmware.com/support/pubs](http://www.vmware.com/support/pubs).

### Online and Telephone Support

To use online support to submit technical support requests, view your product and contract information, and register your products, go to [http://www.vmware.com/support](http://www.vmware.com/support).

Customers with appropriate support contracts should use telephone support for the fastest response on priority 1 issues. Go to [http://www.vmware.com/support/phone_support.html](http://www.vmware.com/support/phone_support.html).

### Support Offerings

To find out how VMware support offerings can help meet your business needs, go to [http://www.vmware.com/support/services](http://www.vmware.com/support/services).

### VMware Professional Services

VMware Education Services courses offer extensive hands-on labs, case study examples, and course materials designed to be used as on-the-job reference tools. Courses are available onsite, in the classroom, and live online. For onsite pilot programs and implementation best practices, VMware Consulting Services provides offerings to help you assess, plan, build, and manage your virtual environment. To access information about education classes, certification programs, and consulting services, go to [http://www.vmware.com/services](http://www.vmware.com/services).
VMware vCenter Orchestrator is a development and process-automation platform that provides a library of extensible workflows to allow you to create and run automated, configurable processes to manage the VMware vCenter infrastructure.

Orchestrator exposes every operation in the vCenter Server API, allowing you to integrate all of these operations into your automated processes. Orchestrator also allows you to integrate with other management and administration solutions through its open plug-in architecture.

This chapter includes the following topics:
- “Key Features of the Orchestrator Platform,” on page 9
- “Orchestrator User Roles and Related Tasks,” on page 10
- “Orchestrator Architecture,” on page 11

### Key Features of the Orchestrator Platform

Orchestrator is composed of three distinct layers: an orchestration platform that provides the common features required for an orchestration tool, a plug-in architecture to integrate control of subsystems, and a library of preexisting processes. Orchestrator is an open platform that can be extended with new plug-ins and libraries, and can be integrated into larger SOAP architectures through a set of APIs.

The following list presents the key Orchestrator features.

**Persistence**
Production grade external databases are used to store relevant information, such as processes, states, and configuration information.

**Central management**
Orchestrator provides a central way to manage your processes. The application server-based platform, with full version history, allows you to have scripts and process-related primitives in one place. This way, you can avoid scripts without versioning and proper change control spread on your servers.

**Check-pointing**
Every step of a process is saved in the database, which allows you to restart the server without losing state and context. This feature is especially useful for long-running processes.

**Versioning**
All Orchestrator Platform objects have an associated version history. This feature allows basic change management when distributing processes to different project stages or locations.
Scripting engine

The Mozilla Rhino JavaScript engine provides a way to create new building blocks for Orchestrator Platform. The scripting engine is enhanced with basic version control, variable type checking, name space management and exception handling. It can be used in the following building blocks:

- Actions
- Workflows
- Policies

Workflow engine

The workflow engine allows you to capture business processes. It uses one of the following methods to create a step-by-step automation:

- Building blocks of the library
- Building blocks provided by the customer
- Plug-ins

Users, a schedule, or a policy can start workflows.

Policy engine

The policy engine allows monitoring and event generation to react to changing conditions. Policies can aggregate events from the platform or any of the plug-ins, which allows you to handle changing conditions on any of the integrated technologies.

Web 2.0 front end

The Web 2.0 front end allows new possibilities of expression and flexibility. It provides a library of user customizable components to access vCO orchestrated objects and uses Ajax technology to dynamically update content without reloading complete pages.

Security

Orchestrator provides the following advanced security functions:

- Public Key Infrastructure (PKI) to sign and encrypt content imported and exported between servers
- Digital Rights Management (DRM) to control how exported content might be viewed, edited and redistributed
- Secure Sockets Layer (SSL) encrypted communications between the desktop client and the server and HTTPS access to the Web front end.
- Advanced access rights management to provide control over access to processes and the objects manipulated by these processes.

Orchestrator User Roles and Related Tasks

vCenter Orchestrator provides different tools and interfaces based on the specific responsibilities of the three global user roles: Administrators, Developers, and End Users.

Administrators

This role has full access to all of the Orchestrator platform capabilities. Basic administrative tasks include the following items:

- Installing and configuring Orchestrator
- Managing access rights for Orchestrator and applications
- Importing and exporting packages
- Enabling and disabling Web views
- Running workflows and scheduling tasks
- Managing version control of imported elements

**Developers**

Users in this role are granted access to the Orchestrator client interface and have the following responsibilities:

- Creating applications to extend the Orchestrator platform functionality
- Automating processes by customizing existing workflows and creating new workflows
- Customizing Web front ends for these processes, using Web 2.0 technologies

**End Users**

Users in this role are granted access to only the Web front end. They can run and schedule workflows and policies.

**Orchestrator Architecture**

Orchestrator contains a workflow library and workflow engine to allow you to create and run workflows that automate orchestration processes. You run workflows on the objects of different technologies that Orchestrator accesses through a series of plug-ins.

Orchestrator provides a standard set of plug-ins, including a plug-in to VMware vCenter Server 4.0, to allow you to orchestrate tasks in the different environments that the plug-ins expose.

Orchestrator also presents an open architecture to allow you to plug in external third-party applications to the orchestration platform. You can run workflows on the objects of the plugged-in technologies that you define yourself. Orchestrator connects to a directory services server to manage user accounts, and to a database to store information from the workflows that it runs. You can access Orchestrator and the workflows and objects it exposes through the Orchestrator client interface, through a Web browser, or through Web services.

Figure 1-1 shows the architecture of Orchestrator.

**Figure 1-1. VMware vCenter Orchestrator Architecture**
You develop workflows in the Orchestrator client interface. Workflow development involves using the workflow workbench, the built-in Mozilla Rhino JavaScript scripting engine, and the Orchestrator API.

- **Principal Phases in the Workflow Development Process** on page 14
  The process for developing a workflow involves a series of phases.

- **Testing Workflows During Development** on page 14
  You can test workflows at any point during the development process, even if you have not completed the workflow or included an end element.

- **Workflow Workbench** on page 15
  You create and edit workflows by using the workflow workbench. The workflow workbench is the Orchestrator client's IDE for developing workflows.

- **Provide General Workflow Information** on page 16
  You provide a workflow name and description, define attributes and certain aspects of workflow behavior, set the version number, check the signature, and set user permissions in the **General** tab in the workflow workbench.

- **Defining Attributes and Parameters** on page 17
  After you create a workflow, you must determine the workflow's global attributes and input and output parameters.

- **Workflow Schema** on page 19
  A workflow schema is a graphical representation of a workflow that shows the workflow as a flow diagram of interconnected workflow elements.

- **Obtaining Input Parameters from Users When a Workflow Starts** on page 35
  If a workflow requires input parameters, it opens a dialog box in which users enter the required input parameter values when it runs. You can organize the content and layout, or presentation, of this dialog box in **Presentation** tab in the workflow workbench.

- **Requesting User Interactions While a Workflow Runs** on page 40
  A workflow can sometimes require additional input parameters from an outside source while it runs. These input parameters can come from another application or workflow, or the user can provide them directly.

- **Calling Workflows Within Workflows** on page 42
  Workflows can call on other workflows during their run. A workflow can start another workflow either because it requires the result of that workflow as an input parameter for its own run, or it can start a workflow and let it continue its own run independently. Workflows can also start a workflow at a given time in the future, or start multiple workflows simultaneously.
A workflow in a waiting state consumes system resources by constantly polling the object from which it is awaiting a response. If you know that a workflow element will potentially wait for a long time before it receives the response it requires, you can implement that workflow element as a long-running element.

A configuration element is a list of attributes you can use to configure constants across a whole Orchestrator server deployment.

A workflow defines levels of permissions that you can apply to users or groups.

A workflow runs according to a logical flow of events.

Developing a simple example workflow demonstrates the most common steps in the workflow development process.

Developing a complex example workflow demonstrates the most common steps in the workflow development process, as well as more advanced scenarios, such as creating custom decisions and loops.

**Principal Phases in the Workflow Development Process**

The process for developing a workflow involves a series of phases. The order in which you perform the tasks that developing a workflow involves generally conforms to the following sequence of phases.

1. Provide general information about the workflow.
2. Create the input parameters.
3. Create the logical flow of the workflow by laying out and linking the schema.
4. Bind the input and output parameters of each element to workflow attributes, creating the necessary parameters and attributes as you define each element.
5. Write any necessary scripts for scriptable task or custom decision elements.
6. Create the layout and behavior of the input parameters dialog box that the user sees when they run the workflow by creating the workflow presentation.
7. Validate the workflow.

**Testing Workflows During Development**

You can test workflows at any point during the development process, even if you have not completed the workflow or included an end element.

By default, Orchestrator checks that a workflow is valid before you can run it. You can deactivate automatic validation during workflow development, to run partial workflows for testing purposes.

*NOTE* Do not forget to reactivate automatic validation when you finish developing the workflow.
Procedure

1. In the Orchestrator client menu, click **Tools > User Preferences**.
2. Click the **Workflows** tab.
3. Uncheck the **Validate workflow before executing it** check box.

You deactivated automatic workflow validation.

**Workflow Workbench**

You create and edit workflows by using the workflow workbench. The workflow workbench is the Orchestrator client’s IDE for developing workflows.

You open the workflow workbench by editing an existing workflow.

**Create a Workflow**

You can create new workflows in the workflows hierarchical list in the Orchestrator client interface.

Procedure

1. In the Orchestrator client, click the **Workflows** view.
2. (Optional) Right-click the root of the workflows hierarchical list, or a category in the list, and select **Add Category** to create a new workflow category.
3. (Optional) Provide a name for the new category.
4. Right-click the new category or an existing category and select **New workflow**.
5. Give the new workflow a name and click **OK**.

The new empty workflow is created in the category you chose.

**What to do next**

You can edit the workflow.

**Edit a Workflow in the Workflow Workbench**

You edit a new or existing workflow by using the Orchestrator client's workflow workbench.

Procedure

1. In the Orchestrator client, click the **Workflows** view.
2. Expand the workflows hierarchical list to navigate to the workflow to edit.
3. Click the workflow to edit.
4. Open the workflow for editing by right-clicking the workflow and selecting **Edit**.

The workflow workbench opens, allowing you to edit the workflow.
**Workflow Workbench Tabs**

The workflow workbench consists of tabs in which you edit the different components of the workflows.

**Table 2-1. Workflow Palette Tabs**

<table>
<thead>
<tr>
<th>Tab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Edit the workflow name, provide a description of what the workflow does, set the version number, define the behavior of the workflow if the Orchestrator server restarts, and define the workflow’s global attributes.</td>
</tr>
<tr>
<td>Inputs</td>
<td>Define the parameters that the workflow requires when it runs. These input parameters are the data that the workflow processes. The workflow’s behavior changes according to these parameters.</td>
</tr>
<tr>
<td>Outputs</td>
<td>Define the values that the workflow generates when it completes its run. Other workflows or actions can use these values when they run.</td>
</tr>
<tr>
<td>Schema</td>
<td>Build the workflow. You build the workflow by dragging workflow schema elements from the workflow palette on the left side of the Schema tab. Clicking an element in the schema diagram allows you to define and edit the element’s behavior in the bottom half of the Schema tab.</td>
</tr>
<tr>
<td>Presentation</td>
<td>Defines the layout of the user input dialog box that appears when users run a workflow. You arrange the parameters and attributes into presentation steps and groups to ease identification of parameters in the input parameters dialog box. You define the constraints on the input parameters that users can provide in the presentation.</td>
</tr>
<tr>
<td>Parameters Reference</td>
<td>Shows which workflow elements consume the attributes and parameters in the logical flow of the workflow. This tab also shows the constraints on these parameters and attributes that you define in the Presentation tab.</td>
</tr>
<tr>
<td>Executions</td>
<td>Provides details about each time a particular workflow runs. This information includes the workflow’s status, the user who ran it, the business status of the current element, and the time and date when the workflow started and ended.</td>
</tr>
<tr>
<td>Events</td>
<td>Provides information about each individual event that occurs when the workflow runs. This information includes a description of the event, the user who triggered it, the type and origin of the event, and the time and date when it occurred.</td>
</tr>
<tr>
<td>Permissions</td>
<td>Sets the permissions to interact with the workflow for users or groups of users.</td>
</tr>
</tbody>
</table>

**Provide General Workflow Information**

You provide a workflow name and description, define attributes and certain aspects of workflow behavior, set the version number, check the signature, and set user permissions in the General tab in the workflow workbench.

**Prerequisites**

You must have created a workflow and opened the workflow workbench for that workflow.
Procedure

1. Click the General tab in the workflow workbench.
2. Click the Version digits to set a version number for the workflow.
   The Version Comment dialog box opens.
3. Provide a comment for this version of the workflow and click OK.
   For example, Initial creation if you have just created the workflow.
4. Check the Allowed Operations check boxes to define the operations that users can perform on this workflow.
   You can allow users to perform the following actions on the workflow.
   - View the content of the workflow
   - Add the workflow to a package
   - Edit elements of the workflow
5. Define how the workflow behaves if the Orchestrator server restarts by setting the Server restart behavior value.
   - Leave the default value of Resume workflow execution to make the workflow resume at the point at which its run was interrupted when the server stopped.
   - Click Resume workflow execution and select DON'T resume workflow execution (set as FAILED) to prevent the workflow from restarting if the Orchestrator server restarts.
   You should prevent the workflow from restarting if the workflow depends on the environment in which it runs. For example, if a workflow requires a specific vCenter Server and you reconfigure Orchestrator to connect to a different vCenter Server, restarting the workflow after you restart the Orchestrator server will cause the workflow to fail.
6. Add a detailed description of the workflow in the Description text box.
7. Click Save at the bottom of the workflow workbench.
   A green message at the bottom left of the workflow workbench confirms you have saved your changes.

You defined aspects of the workflow behavior, set the version, and defined the operations that users can perform on the workflow.

What to do next

You must define the workflow attributes and parameters.

Defining Attributes and Parameters

After you create a workflow, you must determine the workflow’s global attributes and input and output parameters.

Workflow attributes are data that workflows process internally. Workflow input parameters are data that comes from an outside source, such as a user or another workflow. Workflow output parameters are data that the workflow delivers when it ends.

- Define Workflow Attributes on page 18
  Workflow attributes are the data that workflows process.
- Define Workflow Parameters on page 18
  Input and output parameters allow you to pass information and data into and out of the workflow.
Define Workflow Attributes

Workflow attributes are the data that workflows process.

**NOTE** You can also define workflow attributes in the workflow schema elements when you create the workflow schema. It is often easier to define an attribute when you create the workflow schema element that processes it.

**Prerequisites**

You must have created a workflow and opened the workflow workbench for that workflow.

**Procedure**

1. Click the *General* tab in the workflow workbench.
   The attributes pane appears in the bottom half of the *General* tab.
2. Right-click in the attributes pane and select *Add Attribute*.
   A new attribute appears in the attributes list, with String as its default type.
3. Click the attribute name to change it.
   The default name is \texttt{att<X>}, where \texttt{X} is a number.
   **NOTE** Workflow attributes must not have the same name as any of the workflow's parameters.
4. Click the attribute type to select a new type from a list of possible values.
   The default attribute type is String.
5. Click the attribute value to set or select a value according to the attribute type.
6. Add a description of the attribute in the *Description* text box.
7. If the attribute is a constant rather than a variable, click the check box to the left of the attribute name to make its value read-only.
   The lock icon (🔒) identifies the column of read-only check boxes.
8. (Optional) If you decide that the attribute should be an input or output parameter rather than an attribute, right-click the attribute and select *Move as INPUT/OUTPUT parameter* to change the attribute into a parameter.

You defined an attribute for the workflow.

**What to do next**

You can define the workflow's input and output parameters.

Define Workflow Parameters

Input and output parameters allow you to pass information and data into and out of the workflow.

You define a workflow's parameters in the workflow workbench. The input parameters are the data upon which the workflow acts that the user provides when they run the workflow. The output parameters are the data the workflow returns when it completes.

**Prerequisites**

You must have created a workflow and opened the workflow workbench for that workflow.
Procedure

1. Click the appropriate tab in the workflow workbench.
   - Click Inputs to create input parameters.
   - Click Outputs to create output parameters.

2. Right-click in the parameters tab and select Add Parameter.
   A new parameter appears in the attributes list, with String as its default type.

3. Click the parameter name to change it.
   The default name is arg_in_<X> for input parameters and arg_out_<X> for output parameters, where <X> is a number.

4. Click the parameter type value to change it from String to a different value from a list of possible values.

5. Add a description of the parameter in the Description text box.

6. (Optional) If you later decide that the parameter should be an attribute rather than a parameter, right-click the parameter and select Move as attribute to change the parameter into an attribute.

You have defined an input or output parameter for the workflow.

What to do next

After you define the workflow's parameters, build the workflow schema.

Workflow Schema

A workflow schema is a graphical representation of a workflow that shows the workflow as a flow diagram of interconnected workflow elements.

- View Workflow Schema on page 20
  You view a workflow schema in the Schema tab for that workflow in the Orchestrator client.

- Building a Workflow in the Workflow Schema on page 20
  Workflow schemas consist of a sequence of schema elements. Workflow schema elements are the building blocks of the workflow, and can represent decisions, scripted tasks, actions, exception handlers, or even other workflows.

- Schema Elements on page 21
  The workflow workbench presents the workflow schema elements in menus in the Schema tab.

- Schema Element Properties on page 24
  Schema elements have properties that you can define and edit in the Schema tab of the workflow palette.

- Links and Bindings on page 26
  Links between elements determine the logical flow of the workflow. Bindings populate elements with data from other elements by binding input and output parameters to workflow attributes.

- Decisions on page 32
  Workflows can implement decision functions that define different courses of action according to a Boolean true or false statement.

- Exception Handling on page 34
  Exception handling catches any errors that occur when a schema element runs. Exception handling defines how the schema element behaves when the error occurs.
View Workflow Schema

You view a workflow schema in the Schema tab for that workflow in the Orchestrator client.

Procedure

1. Click the Workflows view in the Orchestrator client.
2. Navigate to a workflow in the workflow hierarchical list.
3. Click the workflow to show information about that workflow in the right pane.
4. Select the Schema tab in the right pane.

You see the graphical representation of the workflow.

Building a Workflow in the Workflow Schema

Workflow schemas consist of a sequence of schema elements. Workflow schema elements are the building blocks of the workflow, and can represent decisions, scripted tasks, actions, exception handlers, or even other workflows.

You build workflows in the workflow workbench by dragging schema elements from the workflow palette on the left of the workflow workbench into the workflow schema diagram.

Edit a Workflow Schema

You build a workflow by creating a sequence of schema elements, in the workflow workbench’s Schema tab.

Prerequisites

You must have created an empty workflow and opened the workflow for editing in the workflow workbench.

Procedure

1. Click the Schema tab in the workflow workbench.
2. Click the Generic menu on the left of the Schema tab.
3. Drag a schema element from the Generic menu into the workflow schema.
4. Double-click the element you dragged into the workflow schema.

Double-clicking an element allows you to name the element. You must provide elements with unique names in the context of the workflow.

a. Type an appropriate element name inside the schema element.
b. Press Enter.

You cannot rename Waiting timer, Waiting event, End workflow, or Throw exception elements.

5. Right-click an element in the schema and select Copy.
6. Right-click at an appropriate position in the schema and select Paste.

Copying and pasting existing schema elements is a quick way of adding similar elements to the schema. All the settings of the copied element appear in the pasted element. You will need adjust the pasted element settings accordingly.

7. Drag schema elements from the Basic, Log, or Network menus into the workflow schema.

Although elements from the Basic, Log, or Network menus are predefined tasks, you can edit their names. However, you cannot edit their scripting.
8 Drag schema elements from the **Action & Workflow** menu into the workflow schema.

When you drag actions or workflows into the workflow schema, a dialog box appears that allows you to search for the action or workflow to insert.

9 Enter the name or part of the name of the workflow or action to insert into the workflow in the **Search** text box.

The workflows or actions that match the search appear in the dialog box.

10 Double-click a workflow or action to select it.

You inserted the workflow or action into the workflow schema.

11 Repeat the preceding steps until you have added all the required schema elements to the workflow schema.

A workflow schema must have at least one **End workflow** element, but it can have several.

**What to do next**

After you add all the necessary elements to the workflow schema, define the properties of those elements and link and bind them all together.

**Modify Search Results**

You use the **Search** text box to find elements such as workflows or actions. If a search returns a partial result, you can modify the number of results that the search returns.

When you use the search for an element, a green message box indicates that the search lists all the results. A yellow message box indicates that the search lists only partial results.

**Procedure**

1 (Optional) If you are editing a workflow in the workflow workbench, click **Save and Close** to exit the workbench.

2 Click **Tools > User Preferences** in the Orchestrator client menu.

3 Click the **General** tab.

4 Enter the number of results for searches to return in the **Finder Maximum Size** text box.

5 Click **Save and Close** in the User Preferences dialog box.

You modified the number of results that searches return.

**Schema Elements**

The workflow workbench presents the workflow schema elements in menus in the **Schema** tab.

**Table 2-2** describes all of the schema elements from which you can build workflows.

<table>
<thead>
<tr>
<th>Schema Element Name</th>
<th>Description</th>
<th>Icon Location in Workflow Workbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Workflow</td>
<td>The starting point of the workflow. All workflows contain this element and it cannot be removed from the workflow schema. A workflow can have only one start element. Start elements have one output and no input.</td>
<td>Always present in the <strong>Schema</strong> tab</td>
</tr>
<tr>
<td>Scriptable Task</td>
<td>General purpose tasks you define. You write JavaScript functions in this element.</td>
<td><strong>Generic</strong> workflow palette</td>
</tr>
</tbody>
</table>

**Table 2-2. Schema Elements and Icons**
<table>
<thead>
<tr>
<th>Schema Element Name</th>
<th>Description</th>
<th>Icon</th>
<th>Icon Location in Workflow Workbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision</td>
<td>Boolean function. Decision elements take one input parameter and return either true or false. The type of decision the element makes depends on the type of the input parameter. Decision elements allow the workflow to branch into different directions, depending on the input parameter the decision element receives. If the received input parameter corresponds to an expected value, the workflow continues along a certain route. If the input is not the expected value, the workflow continues on an alternative path.</td>
<td><img src="image" alt="Icon" /></td>
<td>Generic workflow palette</td>
</tr>
<tr>
<td>Custom Decision</td>
<td>Boolean function. Custom decisions can take several input parameters and act upon them according to custom scripts. Returns either true or false.</td>
<td><img src="image" alt="Icon" /></td>
<td>Generic workflow palette</td>
</tr>
<tr>
<td>User Interaction</td>
<td>Allows users to pass new input parameters into the workflow. You can design how the user interaction element presents the request for input parameters and place constraints on the parameters that users can provide. You can set permissions to determine which users can provide the input parameters. When a running workflow arrives at a user interaction element, it enters a passive state and prompts the user for input. You can set a timeout period within which the users can answer. The workflow resumes according to the data the user passes to it, or returns an exception if the timeout period expires. While it is waiting for the user to respond, the workflow token is in the waiting state.</td>
<td><img src="image" alt="Icon" /></td>
<td>Generic workflow palette</td>
</tr>
<tr>
<td>Waiting Timer</td>
<td>Used by long-running workflows. When a running workflow arrives at a Waiting Timer element it enters a passive state. You set an absolute date at which the workflow resumes running. While it is waiting for the date, the workflow token is in the waiting-signal state.</td>
<td><img src="image" alt="Icon" /></td>
<td>Generic workflow palette</td>
</tr>
<tr>
<td>Waiting Event</td>
<td>Used in long-running workflows. When a running workflow arrives at a Waiting Event element it enters a passive state. You define a trigger event that the workflow awaits before it resumes running. While it is waiting for the event, the workflow token is in the waiting-signal state.</td>
<td><img src="image" alt="Icon" /></td>
<td>Generic workflow palette</td>
</tr>
<tr>
<td>End Workflow</td>
<td>The end point of the workflow. You can have multiple end elements in a schema, to represent the different possible outcomes of the workflow. End elements have one input with no output. When a workflow reaches an End Workflow element, the workflow token enters the completed state.</td>
<td><img src="image" alt="Icon" /></td>
<td>Generic workflow palette</td>
</tr>
<tr>
<td>Schema Element Name</td>
<td>Description</td>
<td>Icon</td>
<td>Icon Location in Workflow Workbench</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>Thrown Exception</td>
<td>Creates an exception and stops the workflow. Multiple occurrences of this element can be present in the workflow schema. Exception elements have one input parameter, which can only be of the String type, and have no output parameter. When a workflow reaches an Exception element, the workflow token enters the failed state.</td>
<td>![Exception Icon]</td>
<td>Generic workflow palette</td>
</tr>
<tr>
<td>Workflow Note</td>
<td>Allows you to annotate sections of the workflow. You can stretch notes to delineate sections of the workflow. You can change the background color of the notes to differentiate between different workflow zones. Workflow notes provide visual information only, to help you understand the schema.</td>
<td>![Note Icon]</td>
<td>Generic workflow palette</td>
</tr>
<tr>
<td>Pre-Defined Task</td>
<td>Noneditable scripted elements that perform standard tasks that workflows commonly use. The following tasks are predefined:</td>
<td>![Task Icon]</td>
<td>Basic, Log, and Network workflow palette</td>
</tr>
<tr>
<td></td>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Sleep</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Change credential</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Wait until date</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Wait for custom event</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Increase counter</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Decrease counter</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Add hours to date</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Log</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- System log</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- System warning</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- System error</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Server log</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Server warning</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Server error</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- System+server log</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- System+server warning</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- System+server error</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Network</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- HTTP post</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- HTTP get</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Send custom event</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>Calls on an action from the Orchestrator libraries of actions. When a workflow reaches an action element, it calls and runs that action.</td>
<td>![Action Icon]</td>
<td>Action &amp; Workflow workflow palette</td>
</tr>
<tr>
<td>Workflow</td>
<td>Starts another workflow synchronously. As soon as a workflow reaches a workflow element in its schema, it runs that workflow as part of its own process. The original workflow does not continue until the called workflow completes its run.</td>
<td>![Workflow Icon]</td>
<td>Action &amp; Workflow workflow palette</td>
</tr>
</tbody>
</table>
Table 2-2. Schema Elements and Icons (Continued)

<table>
<thead>
<tr>
<th>Schema Element Name</th>
<th>Description</th>
<th>Icon</th>
<th>Icon Location in Workflow Workbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous Workflows</td>
<td>Starts a workflow asynchronously. When a workflow reaches an asynchronous workflow element, it starts that workflow and continues its own run. The original workflow does not wait for the called workflow to finish before continuing.</td>
<td><img src="image" alt="Icon" /></td>
<td>Action &amp; Workflow workflow palette</td>
</tr>
<tr>
<td>Schedule Workflow</td>
<td>Creates a task to run the workflow at a set time, then the workflow continues its run.</td>
<td><img src="image" alt="Icon" /></td>
<td>Action &amp; Workflow workflow palette</td>
</tr>
<tr>
<td>Nested Workflows</td>
<td>Starts several workflows simultaneously. You can choose to nest local workflows and remote workflows that are in a different Orchestrator server. You can also run workflows with different credentials. The workflow waits until all the nested workflows complete before it continues its run.</td>
<td><img src="image" alt="Icon" /></td>
<td>Action &amp; Workflow workflow palette</td>
</tr>
</tbody>
</table>

Schema Element Properties

Schema elements have properties that you can define and edit in the Schema tab of the workflow palette.

**Edit a Schema Element’s Global Properties**

You define a schema element’s global properties in the schema element’s Info tab.

**Prerequisites**

The Schema tab of the workflow workbench must contain elements.

**Procedure**

1. Click the Schema tab in the workflow workbench.
2. Select an element to edit by clicking a schema element in the workflow schema.
   
   The schema element’s properties tabs appear at the bottom of the workflow workbench.
3. Click the Info tab.
4. Provide a name for the schema element in the Name text box.
   
   This is the name that appears in the schema element in the workflow schema diagram.
5. Click the Interaction text box and select a description from the list.
   
   The Interaction property allows you to select between standard descriptions of how this element interacts with objects outside of the workflow. This property is for information only.
6. (Optional) Click Color to change the background color of the schema element.
   
   You can highlight certain sections of the schema by changing the color of individual workflow elements.
7. Provide a business status description in the Business Status text box.
   
   The Business Status property is a brief description of what this element does. When a workflow is running, the workflow token shows the Business Status of each element as it runs. This feature is useful for tracking workflow status.
### Schema Element Properties Tabs

You access a schema element’s properties by clicking on a schema element that you have dragged into the workflow schema. The element properties appear in tabs at the bottom of the workflow workbench.

Different element types present different properties tabs, as shown in Table 2-3.

**Table 2-3. Properties Tabs per Schema Element**

<table>
<thead>
<tr>
<th>Schema Element Property Tab</th>
<th>Description</th>
<th>Applies to Schema Element Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attributes</strong></td>
<td>Attributes that elements require from an external source, such as the user, an event, or a timer. The attributes can be a timeout limit, a time and date, a trigger, or user credentials.</td>
<td>User Interaction, Waiting Event, Waiting Timer</td>
</tr>
<tr>
<td><strong>Decision</strong></td>
<td>Defines the decision statement. The input parameter that the decision element receives either matches or does not match the decision statement, resulting in two possible courses of action.</td>
<td>Decision</td>
</tr>
<tr>
<td><strong>End Workflow</strong></td>
<td>Stops the workflow, either because the workflow completed successfully, or because it encountered an error and returned an exception.</td>
<td>End, Exception</td>
</tr>
<tr>
<td><strong>Exception</strong></td>
<td>How this schema element behaves in the event of an exception.</td>
<td>Action, Asynchronous Workflow, Exception, Nested Workflows, Predefined Task, Schedule Workflow, Scriptable Task, User Interaction, Waiting Event, Waiting Timer, Workflow</td>
</tr>
<tr>
<td><strong>External Inputs</strong></td>
<td>Input parameters that the user must provide at a certain moment while the workflow runs.</td>
<td>User Interaction</td>
</tr>
<tr>
<td><strong>IN</strong></td>
<td>The IN binding for this element. The IN binding defines the way in which the schema element receives input from the element that precedes it in the workflow.</td>
<td>Action, Asynchronous Workflow, Custom Decision, Predefined Task, Schedule Workflow, Scriptable Task, Workflow</td>
</tr>
<tr>
<td>Schema Element Property Tab</td>
<td>Description</td>
<td>Applies to Schema Element Type</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
<td>--------------------------------</td>
</tr>
</tbody>
</table>
| **Info**                    | The schema element's general properties and description. The information the Info tab displays depends on the type of schema element. | ■ Action  
■ Asynchronous Workflow  
■ Custom Decision  
■ Decision  
■ Nested Workflows  
■ Note  
■ Predefined Task  
■ Schedule Workflow  
■ Scriptable Task  
■ User Interaction  
■ Waiting Event  
■ Waiting Timer  
■ Workflow |
| **OUT**                     | The OUT binding for this element. The OUT binding defines the way in which the schema element binds output parameters to the workflow attributes or to the workflow output parameters. | ■ Action  
■ Asynchronous Workflow  
■ Predefined Task  
■ Schedule Workflow  
■ Scriptable Task  
■ Workflow |
| **Presentation**            | Defines the layout of the input parameters dialog box the user sees if the workflow needs user input while it is running. | User Interaction |
| **Scripting**               | Shows the JavaScript function that defines the behavior of this schema element. For Asynchronous Workflow, Schedule Workflow, and Action elements this scripting is read-only. For scriptable task and custom decision elements, you edit the JavaScript in this tab. | ■ Action  
■ Asynchronous Workflow  
■ Custom Decision  
■ Predefined Task  
■ Schedule Workflow  
■ Scriptable Task |
| **Visual Binding**          | Shows a graphical representation of how the parameters and attributes of this schema element bind to the parameters and attributes of the elements that come before and after it in the workflow. This is another representation of the element's IN and OUT bindings. | ■ Action  
■ Asynchronous Workflow  
■ Predefined Task  
■ Schedule Workflow  
■ Scriptable Task  
■ Workflow |
| **Workflows**               | Selects the workflows to nest. | Nested Workflows |

**Links and Bindings**

Links between elements determine the logical flow of the workflow. Bindings populate elements with data from other elements by binding input and output parameters to workflow attributes.

To understand links and bindings, you must understand the difference between the logical flow of a workflow and the data flow of a workflow.
**Logical Flow of a Workflow**

The logical flow of a workflow is the progression of the workflow from one element to the next in the schema as the workflow runs. You define the logical flow of the workflow by linking elements in the schema.

The standard path is the path that the workflow takes through the logical flow if all elements run normally. The exception path is the path that the workflow takes through the logical flow if an element does not run normally.

Different styles of arrows in the workflow schema denote the different paths that the workflow can take through its logical flow.

- A black arrow denotes the standard path that the workflow takes from one element to the next.
- A green arrow denotes the path that the workflow takes if a Boolean decision element returns `true`.
- A red dotted arrow denotes the path that the workflow takes if a Boolean decision element returns `false`.
- A thick red dotted arrow denotes the exception path that the workflow takes if a workflow element does not run correctly.

Figure 2-1 shows an example workflow schema that demonstrates the different paths that workflows can take.

**Figure 2-1. Different Workflow Paths Through the Logical Flow of the Workflow**

This example workflow can take the following paths through its logical flow.

- **Standard path, true decision result, no exceptions.**
  - a  The decision element returns `true`.
  - b  The `SnapVMsInResourcePool` workflow runs successfully.
  - c  The `sendHtmlEmail` action runs successfully.
  - d  The workflow ends successfully in the `completed` state.

- **Standard path, false decision result, no exceptions.**
  - a  The decision element returns `false`.
  - b  The operation the scriptable task element defines runs successfully.
  - c  The `sendHtmlEmail` action runs successfully.
  - d  The workflow ends successfully in the `completed` state.
true decision result, exception.
   a The decision element returns true.
   b The SnapVMsInResourcePool workflow encounters an error.
   c The workflow returns an exception and stops in the failed state.
false decision result, exception.
   a The decision element returns false.
   b The operation the Scriptable task element defines encounters an error.
   c The workflow returns an exception and stops in the failed state.

Element Links

Links connect schema elements and define the logical flow of the workflow from one element to the next. Elements can usually set only one outgoing link to another element in the workflow and one exception link to an element that defines its exception behavior. The outgoing link defines the standard path of the workflow. The exception link defines the exception path of the workflow. In most cases, a single schema element can receive incoming standard path links from multiple elements.

The following elements are exceptions to the preceding statements.

- The Start Workflow element cannot receive incoming links and has no exception link.
- Exception elements can receive multiple incoming exception links, and have no outgoing or exception links.
- Decision elements have two outgoing links that define the paths the workflow takes depending on the decision's true or false result. Decisions have no exception link.
- End Workflow elements cannot have outgoing links or exception links.

Create Standard Path Links

You link elements by connecting them using the connector tool in the Schema tab of the workflow workbench. When you link one element to another, you always link the elements in the order in which they run in the workflow. You always start from the element that runs first to create a link between two elements.

Prerequisites

To link elements, you must have the workflow workbench open and the Schema must contain elements.

Procedure

1 Click the connector tool button in the toolbar at the top of the Schema tab to activate the connector tool.
2 Click an element to link to another element.
3 Move the pointer over the highlighted element to link to another element.
   A black rectangle appears at the bottom of the element.
4 Left-click inside the element near the black rectangle, hold down the left mouse button, and move the pointer to the target element.
   An arrow appears between the two elements and the target element turns green.
5 Release the left mouse button.
   The arrow remains between the two elements.

A standard path now links the elements.
What to do next

The elements are joined, but you have not defined the data flow. You must define the IN and OUT bindings to bind incoming and outgoing data to workflow attributes.

Data Flow of a Workflow

The data flow of a workflow is the manner in which workflow element input and output parameters bind to workflow attributes as each element of the workflow runs. You define the data flow of a workflow by using schema element bindings.

When an element in the workflow schema runs, it requires data in the form of input parameters. It takes the data for its input parameters by binding to a workflow attribute that you set when you create the workflow, or by binding to an attribute that a preceding element in the workflow set when it ran.

The element processes the data, possibly transforms it, and generates the results of its run in the form of output parameters. The element binds its resulting output parameters to new workflow attributes that it creates. Other elements in the schema can bind to these new workflow attributes as their input parameters. The workflow can generate the attributes as its output parameters at the end of its run.

Figure 2-2 shows a very simple workflow. The black arrows represent the element linking and the logical flow of the workflow. The red lines show the data flow of the workflow.

Figure 2-2. Example of Workflow Data Flow

The data flows through the workflow as follows.
1. The workflow starts with input parameters a and b.
2. The first element processes parameter a and binds the result of the processing to workflow attribute c.
3. The first element processes parameter b and binds the result of the processing to workflow attribute d.
4. The second element takes workflow attribute c as an input parameter, processes it, and binds the resulting output parameter to workflow attribute e.
5. The second element takes workflow attribute d as an input parameter, processes it, and generates output parameter f.
6. The workflow ends and generates workflow attribute f as its output parameter, the result of its run.
Element Bindings

You must bind all workflow element input and output parameters to workflow attributes. Bindings set data in the elements, and define the output and exception behavior of the elements. Links define the logical flow of the workflow, whereas bindings define the data flow.

To set data in an element, generate output parameters from the element after processing, and handle any errors that might occur when the element runs, you must set the element binding.

**IN bindings**
Set a schema element’s incoming data. You bind the element's local input parameters to source workflow attributes. The IN tab lists the element's input parameters in the Local Parameter column. The IN tab lists the workflow attributes to which the local parameter binds in the Source Parameter column. The tab also shows the parameter type and a description of the parameter.

**OUT bindings**
Change workflow attributes and generate output parameters when an element finishes its run. The OUT tab lists the element's output parameters in the Local Parameter column. The OUT tab lists the workflow attributes to which the local parameter binds in the Source Parameter column. The tab also shows the parameter type and a description of the parameter.

**Exception bindings**
Link to exception handlers if the element encounters an exception when it runs.

You must use IN bindings to bind every attribute or input parameter you use in a schema element to a workflow attribute. If the element changes the values of the input parameters it receives when it runs, you must bind them to a workflow attribute by using an OUT binding. Binding the element's output parameters to workflow elements allows other elements that follow it in the workflow schema to take those output parameters as their input parameters.

A common mistake when creating workflows is to forget to bind output parameter values to reflect the changes the element makes to the workflow attributes.

**IMPORTANT** When you add an element that requires input and output parameters of a type that you already defined in the workflow, Orchestrator sets the bindings to these parameters. You must check that the parameters Orchestrator binds are correct, in case the workflow defines different parameters of the same type to which the element could bind.

Define Element Bindings

After you link elements to create the logical flow of the workflow, you define element bindings to define how each element processes the data it receives and generates.

**Prerequisites**
You must have a workflow schema in the Schema tab of the workflow workbench, and have created links between the elements.
Procedure

1. Click an element on which to set the bindings.
   The element is highlighted and the element attributes tabs appear at the bottom of the Schema tab.

2. Click the IN tab.
   The contents of the IN tab depend on the type of element you selected.
   - If you selected a predefined task, workflow, or action element, the IN tab lists the possible local input parameters for that type of element, but the binding is not set.
   - If you selected another type of element, you can select from a list of input parameters and attributes you already defined for the workflow by right-clicking in the IN tab and selecting Bind to workflow attribute/parameter.
   - If the required attribute does not exist yet, you can create it by right-clicking in the IN tab and selecting Bind to workflow attribute/parameter > Create attribute/parameter in workflow.

3. If an appropriate parameter exists, choose an input parameter to bind, and click the Not set link in the Source Parameter text box.
   A list of possible source parameters and attributes to bind to appears.

4. Choose a source parameter to bind to the local input parameter from the list proposed.

5. (Optional) If you have not defined the source parameter to which to bind, you can create it by clicking the Create attribute/parameter in workflow link in the parameter selection dialog box.

6. Click the OUT tab.
   The contents of the OUT tab depend on the type of element you selected.
   - If you selected a predefined task, workflow, or action element, the OUT tab lists the possible local output parameters for that type of element, but the binding is not set.
   - If you selected another type of element, you can select from a list of output parameters and attributes you defined for the workflow by right-clicking in the OUT tab and selecting Bind to workflow attribute/parameter.
   - If the required attribute does not exist, you can create it by right-clicking in the IN tab and selecting Bind to workflow attribute/parameter > Create attribute/parameter in workflow.

7. Choose a parameter to bind.

8. Click the Source Parameter > Not set link.

9. Choose a source parameter to bind to the input parameter.

10. (Optional) If you did not define the parameter to which to bind, you can create it by clicking the Create attribute/parameter in workflow link in the parameter selection dialog box.

You defined the input parameters that the element receives and the output parameters that it generates, and bound them to workflow attributes and parameters.

What to do next

You can create forks in the path of the workflow by defining decisions.
Decisions

Workflows can implement decision functions that define different courses of action according to a Boolean true or false statement.

Decisions are forks in the workflow. Workflow decisions are made according to inputs provided by you, by other workflows, by applications, or by the environment in which the workflow is running. The value of the input parameter that the decision element receives determines which branch of the fork the workflow takes. For example, a workflow decision might receive the power status of a given virtual machine as its input. If the virtual machine is powered on, the workflow takes a certain path through its logical flow. If the virtual machine is powered off, the workflow takes a different path.

Decisions are always Boolean functions. The only possible outcomes for each decision are true or false.

Custom Decisions

Custom decisions differ from standard decisions in that you define the decision statement in a script. Custom decisions return true or false according to the statement you define, as the following example shows.

```java
if (decision_statement){
    return true;
}else{
    return false;
}
```

Create Decision Element Links

Decision elements differ from other elements in that they have only true or false output parameters. Decision elements have no exception linking.

Prerequisites

You must have the workflow workbench open and the Schema tab must contain elements, including at least one decision element.

Procedure

1. Click a decision element to link to two other elements to define two possible branches in the workflow.
2. Click the connector tool button in the toolbar at the top of the Schema tab.
3. Move the pointer over the highlighted decision element to link to two other elements.
   - If you hold the pointer over the left side of the decision element, a green arrow appears at the bottom of the element. The green arrow represents the true path the workflow takes if the input parameter or attribute received by the decision element matches the decision statement.
   - If you hold the pointer over the right side of the decision element, a red arrow appears at the bottom of the element. The red arrow represents the false path the workflow takes if the input parameter or attribute received by the decision element does not match the decision statement.
4. Left-click inside the left side of the decision element, hold down the left mouse button, and move the pointer to the target element.
   A green arrow appears between the two elements and the target element turns green.
5. Release the left mouse button.
   The green arrow remains between the two elements. You have defined the path the workflow takes when the decision element receives the expected value.
6 Left-click inside the right side of the decision element, hold down the left mouse button, and move the pointer to the target element.

A dotted red arrow appears between the two elements and the target element turns green.

7 Release the left mouse button.

The dotted red arrow remains between the two elements. You have defined the path the workflow takes when the decision element receives unexpected input.

You have defined two possible true or false paths for the workflow to take, depending on the input parameter or attribute the decision element receives.

**What to do next**

The decision element is linked to two other elements, but you did not define how the workflow determines which path to take. You must define the decision statement.

---

**Create Workflow Branches Using Decisions**

Decision elements are simple Boolean functions that you use to create branches in workflows. Decision elements determine whether or not the input received matches the decision statement you set. As a function of this decision, the workflow continues its course along one of two possible paths.

**Prerequisites**

You must have a decision element linked to two other elements in the schema in the workflow workbench before you define the decision.

**Procedure**

1 Click the decision element.

2 Click the **Decision** tab in the element properties tabs at the bottom of the **Schema** tab.

3 Click the **Not Set (NULL)** link to select the possible source input parameter for this decision.

   A dialog box appears, which lists all the attributes and input parameters you defined in this workflow.

4 Select an input parameter from the list by double-clicking it.

5 (Optional) If you did not define the source parameter to which to bind, you can create it by clicking the **Create attribute/parameter in workflow** link in the parameter selection dialog box.

6 Select a decision statement from the drop-down menu.

   The statements the menu proposes are contextual, and differ according to the type of input parameter selected.

7 Add a value for the statement to match.

   Depending on the input type and the statement you select, you might see a **Not Set (NULL)** link in the value text box. Clicking this link gives you a predefined choice of values. Otherwise, for example for Strings, this is a text box in which you provide a value.

You defined a statement for the decision element. When the decision element receives the input parameter, it compares the value of the input parameter to the value in the statement and determines whether the statement is true or false.

**What to do next**

You must set up how the workflow handles exceptions.
Exception Handling

Exception handling catches any errors that occur when a schema element runs. Exception handling defines how the schema element behaves when the error occurs.

All elements in a workflow, except for decisions and start and end elements, contain a specific output parameter type that serves only for handling exceptions. If an element encounters an error during its run, it can send an error signal to an exception handler. Exception handlers catch the error and react according to the errors they receive. If the exception handlers you define cannot handle a certain error, you can bind an element’s exception output parameter to an Exception element, which ends the workflow run in the failed state.

Exceptions act as a try and catch sequence within a workflow element. If you do not need to handle a given exception in an element, you do not have to bind that element’s exception output parameter.

The output parameter type for exceptions is always an errorCode object.

Create Exception Bindings

Elements can set bindings that define how the workflow behaves if it encounters an error in that element.

Prerequisites

The Schema tab of the workflow workbench must contain elements.

Procedure

1. Click the element on which to set the exception binding.
2. Click the connector tool button in the toolbar at the top of the Schema tab or press Ctrl.
3. Move the pointer over the right of the element for which to set the exception binding.
   
   A red rectangle appears on the right of the element.
4. Left-click inside the element near the red rectangle, hold down the left mouse button, and move the pointer to the target element.
   
   A thick dotted red arrow links the two elements. The target element defines the behavior of the workflow if the element that links to it encounters an error.
5. Click the element that links to the exception handling element.
6. Click the Exceptions tab in the schema element properties tabs at the bottom of the Schema tab.
7. Click the Not set link to the Output Exception Binding value.
   
   Select a parameter to bind to the exception output parameter from the exception attribute binding dialog box.
   
   Click Create parameter/attribute in workflow to create an exception output parameter.
8. Click the target element that defines the exception handling behavior.
9. Click the IN tab in the schema element properties tabs at the bottom of the Schema tab.
10. Right-click in the IN tab and select Bind to workflow parameter/attribute.
11. Select the exception output parameter and click Select.
12 Click the **OUT** tab for the exception handling element in the schema element properties tabs at the bottom of the **Schema** tab.

13 Define the behavior of the exception handling element.

- Right-click in the **OUT** tab and select **Bind to workflow parameter/attribute** to select an output parameter for the exception handling element to generate.
- Click the **Scripting** tab and use JavaScript to define the behavior of the exception handling element.

You defined how the element handles exceptions.

**What to do next**

You must define how to obtain input parameters from users when they run the workflow.

**Obtaining Input Parameters from Users When a Workflow Starts**

If a workflow requires input parameters, it opens a dialog box in which users enter the required input parameter values when it runs. You can organize the content and layout, or presentation, of this dialog box in **Presentation** tab in the workflow workbench.

The way you organize parameters in the **Presentation** tab translates into the input parameters dialog box when the workflow runs, and in the dialog box that opens when you run a workflow from a Web view.

The **Presentation** tab also allows you to add descriptions of the input parameters to help users when they provide input parameters. You can also set properties and constraints on parameters in the **Presentation** tab to restrict the parameters that users provide. If the parameters the user provides do not meet the constraints you set in the **Presentation** tab, the workflow will not run.

**Creating the Input Parameters Dialog Box In the Presentation Tab**

You define the layout of the dialog box in which users provide input parameters when they run a workflow in the **Presentation** tab of the workflow workbench.

The **Presentation** tab allows you to group input parameters into categories and to define the order in which these categories appear in the input parameters dialog box.

**Presentation Descriptions**

You can add an associated description for each parameter or group of parameters, which appears in the input parameters dialog box. The descriptions provide information to the users to help them provide the correct input parameters. You can enhance the layout of the description text by using HTML formatting.

**Defining Presentation Input Steps**

By default, the input parameters dialog box lists all the required input parameters in a single list, ordered alphabetically. To help users enter input parameters, you can define nodes, called input steps, in the presentation node. Input steps group input parameters of a similar nature. The input parameters under an input step appear in a distinct section in the input parameters dialog box when the workflow runs.

**Defining Presentation Display Groups**

Each input step can have nodes of its own called display groups. The display groups define the order in which parameter input text boxes appear within their section of the input parameters dialog box. You can define display groups independently of input steps.
Create the Presentation of the Input Parameters Dialog Box

You create the presentation of the dialog box in which users provide input parameters when they run a workflow in the Presentation tab in the workflow workbench.

Prerequisites

You must have created a workflow and a defined list of input parameters.

Procedure

1. In the workflow workbench, click the Presentation tab.
   By default, all of the workflow’s parameters appear under the main Presentation node in the order in which you create them.
2. Right-click the Presentation node and select New Step.
   A New Step node appears under the Presentation node.
3. Double-click the New Step node to provide it with an appropriate name and press Enter.
   This name appears as a section header in the input parameters dialog box when the workflow runs.
4. Click the input step and add a description in the General tab in the bottom half of the Presentation tab.
   This description appears in the input parameters dialog box to provide information to the users to help them provide the correct input parameters. You can enhance the layout of the description text by using HTML formatting.
5. Right-click the input step you created and select Create Display group.
   A New Group node appears under the input step node.
6. Double-click the New Group node and provide it with an appropriate name.
   This name appears as a subsection header in the input parameters dialog box when the workflow runs.
7. Click the display group and add a description in the General tab in the bottom half of the Presentation tab.
   This description appears in the input parameters dialog box. You can enhance the layout of the description text by using HTML formatting. You can add a parameter value to a group description by using an OGNL statement, such as ${#param}.
8. Repeat the preceding steps until you have created all the input steps and display groups to appear in the input parameters dialog box when the workflow runs.
9. Drag parameters from under the Presentation node to the steps and groups of your choice.

You created the layout of the input parameters dialog box through which users provide input parameter values when the workflow runs.

What to do next

You must set the parameter properties.
Setting Parameter Properties

Orchestrator allows you to define properties to qualify the input parameter values that users provide when they run workflows. The parameter properties you define impose limits on the types and values of the input parameters the users provide.

Every parameter can have several properties. You define an input parameter’s properties in the Properties tab for a given parameter in the Presentation tab.

Parameter properties validate the input parameters and modify the way that text boxes appear in the input parameters dialog box. Some parameter properties can create dependencies between parameters.

Static and Dynamic Parameter Property Values

A parameter property value can be either static or dynamic. Static property values remain constant. If you set a property value to static, you set or select the property’s value from a list that the workflow workbench generates according to the parameter type.

Dynamic property values depend on the value of another parameter or attribute. You define the functions by which dynamic properties obtain values by using an object graph navigation language (OGNL) expression. If a dynamic parameter property value depends on the value of another parameter property value and the other parameter property value changes, the OGNL expression recalculates and changes the dynamic property value.

Set Parameter Properties

When a workflow starts, it validates input parameter values from users against any parameter properties that you set.

Prerequisites

You must have a workflow and defined a list of input parameters before setting the parameter constraints.

Procedure

1. In the workflow workbench, click the Presentation tab.
2. Click a parameter in the Presentation tab. The parameter’s General and Properties tabs appear at the bottom of the Presentation tab.
3. Click the parameter’s Properties tab.
4. Right-click in the Properties tab and select Add property. A dialog box opens, presenting a list of the possible properties for a parameter of the type selected.
5. Select a property from the list presented in the dialog box and click OK. The property appears in the Properties tab.
6. Under Value, make the property value either static or dynamic by selecting the corresponding symbol from the drop-down menu.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Static property" /></td>
<td>Static property</td>
</tr>
<tr>
<td><img src="image" alt="Dynamic property" /></td>
<td>Dynamic property</td>
</tr>
</tbody>
</table>
7 If you set the property value to static, you select a property value according to the type of parameter for which you are setting the properties.

8 If you set the property value to dynamic, you define the function to obtain the parameter property value by using an OGNL expression.

The workflow workbench provides help writing the OGNL expression.

a Click the ☐ icon to obtain a list of all the attributes and parameters defined by the workflow that this expression can call upon.

b Click the ☐ icon to obtain a list of all the actions in the Orchestrator API that return an output parameter of the type for which you are defining the properties.

Clicking items in the proposed lists of parameters and actions adds them to the OGNL expression.

9 Click Save at the bottom of the workflow workbench.

You defined the properties of the workflow's input parameters.

What to do next

Validate and debug the workflow.

Workflow Input Parameter Properties

You can constrain the input parameters that users provide when they run workflows by setting parameter properties.

The possible properties for each type of parameter are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter Property</th>
<th>Parameter Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum string length</td>
<td>String</td>
<td>Sets a maximum length for the parameter.</td>
</tr>
<tr>
<td>Minimum string length</td>
<td>String</td>
<td>Sets a minimum length for the parameter.</td>
</tr>
<tr>
<td>Matching regular expression</td>
<td>String</td>
<td>Validates the input using a regular expression.</td>
</tr>
<tr>
<td>Maximum number value</td>
<td>Number</td>
<td>Sets a maximum value for the parameter.</td>
</tr>
<tr>
<td>Minimum number value</td>
<td>Number</td>
<td>Sets a minimum value for the parameter.</td>
</tr>
<tr>
<td>Number format</td>
<td>Number</td>
<td>Formats the input for the parameter.</td>
</tr>
<tr>
<td>Enumeration</td>
<td>Any</td>
<td>Specifies an ordered list of possible values.</td>
</tr>
<tr>
<td>Mandatory</td>
<td>Any</td>
<td>Makes the parameter mandatory.</td>
</tr>
<tr>
<td>Choice from another parameter or attribute</td>
<td>Any</td>
<td>Derives possible user inputs from another parameter. For example, if this parameter is an SSH:File and a parameter in a previous step is an SSH:Folder, you can set this property to restrict the possible input parameter values to files contained in the SSH:Folder.</td>
</tr>
<tr>
<td>Predefined list of elements</td>
<td>Any</td>
<td>Similar to Choice from another parameter or attribute, but the user can add a different value to the one derived from the preceding parameter.</td>
</tr>
</tbody>
</table>
Table 2-4. Workflow Input Parameter Properties (Continued)

<table>
<thead>
<tr>
<th>Parameter Property</th>
<th>Parameter Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show parameter input</td>
<td>Any</td>
<td>Shows or hides a parameter text box in the presentation dialog box, depending on the value of a preceding Boolean parameter.</td>
</tr>
<tr>
<td>Hide parameter input</td>
<td>Any</td>
<td>Similar to Show parameter input but takes the negative value of a previous Boolean parameter.</td>
</tr>
<tr>
<td>Matching expression</td>
<td>Any parameter type obtained from a plug-in</td>
<td>The input parameter matches a given expression.</td>
</tr>
<tr>
<td>Show in inventory</td>
<td>Any parameter type obtained from a plug-in</td>
<td>If set, you can run the present workflow on any object of this type by right-clicking it in the inventory view and selecting Execute operation.</td>
</tr>
<tr>
<td>Specify root object in selector</td>
<td>Any parameter type obtained from a plug-in</td>
<td>Specifies the root object if the selector for this parameter is a hierarchical list selector.</td>
</tr>
<tr>
<td>Select as</td>
<td>Any parameter type obtained from a plug-in</td>
<td>Use a list or hierarchical list selector to select the parameter.</td>
</tr>
<tr>
<td>Default value</td>
<td>Any</td>
<td>Default value for this parameter.</td>
</tr>
<tr>
<td>Custom validation</td>
<td>OGNL scriptable validation</td>
<td>If the invocation of the OGNL expression returns a String, the validation shows this String as the text of the error result.</td>
</tr>
<tr>
<td>Auto start</td>
<td>Boolean</td>
<td>Starts the workflow automatically.</td>
</tr>
<tr>
<td>Mandatory input</td>
<td>Boolean</td>
<td>Makes this parameter mandatory. The workflow will not run without it.</td>
</tr>
</tbody>
</table>

Predefined Constant Values for OGNL Expressions

You can use predefined constants when you create OGNL expressions to obtain dynamic parameter property values.

Orchestrator defines the following constants for use in OGNL expressions.

Table 2-5. Predefined OGNL Constant Values

<table>
<thead>
<tr>
<th>Constant Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>${__current}</td>
<td>Current value of the custom validation property or matching expression property</td>
</tr>
<tr>
<td>${__username}</td>
<td>Username of the user who started the workflow</td>
</tr>
<tr>
<td>${__userDisplayName}</td>
<td>Display name of the user who started the workflow</td>
</tr>
<tr>
<td>${__serverUrl}</td>
<td>URL containing the IP address of the server from which the user starts the workflow</td>
</tr>
<tr>
<td>${__datetime}</td>
<td>Current date and time</td>
</tr>
<tr>
<td>${__date}</td>
<td>Current date, with time set to 00:00:00</td>
</tr>
<tr>
<td>${__timezone}</td>
<td>Current timezone</td>
</tr>
</tbody>
</table>
Requesting User Interactions While a Workflow Runs

A workflow can sometimes require additional input parameters from an outside source while it runs. These input parameters can come from another application or workflow, or the user can provide them directly.

For example, if a certain event occurs while a workflow runs, the workflow can request human interaction to decide what course of action to take. The workflow waits before continuing, either until the user responds to the request for information, or until the waiting time exceeds a possible timeout period. If the waiting time exceeds the timeout period, the workflow returns an exception.

User interactions have two default attributes, security.group and timeout.date. By setting the security.group attribute to a given LDAP user group, you restrict the permission to respond to the user interaction request to members of that user group. These attributes are mandatory.

By setting the timeout.date attribute, you set an absolute time and date until which the workflow waits for the information from the user. If necessary, you can create a workflow to calculate a relative time.

Add a User Interaction to a Workflow

You request input parameters from users after a workflow has started by adding user interaction schema elements to the workflow. When a workflow encounters a user interaction schema element, it suspends its run and waits for the user to provide the data it requires.

Workflows prompt users by email to provide input parameters by using an input parameters dialog box, in the same way that they obtain input parameters when a workflow first starts. You set the layout of the input parameters dialog box in the user interaction schema element properties tabs.

Prerequisites

You must have created a workflow, opened it for editing in the workflow workbench, and added some elements to the workflow schema.

Procedure

1. Drag a User Interaction element to the appropriate position in the workflow schema.
2. Link the User Interaction element to the elements that precede and follow it.
3. Click the User Interaction element to show its properties tabs in the bottom half of the Schema tab.
4. Provide a name and a description for the user interaction in the General tab.
5. Click Attributes to define the user interaction attributes.
6. Click the security.group parameter’s Not set link to set the parameter value.
   - Select NULL to allow all users to respond to the request.
   - Set the security.group parameter to a specific LDAP user group to restrict the permission to respond to that group. This parameter is mandatory.
7. (Optional) Click the Create parameter/attribute in workflow to set the security.group parameter to a specific LDAP user group.

The Parameter information dialog box appears.
   a. Give the parameter an appropriate name.
   b. Select Create workflow ATTRIBUTE with the same name to create the LdapGroup attribute in the workflow.
   c. Click the parameter value’s Not set link.

The LdapGroup selection box appears.
d Search for the LDAP user group to which to restrict the permission to respond to the user interaction request.

For example, selecting the **Administrators** group means that only members of that group can respond to this request for user interaction.

e Click **OK**.

8 Click the **timeout.date** parameter's **not set** link to set the parameter value.

   - Select **NULL** to make the workflow wait indefinitely for the user to respond to the request for user interaction.
   - Set the **timeout.date** parameter to set a time and date until which the workflow waits for the user to respond to the request for user interaction.

9 (Optional) Click the **Create parameter/attribute in workflow** to set the **timeout.date** parameter to a specific timeout date.

   The **Parameter information** dialog box appears.

   a Give the parameter an appropriate name.

   b Select **Create workflow ATTRIBUTE with the same name** to create the **Date** attribute in the workflow.

   c Click the parameter value's **Not set** link.

      A calendar appears

   d Use the calendar to select an absolute date and time until which the workflow will wait for the user to respond.

      Alternatively, set the **Date** parameter to the output of a workflow or script that calculates a relative time and date.

   e Click **OK**.

10 Click the **External Inputs** tab.

11 Right-click in the **External Inputs** tab and select **Bind to workflow parameter/attribute** to define the parameters that the user must provide in the user interaction.

   - Select a parameter from the proposed list.

   - Click **Create parameter/attribute in workflow** to create a new input parameter if you have not yet defined this parameter in the workflow.

12 (Optional) Click the **Exception** tab.

13 (Optional) Define the exception parameter as an exception string named **errorCode**, in the same way as described in “Create Exception Bindings,” on page 34.

14 (Optional) Click the **Presentation** tab in the element properties tab.

   Define the layout and content of the input parameters dialog box that the user sees in the same way as described in “Creating the Input Parameters Dialog Box In the Presentation Tab,” on page 35.

   **Note** Take care to define the user interaction input parameters dialog box in the user interaction element's **Presentation** tab, not in the **Presentation** tab for the whole workflow.

15 Click **Save** at the bottom of the workflow workbench.

You defined a user interaction, in which the workflow waits for information from the user before continuing its run.
What to do next
You can call on workflows within other workflows.

Calling Workflows Within Workflows

Workflows can call on other workflows during their run. A workflow can start another workflow either because it requires the result of that workflow as an input parameter for its own run, or it can start a workflow and let it continue its own run independently. Workflows can also start a workflow at a given time in the future, or start multiple workflows simultaneously.

- **Workflow Elements that Call Workflows** on page 42
  There are four ways to call other workflows from within a workflow. Each way of calling a workflow or workflows is represented by a different workflow schema element.

- **Call a Workflow Synchronously** on page 44
  Calling a workflow synchronously runs the called workflow as a part of the run of the calling workflow. The calling workflow can use the called workflow's output parameters as input parameters when it runs its subsequent schema elements.

- **Call a Workflow Asynchronously** on page 45
  Calling a workflow asynchronously runs the called workflow independently of the calling workflow. The calling workflow continues its run without waiting for the called workflow to complete.

- **Schedule a Workflow** on page 45
  You can call a workflow from within a workflow and schedule it to start at a later time and date.

- **Call Several Workflows Simultaneously** on page 46
  Calling several workflows simultaneously runs the called workflows synchronously as a part of the run of the calling workflow. The calling workflow waits for all the called workflows to complete before it continues. The calling workflow can use the called workflows' results as input parameters when it runs its subsequent schema elements.

Workflow Elements that Call Workflows

There are four ways to call other workflows from within a workflow. Each way of calling a workflow or workflows is represented by a different workflow schema element.

- **Synchronous Workflows**
  A workflow can start another workflow synchronously. The called workflow runs as an integral part of the calling workflow's run, and runs in the same memory space as the calling workflow. The calling workflow starts another workflow, then waits until the end of the called workflow's run before it starts running the next element in its schema. Usually, you call a workflow synchronously because the calling workflow requires the output of the called workflow as an input parameter for a subsequent schema element. For example, a workflow can call the Start VM workflow to start a virtual machine, and then obtain the IP address of this virtual machine to pass to another element or to a user by email.

- **Asynchronous Workflows**
  A workflow can start a workflow asynchronously. The calling workflow starts another workflow, but the calling workflow immediately continues running the next element in its schema, without waiting for the result of the called workflow. The called workflows run with input parameters that the calling workflow defines, but the lifecycle of the called workflow is independent from the lifecycle of the calling workflow. Asynchronous workflows allow you to create chains of workflows that pass input parameters from one workflow to the next. For example, a workflow can create various objects during its run. The
workflow can then start asynchronous workflows that use these objects as input parameters in their own runs. When the original workflow has started all the required workflows and run its remaining elements, it ends. However, the asynchronous workflows it started continue their runs independently of the workflow that started them.

To make the calling workflow wait for the result of the called workflow, either use a nested workflow or create a scriptable task that retrieves the state of the workflow token of the called workflow and then retrieves the result of the workflow when it completes.

**Scheduled Workflows**  
A workflow can call a workflow but defer starting that workflow until a later time and date. The calling workflow then continues its run until it ends. Calling a scheduled workflow creates a task to start that workflow at the given time and date. When the calling workflow has run, you can view the scheduled workflow in the **Tasks** and **My Orchestrator** views in the Orchestrator client. Scheduled workflows only run once. You can schedule a workflow to run recurrently by calling the `Workflow.scheduleRecurrently` method in a scriptable task element in a synchronous workflow.

**Nested Workflows**  
A workflow can start several workflows simultaneously by nesting several workflows in a single schema element. All the workflows listed in the nested workflow element start simultaneously when the calling workflow arrives at the nested workflows element in its schema. Significantly, each nested workflow starts in a different memory space from the memory space of the calling workflow. The calling workflow waits until all the nested workflows have completed their runs before it starts running the next element in its schema. The calling workflow can thus use the results of the nested workflows as input parameters when it runs its remaining elements.

**Propagate Workflow Changes to other Workflows**

If you call a workflow from within another workflow, Orchestrator imports that workflow's input parameters into the calling workflow at the moment you add the workflow element to the schema, rather than referencing it.

As a consequence, if you change the called workflow after you have added it to another workflow, the calling workflow calls on the new version of the called workflow, but does not import any new input parameters. To prevent changes to workflows affecting the behavior of other workflows that call them, Orchestrator does not propagate the new input parameters automatically to the calling workflows.

To propagate parameters from one workflow to other workflows that call it, you must find the workflows that call the workflow, and synchronize the workflows manually.

**Prerequisites**

You need a workflow that another workflow or workflows call.

**Procedure**

1. Modify and save a workflow that other workflows call.
2. Close the workflow workbench.
3. Navigate to the workflow you changed in the hierarchical list in the **Workflows** view in the Orchestrator client.
4. Right-click the workflow, and select **References > Find Elements that Use this Element**.

A list of workflows that call this workflow appears.
5 Double-click a workflow in the list to highlight it in the **Workflows** view in the Orchestrator client.

6 Right-click the workflow, and select **Edit**.

   The workflow workbench opens.

7 Click the **Schema** tab in in the workflow workbench.

8 Right-click the workflow element for the changed workflow from the workflow schema and select **Synchronize > Synchronize Parameters**.

9 Click **Continue** in the confirmation dialog box.

10 Save and close the workflow workbench.

11 Repeat Step 5 to Step 10 for all the workflows that use the modified workflow.

You propagated a changed workflow to other workflows that call it.

---

### Call a Workflow Synchronously

Calling a workflow synchronously runs the called workflow as a part of the run of the calling workflow. The calling workflow can use the called workflow’s output parameters as input parameters when it runs its subsequent schema elements.

You call workflows synchronously from another workflow by using the **Workflow** element.

**Prerequisites**

You must have created a workflow, opened it for editing in the workflow workbench, and added some elements to the workflow schema.

**Procedure**

1 Drag a **Workflow** element from the **Action & Workflow** menu to the appropriate position in the workflow schema.

   The **Choose workflow** selection dialog box appears.

2 Search for the desired workflow by entering part of its name in the **Search** text box.

   If the search returns a partial result, narrow your search criterion or increase the number of search results in the **Tools > User Preferences** menu in the client.

3 Select the desired workflow from the list and click **OK**.

4 Link the **Workflow** element to the elements that precede and follow it in the workflow schema.

5 Click the **Workflow** element to show its properties tabs in the bottom half of the **Schema** tab.

6 Bind the required input parameters to the workflow in the workflow schema element’s **IN** tab.

7 Bind the required output parameters to the workflow in the workflow schema element’s **OUT** tab.

8 Define the workflow’s exception behavior in the **Exceptions** tab.

9 Click **Save** at the bottom of the workflow workbench.

You called a workflow synchronously from within another workflow. When the workflow reaches the synchronous workflow during its run, the synchronous workflow starts, and the initial workflow waits for it to complete before continuing its run.

**What to do next**

You can call a workflow asynchronously from within a workflow.
Call a Workflow Asynchronously

Calling a workflow asynchronously runs the called workflow independently of the calling workflow. The calling workflow continues its run without waiting for the called workflow to complete.

You call workflows asynchronously from another workflow by using the Asynchronous Workflow element.

Prerequisites

You must have created a workflow, opened it for editing in the workflow workbench, and added some elements to the workflow schema.

Procedure

1. Drag an Asynchronous Workflow element from the Action & Workflow menu to the appropriate position in the workflow schema.
   The Choose workflow selection dialog box appears.
2. Search for the desired workflow by entering part of its name in the Search text box.
3. Select the desired workflow from the list and click OK.
4. Link the Asynchronous Workflow element to the elements that precede and follow it in the workflow schema.
5. Click the Asynchronous Workflow element to show its properties tabs in the bottom half of the Schema tab.
6. Bind the required input parameters to the workflow in the asynchronous workflow element's IN tab.
7. Bind the required output parameter in the asynchronous workflow element's OUT tab.
   You can bind the output parameter either to the called workflow, or to that workflow's result.
   - Bind to the called workflow to return that workflow as an output parameter
   - Bind to the called workflow's workflow token to return the result of running the called workflow.
8. Define the asynchronous workflow element's exception behavior in the Exceptions tab.
9. Click Save at the bottom of the workflow workbench.

You called a workflow asynchronously from within another workflow. When the workflow reaches the asynchronous workflow during its run, the asynchronous workflow starts, and the initial workflow continues its run without waiting for the asynchronous workflow to finish.

What to do next

You can schedule a workflow to start at a later time and date.

Schedule a Workflow

You can call a workflow from within a workflow and schedule it to start at a later time and date.

You schedule workflows in another workflow by using the Schedule Workflow element.

Prerequisites

You must have created a workflow, opened it for editing in the workflow workbench, and added some elements to the workflow schema.
Procedure

1. Drag a Schedule Workflow element from the Action & Workflow menu to the appropriate position in the workflow schema.

   The Choose Workflow selection dialog box appears.

2. Search for the desired workflow by entering part of its name in the Search text box.

3. Select the desired workflow from the list and click OK.

4. Link the Schedule Workflow element to the elements that precede and follow it in the workflow schema.

5. Click the Schedule Workflow element to show its properties tabs in the bottom half of the Schema tab.

6. Click the IN property tab.

   A parameter named workflowScheduleDate appears in the list of properties to define, together with the input parameters of the calling workflow.

7. Click the workflowScheduleDate parameter’s Not set link to set the parameter.

   The Set parameter dialog box appears.

8. Click Create parameter/attribute in workflow to create the parameter and set the parameter’s value.

   The Parameter information dialog box opens.

9. Click the ValueNot set link to set the parameter value.

   A calendar appears.

10. Use the calendar to set the date and time to start the scheduled workflow.

11. Click OK.

12. Bind the remaining input parameters to the scheduled workflow in the scheduled workflow element’s IN tab.

13. Bind the required output parameters to the Task object in the scheduled workflow element’s OUT tab.

14. Define the scheduled workflow element’s exception behavior in the Exceptions tab.

15. Click Save at the bottom of the workflow workbench.

You scheduled a workflow to start at a given time and date from within another workflow.

What to do next

You can call multiple workflows simultaneously within a workflow.

Call Several Workflows Simultaneously

Calling several workflows simultaneously runs the called workflows synchronously as a part of the run of the calling workflow. The calling workflow waits for all the called workflows to complete before it continues. The calling workflow can use the called workflows’ results as input parameters when it runs its subsequent schema elements.

You call several workflows simultaneously from another workflow by using the Nested Workflows element. You can use nested workflows to run workflows with different user credentials to those of the user of the calling workflow.

Prerequisites

You must have created a workflow, opened it for editing in the workflow workbench, and added some elements to the workflow schema.
Procedure

1. Drag a Nested Workflows element from the Action & Workflow menu to the appropriate position in the workflow schema.

   The Choose workflow selection dialog box appears.

2. Search for a workflow to start by entering part of its name in the Search text box.

3. Select the appropriate workflow from the list and click OK.

4. Link the Nested Workflows element to the elements that precede and follow it in the workflow schema.

5. Click the Nested Workflows element to show its properties tabs in the bottom half of the Schema tab.

6. Click the Workflows schema element properties tab.

   The workflow you selected in Step 3 appears in the tab.

7. Set the IN and OUT bindings for this workflow in the IN and OUT tabs in the right panel of the Workflows schema element properties tab.

8. Click the Connection Info tab in the right panel of the Workflows schema element properties tab.

   The Connection Info tab allows you to access workflows stored in a different server to the local one, using the appropriate credentials.

9. To access workflows on a remote server, click the Remote Not set link and provide a host name or IP address for the remote server.

10. Define the credentials with which to access the remote server.

    - Select Inherit to use the same credentials as the user who runs the calling workflow.
    - Click the Dynamic Not set link to select a set of dynamic credentials that a parameter of the credentials type defines elsewhere in the workflow.
    - Click the Static Not set link to enter the credentials directly.

11. Click the Add Workflow button in the Workflows tab to select more workflows to add to the nested workflow element.

12. Repeat Step 2 to Step 10 to define the settings for each of the workflows you add.

13. Click the nested workflow element in the workflow schema.

   The number of workflows nested in the element appears as a numeral on the nested workflows element.

You called several workflows simultaneously from within a workflow.

What to do next

You can define long-running workflows.

Developing Long-Running Workflow Elements

A workflow in a waiting state consumes system resources by constantly polling the object from which it is awaiting a response. If you know that a workflow element will potentially wait for a long time before it receives the response it requires, you can implement that workflow element as a long-running element.

Every running workflow consumes a thread. When a workflow reaches a long-running element, the long-running element sets the workflow thread into a passive state. The long-running element then passes the workflow information to a single thread that polls the system for all long-running workflow elements running in the server. Rather than each long-running element constantly attempting to retrieve information from the system, long-running workflow elements remain for a set duration, while the long-running workflow thread polls the system on its behalf.
You set the duration of the wait in one of two ways:

- Set a timer that suspends the workflow until a certain time and date, encapsulated in a `Date` object. You implement timer-based long-running workflow elements by including a **Waiting Timer** element in the schema.

- Define a trigger event, encapsulated in a `Trigger` object, that restarts the workflow after the trigger event occurs. You implement trigger-based long-running workflow elements by adding a **Waiting Event** element or a **User Interaction** element in the schema.

### Create a Date Object

You encapsulate the time and date that a timer-based long-running workflow awaits in a `Date` object. A **Waiting Timer** element binds this `Date` object to its `timer.date` input parameter.

You can set an absolute time and date in the `Date` object directly. When date arrives or when the time and date meet the condition of the long-running workflow, the workflow reactivates and continues its run. For example, you can set the workflow to reactivate at midday on a given date. Alternatively, you can create an element or workflow that calculates and generates a `Date` object according to a function that you define.

The following procedure provides an example of how to create such a function.

**Prerequisites**

You must have created a workflow, opened it for editing in the workflow workbench, and added some elements to the workflow schema.

**Procedure**

1. Drag a **Scriptable Task** element from the **Generic** menu into the schema of a workflow.
2. Link the **Scriptable Task** element to the elements that precede and follow it in the workflow schema.
3. Click the **Scriptable Task** element to show its properties tabs in the bottom half of the **Schema** tab.
4. Provide a name and description for the function in **General** properties tab.
5. In the **OUT** properties tab, create an output parameter with the following properties.
   a. Create the Name property with the value `timerDate`.
   b. Create the Type property with the value `Date`.
6. Define a function to calculate and generate a `Date` object named `timerDate` in the **Scripting** tab.
   
   The timeout period is a relative delay in milliseconds.
   
   For example, you could create a `Date` object by implementing the following JavaScript function.
   
   ```javascript
   timerDate = new Date();
   System.log( "Current date : '" + timerDate + "'" );
   timerDate.setTime( timerDate.getTime() + (50 * 1000) );
   System.log( "Waiting timer will expire at '" + timerDate + "'" );
   ```

   The preceding example JavaScript function defines a `Date` object that obtains the current date and time using the `getTime` method and adds 50 seconds. The **Scriptable Task** element generates this value as its output parameter, to which a **Waiting Timer** element can bind. When the workflow arrives at the **Waiting Timer** element, it will suspend its run and wait for 50 seconds before continuing.

7. Click **Save** at the bottom of the workflow workbench.

You created a function that calculates and generates a `Date` object. A **Waiting Timer** element can receive this `Date` object as an input parameter, to suspend a long-running workflow until the date encapsulated in this object.
What to do next

You must add a Waiting Timer element to a workflow to implement a timer-based long-running workflow.

Create a Timer-Based Long-Running Workflow

If you know a workflow will have to wait for a response from an outside source for a predictable time, you can implement it as a timer-based long-running workflow. A timer-based long-running workflow waits until a given time and date before resuming.

You implement a workflow as a timer-based long-running workflow by using the Waiting Timer element.

Prerequisites

You must have created a workflow, opened it for editing in the workflow workbench, and added some elements to the workflow schema.

Procedure

1. Drag a Waiting Timer element from the Generic menu to the position in the workflow schema at which to suspend the workflow’s run.

2. Link the Waiting Timer element to the elements that precede and follow it in the workflow schema.

   If you implement a scriptable task to calculate the time and date, this element must immediately precede the Waiting Timer element.

3. Click the Waiting Timer element to show its properties tabs in the bottom half of the Schema tab.

4. Provide a description of the reason for implementing the timer in the General properties tab.

5. Click the Attributes properties tab.

   The timer.date parameter appears in the list of attributes.

6. Click the timer.date parameter’s Not set link to bind the parameter to an appropriate Date object.

   The Waiting Timer selection dialog box opens, presenting a list of possible bindings.

   - Select a predefined Date object from the proposed list, for example one defined by a Scriptable Task element elsewhere in the workflow.
     - Alternatively, create a Date object that sets a specific date and time for the workflow to await.

7. (Optional) Create a Date object that sets a specific date and time that the workflow awaits.

   a. Click Create parameter/attribute in workflow in the Waiting Timer selection dialog box.

      The Parameter information dialog box appears.

   b. Give the parameter an appropriate name.

   c. Leave the type set to Date.

   d. Click Create workflow ATTRIBUTE with the same name.

   e. Click the Value property’s Not set link to set the parameter value.

      A calendar appears.

   f. Use the calendar to set a date and time at which to restart workflow.

   g. Click OK.

8. Click Save at the bottom of the workflow workbench.

You defined a timer that suspends a timer-based long-running workflow until a set time and date.
What to do next

You can create a long-running workflow that waits for a trigger event before continuing.

Create a Trigger Object

Trigger objects monitor event triggers that plug-ins define. For example, the vCenter Server plug-in defines these events as Task objects. When the task ends, the trigger sends a message to a waiting trigger-based long-running workflow element, to restart the workflow.

The time-consuming event for which a trigger-based long-running workflow waits must return a VC:Task object. For example, the startVM action to start a virtual machine returns a VC:Task object, so that subsequent elements in a workflow can monitor its progress. A trigger-based long-running workflow’s trigger event requires this VC:Task object as an input parameter.

You create a Trigger object in a JavaScript function in a Scriptable Task element. This Scriptable Task element can be part of the trigger-based long-running workflow that waits for the trigger event. Alternatively, it can be part of a different workflow that provides input parameters to the trigger-based long-running workflow. The trigger function must implement the createEndOfTaskTrigger() method from the Orchestrator API.

**IMPORTANT** You must define a timeout period for all triggers, otherwise the workflow can wait indefinitely.

Prerequisites

You must have created a workflow, opened it for editing in the workflow workbench, and added some elements to the workflow schema. The workflow must declare a VC:Task object as an attribute or input parameter, such as a VC:Task object from a workflow or workflow element that starts or clones a virtual machine.

Procedure

1. Drag a Scriptable Task element from the Generic menu into the schema of a workflow.
2. Link the Scriptable Task element to the elements that precede and follow it in the workflow schema.
   One of the elements that precedes the Scriptable Task must generate a VC:Task object as its output parameter.
3. Click the Scriptable Task element to show its properties tabs in the bottom half of the Schema tab.
4. Provide a name and description for the trigger in Info properties tab.
5. Click the IN properties tab.
6. Right-click in the IN tab and select Bind to workflow parameter/attribute.
   The input parameter selection dialog box opens.
7. Select or create an input parameter of the type VC:Task.
   This VC:Task object represents the time-consuming event that another workflow or element launches.
8. (Optional) Select or create an input parameter of the Number type to define a timeout period in seconds.
9. Click the OUT properties tab.
10. Right-click in the OUT tab and select Bind to workflow parameter/attribute.
    The output parameter selection dialog box opens.
11. Create an output parameter with the following properties.
    a. Create the Name property with the value trigger.
    b. Create the Type property with the value Trigger.
c Click **Create ATTRIBUTE with same name** to create the attribute.

d Leave the value as **Not set**.

12 Define any exception behavior in the **Exceptions** properties tab.

13 Define a function to generate a **Trigger** object in the **Scripting** tab.

   For example, you could create a **Trigger** object by implementing the following JavaScript function.
   
   ```javascript
   trigger = task.createEndOfTaskTrigger(timeout);
   ```

   The `createEndOfTaskTrigger()` method returns a **Trigger** object that monitors a VC:Task object named `task`.

14 Click **Save** at the bottom of the workflow workbench.

You defined a workflow element that creates a trigger event for a trigger-based long-running workflow. The trigger element generates a **Trigger** object as its output parameter, to which a **Waiting Event** element can bind.

**What to do next**

You must bind this trigger event to a **Waiting Event** element in a trigger-based long-running workflow.

---

**Create a Trigger-Based Long-Running Workflow**

If you know a workflow will have to wait for a response from an outside source during its run, but do not know how long that wait will last, you can implement it as a trigger-based long-running workflow. A trigger-based long-running workflow waits for a defined trigger event to occur before resuming.

You implement a workflow as a trigger-based long-running workflow by using the **Waiting Event** element. When the trigger-based long-running workflow arrives at the **Waiting Event** element, it will suspend its run and wait in a passive state until it receives a message from the trigger. During the waiting period, the passive workflow does not consume a thread, but rather the long-running workflow element passes the workflow information to the single thread that monitors all long-running workflows in the server.

**Prerequisites**

You must have created a workflow, opened it for editing in the workflow workbench, added some elements to the workflow schema, and defined a trigger event, encapsulated in a **Trigger** object.

**Procedure**

1 Drag a **Waiting Event** element from the **Generic** menu to the position in the workflow schema at which you want to suspend the workflow's run.

2 Link the **Waiting Event** element to the elements that precede and follow it in the workflow schema.

   The scriptable task that declares the trigger must immediately precede the **Waiting Event** element.

3 Click the **Waiting Event** element to show its properties tabs in the bottom half of the **Schema** tab.

4 Provide a description of the reason for the wait in the **General** properties tab.

5 Click the **Attributes** properties tab.

   The `trigger.ref` parameter appears in the list of attributes.

6 Click the `trigger.ref` parameter's **Not set** link to bind the parameter to an appropriate **Trigger** object.

   The **Waiting Event** selection dialog box opens, presenting a list of possible parameters to which to bind.

7 Select a predefined **Trigger** object from the proposed list.

   This **Trigger** object represents a trigger event that another workflow or workflow element defines.
8 Define any exception behavior in the **Exceptions** properties tab.

9 Click **Save** at the bottom of the workflow workbench.

You defined a workflow element that suspends a trigger-based long-running workflow, that waits for a specific trigger event before restarting.

**What to do next**

You can run a workflow.

---

**Configuration Elements**

A configuration element is a list of attributes you can use to configure constants across a whole Orchestrator server deployment.

All the workflows, actions, policies, and Web views running in a particular Orchestrator server can use the attributes you set in a configuration element. Setting attributes in configuration elements allows you to make the same attribute values available to all the workflows, actions, policies, and Web views running in the Orchestrator server.

If you create a package containing a workflow, action, policy, or Web view that uses an attribute from a configuration element, Orchestrator automatically includes the configuration element in the package, but not its value. If you import a package containing a configuration element into another Orchestrator server, the configuration element attribute values are not set. You must set the attributes with values appropriate to the server in which you have imported the package. For example, if you create a workflow that requires attribute values that depend on the Orchestrator server on which it runs, setting those attributes in a configuration element allows you to export that workflow so that another Orchestrator server can use it. If you set the server-specific attributes directly in the workflow, the workflow might not work if you import it into another server, because it might not find the attribute values it requires. Because the attribute values in an imported configuration element are not set, you have to set them with values appropriate to the new server. Configuration elements therefore allow you to exchange workflows, actions, policies and Web views between servers more easily.

---

**Create a Configuration Element**

Configuration elements allow you to set common attributes across an Orchestrator server. All elements running in the server can call on the attributes you set in a configuration element. Creating configuration elements allows you to define common attributes once in the server, rather than individually in each element.

You create configuration elements in the **Configurations** view in the Orchestrator client.

**Procedure**

1 Click the **Configurations** view in the Orchestrator client.

2 Right-click a folder in the hierarchical list of folders and select **New category** to create a new folder.

3 Provide a name for the category and click **OK**.

4 Right-click the folder you created and select **New element**.

5 Provide a name for the configuration element and click **OK**.

6 Right-click the element and select **Edit**.

   The configuration element workbench opens.

7 Increment the version number by clicking the version digits in the **General** tab and providing a version comment.
8 Check the **Allowed Operations** check boxes in the **General** tab to define the operations that users can perform on this configuration element.

You can allow users to perform the following actions on the workflow.

- View the content of the configuration element
- Add the configuration element to a package
- Edit the configuration element

9 Provide a description of the configuration element in the **Description** text box in the **General** tab.

10 Click the **Attributes** tab.

11 Right-click in the tab and select **Add attribute** to create a new attribute.

12 Click the attribute values under **Name**, **Type**, **Value**, and **Description** to set the attribute name, type, value, and description.

13 Click the **Permissions** tab.

14 Click **Add access rights** to grant permission to access this configuration element to a group of users.

15 Search for a user group in the **Search** text box and select the relevant user group from the proposed list.

16 Check the appropriate check boxes to set the access rights for the selected user group.

You can set the following permissions on the configuration element.

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>The users can view the configuration element, but cannot view the schemas or scripting.</td>
</tr>
<tr>
<td>Inspect</td>
<td>The users can view the configuration element, including the schemas and scripting.</td>
</tr>
<tr>
<td>Execute</td>
<td>The users can run the elements in the configuration element.</td>
</tr>
<tr>
<td>Edit</td>
<td>The users can edit the elements in the configuration element.</td>
</tr>
<tr>
<td>Admin</td>
<td>The users can set permissions on the elements in the configuration element.</td>
</tr>
</tbody>
</table>

17 Click **Save and Close** to exit the configuration element workbench.

You defined a configuration element than sets common attributes across an Orchestrator server.

**What to do next**

You can use the configuration element to provide attributes to workflows or actions.

**Workflow User Permissions**

Orchestrator defines levels of permissions that you can apply to users or groups.

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>The user can view the elements in the workflow, but cannot view the schemas or scripting.</td>
</tr>
<tr>
<td>Inspect</td>
<td>The user can view the elements in the workflow, including the schemas and scripting.</td>
</tr>
<tr>
<td>Execute</td>
<td>The user can run the workflow</td>
</tr>
<tr>
<td>Edit</td>
<td>The user can edit the workflow</td>
</tr>
<tr>
<td>Admin</td>
<td>The user can set permissions on the workflow.</td>
</tr>
</tbody>
</table>
Set User Permissions on a Workflow

You set different levels of permission on a workflow to limit the access that different users or user groups can have to that workflow.

You select the different users and user groups for which to set permissions from the users and user groups in the Orchestrator LDAP server.

Prerequisites

You must have created a workflow, opened it for editing in the workflow workbench, and added to it the necessary elements.

Procedure

1. Click the Permissions tab in the workflow workbench.
2. Click the Add access rights link to define permissions for a new user or user group.
3. Search for a user or user group by entering text in the Search text box.
   - The search results show all the users and user groups from the Orchestrator LDAP server that match the search.
4. Select a user or user group and click OK.
5. Right-click the user and select Add access rights.
6. Check the appropriate check boxes to set the level of permissions for this user and click OK.
   - The levels of permissions are not cumulative. To grant a user permission to view the workflow, inspect the schema and scripting, run and edit the workflow, and change the permissions, you must check all check boxes.
7. Click Save and Close to exit the package editor.

You set the appropriate user permissions on a workflow.

Running Workflows

A workflow runs according to a logical flow of events.

When you run a workflow, each schema element in the workflow runs according to the following sequence.

1. The workflow binds the workflow token attributes and input parameters to the schema element's input parameters.
2. The schema element runs.
3. The schema element's output parameters are copied to the workflow token attributes and workflow output parameters.
4. The workflow token attributes and output parameters are stored in the database.
5. The next schema element starts running.

This sequence repeats for each schema element until the end of the workflow.
Workflow Token Check Points

When a workflow runs, each schema element is a check point. After each schema element runs, Orchestrator stores workflow token attributes in the database, and the following schema element starts running. If the workflow stops unexpectedly, the next time the Orchestrator server restarts, the currently active schema element runs again, and the workflow continues from the start of the schema element that was running when the interruption happened. However, Orchestrator does not implement transaction management or a roll-back function.

End of Workflow

The workflow ends if the current active schema element is an end element. Workflows with no end element are invalid and will not run if automatic validation is activated. After the workflow reaches the end element, other workflows or applications can use the workflow’s output parameters.

Validating Workflows

Orchestrator provides a workflow validation tool. Validating a workflow helps identify errors in the workflow and checks that the data flows from one element to the next correctly.

When you validate a workflow, the validation tool creates a list of any errors or warnings. Clicking an error in the list highlights the workflow element that contains the error.

If you run the validation tool in the workflow workbench, the tool provides suggested quick fixes for the errors it detects. Some quick fixes require you to provide additional information or input parameters. Some quick fixes resolve the error for you.

Note: Workflow validation checks that data flows through the workflow correctly and that all necessary links and bindings are in place. Workflow validation does not check the data processing performed by each element in the workflow. Consequently, it is possible for a valid workflow to run incorrectly and produce erroneous results if a function in a schema element is incorrect.

By default, Orchestrator always performs workflow validation when you run a workflow. You can change the default validation behavior in Tools > User Preferences in the Orchestrator client. For example, sometimes during workflow development you might want to run a workflow that you know to be invalid, for testing purposes.

Validate a Workflow and Fix Validation Errors

You can validate workflows in either the Orchestrator client or in the workflow workbench. However, you can only fix validation errors if you have opened the workflow for editing in the workflow workbench.

Prerequisites

You must have a complete workflow to validate, with schema elements linked and bindings defined.

Procedure

1. Click the Workflows view.
2. Navigate to a workflow in the Workflows hierarchical list.
3. (Optional) Right-click the workflow and select Validate workflow.

   If the workflow is valid, a confirmation message appears. If the workflow is invalid, a list of errors appears.
4. (Optional) Close the Workflow Validation dialog box.
5. Right-click the workflow and select Edit to open the workflow workbench.
6 Click the Schema tab.
7 Click the Validate button in the Schema tab toolbar.

If the workflow is valid, a confirmation message appears. If the workflow is invalid, a list of errors appears.
8 If the workflow is invalid, click on an error message.

The validation tool highlights the schema element in which the error occurs by adding a red icon to it. Where possible, the validation tool proposes a Quick fix action.

- If you agree with the proposed Quick fix action, click it to perform that action.
- If you disagree with the proposed Quick fix action, close the Workflow Validation dialog box and fix the schema element manually.

Always check that the Quick Fix that Orchestrator proposes is appropriate. For example, the proposed action might be to delete an unused attribute, when in fact that attribute has not been correctly bound.
9 Repeat the preceding steps until you have eliminated all validation errors.

You validated a workflow, and possibly fixed any validation errors.

What to do next
You can run the workflow.

Run a Workflow
After you create and validate a workflow, you can run it.

This procedure shows how to run a workflow using an existing workflow from the Orchestrator library, Create VM (Simple).

Prerequisites
You must have a valid workflow.

Procedure
1 Click the Workflows view in the Orchestrator client.
2 In the workflows hierarchical list, open Library > vCenter > Virtual Machine Management > Basic to navigate to the Create VM (Simple) workflow.
3 Right-click the Create VM (Simple) Workflow and select Execute Workflow.

The input parameters dialog box opens.
4 Enter the following information into the Execute Workflow input parameters dialog box to create a virtual machine in a vCenter Server connected to Orchestrator.

a Name the virtual machine orchestrator-test.

b Click the VM Folder value's Not Set link.

A selection dialog box opens.

c Press Enter without entering any text in the Search text box in the selection dialog box.

The selection box lists all the objects of the type VC:VmFolder that the infrastructure contains. If the search returns a partial list, narrow your search criteria or increase the number of search results in the Tools > User Preferences menu in the Orchestrator client.

d Click the VC:VmFolder object of your choice and click Select.

e Enter appropriate numeric values for Size of the new disk in GB and Memory size in MB.

f Select an appropriate number of CPUs from the Number of virtual CPUs drop-down menu.
g Click the **Guest OS** value’s **Not Set** link and select a guest operating system from the proposed list.

h Click the **Host on which VM will be created** value’s **Not Set** link and select a host machine from the proposed list.

i Click the **Resource pool** value’s **Not Set** link and navigate through the vCenter Server infrastructure’s hierarchy to the resource pool of your choice.

j Click the **Network to connect to** value’s **Not Set** link and select a **VC:Network** object from the proposed list.

k Click the **Datastore on which the VM will be created** value’s **Not Set** link and select a **VC:Datastore** object from the proposed list.

5 Click **Submit** to run the workflow.

A workflow token appears in a leaf node under the **Create VM (Simple)** workflow, showing the workflow running icon.

6 Click the workflow token to view the status of the workflow as it runs.

7 Click the **Events** tab in the workflow token view to follow the progress of the workflow token until it completes.

8 Click the **Inventory** view in the Orchestrator client.

9 Navigate through the vCenter Server infrastructure’s hierarchy to the resource pool you defined in **Step 4**.

   If the virtual machine does not appear in the list, click the refresh button to reload the inventory.

   The orchestrator-test virtual machine is present in the resource pool.

10 (Optional) Right-click the orchestrator-test virtual machine in the **Inventory** view to see a contextual list of the workflows you can run on the orchestrator-test virtual machine.

11 (Optional) Select **Destroy VM** to remove the orchestrator-test virtual machine from the inventory.

The workflow or workflows ran successfully.

**Develop a Simple Example Workflow**

Developing a simple example workflow demonstrates the most common steps in the workflow development process.

The example workflow, called Start VM and Send Email, starts an existing virtual machine in the vCenter Server and sends an email to the administrator to confirm that the virtual machine has started.

The example workflow performs the following tasks:

1 Prompts the user for a virtual machine to start.

2 Prompts the user for the email address of a person to inform that the virtual machine has started, or that an error occurred.

3 Checks whether or not the requested virtual machine is already powered on.

4 Sends the request to the vCenter Server to start the requested virtual machine.

5 Waits for vCenter Server to start up the requested virtual machine, and returns an error if the virtual machine fails to start or if starting the virtual machine takes too long.

6 Waits for vCenter Server to start up VMware Tools on the virtual machine. Returns an error if the virtual machine fails to start or if starting VMware Tools takes too long.
7 Verifies that the virtual machine is running.
8 Sends a notification email to the relevant person, to inform them that the machine has started or that an error occurred.

The process for developing a simple workflow consists of the following tasks.

**Prerequisites**

Before you attempt to develop this simple workflow example, read all the other sections of Chapter 2, “Developing Workflows,” on page 13.

**Procedure**

1 Create the Simple Workflow Example on page 59
   The first step in the workflow development process is to create the workflow.
2 Define the Simple Workflow Example Parameters on page 60
   You define workflow attributes and parameters in the workflow workbench.
3 Create the Simple Workflow Example Schema on page 60
   You create a workflow’s schema in the Schema tab of the workflow workbench. The workflow schema contains the elements that the workflow runs.
4 Link the Simple Workflow Example Elements on page 62
   You link a workflow’s elements in the Schema tab of the workflow workbench. The linking defines the flow of data through the workflow.
5 Create Workflow Zones on page 63
   You can emphasize different zones in workflow by adding workflow notes of different colors. Creating different workflow zones helps to make complicated workflow schema easier to read and understand.
6 Define the Simple Workflow Example Decision Bindings on page 64
   You bind a workflow’s elements together in the Schema tab of the workflow workbench. Decision bindings define how decision elements compare the input parameters received to the decision statement, and generate output parameters according to whether the input parameters match the decision statement.
7 Bind the Simple Workflow Example Action Elements on page 65
   You bind a workflow’s elements together in the Schema tab of the workflow workbench. Bindings define how the action elements process input parameters and generate output parameters.
8 Bind the Simple Workflow Example Scripted Task Elements on page 68
   You bind a workflow’s elements together in the Schema tab of the workflow workbench. Bindings define how the scripted task elements process input parameters and generate output parameters. You also bind the scriptable task elements to their JavaScript functions.
9 Define the Simple Example Workflow Exception Bindings on page 76
   You define exception bindings in the Schema tab in the workflow workbench. Exception bindings define how elements process errors.
10 Set the Simple Workflow Example Attribute Read-Write Properties on page 76
    You can define whether parameters and attributes are read-only constants or writable variables. You can also set limitations on the values that users can provide for input parameters.
11 Set the Simple Workflow Example Parameter Properties on page 77
    You set the parameter properties in the Presentation tab in the workflow workbench. Setting the parameter properties affects the behavior of the parameter, and places constraints on the possible values for that parameter.
12 Set the Layout of the Simple Workflow Example Input Parameters Dialog Box on page 78
You create the layout, or presentation, of the input parameters dialog box in the Presentation tab of the workflow workbench. The input parameters dialog box opens when users run a workflow, and is the means by which users enter the input parameters with which the workflow runs.

13 Validate and Run the Simple Workflow Example on page 79
After you create a workflow, you can validate it to discover any possible errors. If the workflow contains no errors, you can run it.

Create the Simple Workflow Example

The first step in the workflow development process is to create the workflow.

This example creates a simple workflow called Start VM and Send Email.

Prerequisites
To create the simple workflow example, you must have the following components installed and configured on the system.

- vCenter 4.0, controlling some virtual machines, at least one of which is powered off
- Access to an SMTP server
- A valid email address

For details about how to install and configure vCenter, see the ESX and vCenter Server Installation Guide. For details about how to configure Orchestrator, see the Orchestrator 4.0 Installation and Configuration Guide.

To write a workflow, you need an Orchestrator log in with at least View, Execute, Inspect, Edit, and preferably Admin permissions on the server or on the workflow category in which you are working.

Procedure
1 Start the Orchestrator Client interface.
2 Log in using the Orchestrator username and password.
3 Click Workflows, on the left side of the client interface.
4 Right-click the root of the workflows hierarchical list and select Add Category.
5 Name the new category Workflow Examples and click OK.
6 Right-click the Workflow Examples category and select New Workflow.
7 Name the new workflow Start VM and Send Email and click OK.
8 Right-click the Start VM and Send Email workflow and select Edit.

The workflow workbench opens.

9 In the General tab, click the version number digits to increment the version number.

Because this is the initial creation of the workflow, set the version to 0.0.1.

10 Check the Allowed operations check boxes in the General tab to set the operations the users can perform on the workflow.

11 Click the Server restart behavior value in the General tab to set whether the workflow resumes after a server restart.
12 Provide a description of what the workflow does in the **Description** text box in the **General** tab. For example, you can add the following description.

*This simple workflow starts a virtual machine and sends a confirmation email to the Orchestrator administrator.*

13 Click **Save** at the bottom of the **General** tab.

You created a new workflow, but you have not defined its functions yet.

**What to do next**

You must define the workflow’s attributes and input and output parameters.

### Define the Simple Workflow Example Parameters

You define workflow attributes and parameters in the workflow workbench.

**Prerequisites**

You must have created the Start VM and Send Email workflow, and opened it for editing in the workflow workbench.

**Procedure**

1. Click the **Inputs** tab in the workflow workbench.
2. Right-click in the **Inputs** tab and select **Add Parameter**.
   
   A parameter named **arg_in_0** appears in the Inputs tab.
3. Click **arg_in_0**.
4. Type the name **vm** in the **Choose Attribute Name** dialog box and click **OK**.
5. Click the **Type** text box and type **vc:virtualM** in the search text box in the parameter type dialog box.
6. Select **VC:VirtualMachine** from the proposed list of parameter types and click **Accept**.
7. Add a description of the parameter in the description text box.
   
   For example, type **The virtual machine to power on**.
8. Repeat the above process to create a second input parameter, with the following values.
   
   - **Name**: **toAddress**
   - **Type**: **String**
   - **Description**: **The email address of the person to inform of the result of this workflow**
9. Click **Save** at the bottom of the **Inputs** tab.

You defined the workflow’s input parameters.

**What to do next**

You must create the workflow’s schema.

### Create the Simple Workflow Example Schema

You create a workflow’s schema in the **Schema** tab of the workflow workbench. The workflow schema contains the elements that the workflow runs.

**Prerequisites**

You must have created the Start VM and Send Email workflow and defined its parameters.
Procedure
1. Click the **Schema** tab in the workflow workbench.
2. Click the **Generic** menu on the left of the **Schema** tab.
3. Drag a decision element to under the start element in the schema.
4. Double-click the decision element and change its name to **VM powered on?**
5. Click **Action & Workflow** and drag an action element to under the decision element.
   - The action selection dialog box appears.
6. Type **start** in the dialog box search text box.
7. Select the **startVM** action and click **Select**.
   
   **NOTE** If you installed the VMware Infrastructure 3.5 plug-in, two **startVM** actions are listed. Select the vCenter 4.0 version, which has a result type value of **VC:Task**.
8. Drag the following action elements into the schema, one beneath the other under the **startVM** action element.
   - **vim3WaitTaskEnd**: Suspends the workflow run and polls an ongoing vCenter Server task at regular intervals, until that task is finished. In the present example, the **startVM** action starts a virtual machine and the **vim3WaitTaskEnd** action makes the workflow wait while the virtual machine starts up. After the virtual machine starts, the **vim3WaitTaskEnd** lets the workflow resume.
   - **vim3WaitToolsStarted**: Suspends the workflow run and waits until VMware Tools have started on the target virtual machine.
9. Click the **Generic** menu and drag a scriptable task element under the **vim3WaitToolsStarted** action element.
10. Double-click the scriptable task element and rename it **OK**.
11. Drag another scriptable task element it to the left of the **startVM** action element.
   - Name this scripted element **Already started**.
12. Drag more scripted elements into the schema, as follows.
   - Drag a scripted element to the right of **startVM** and name it **startVM failed**.
   - Drag a scripted element to the right of **vim3WaitTaskEnd** and name it **Timeout 1**.
   - Drag a scripted element to the right of **vim3WaitToolsStarted** and name it **Timeout 2**.
   - Drag a scripted element to the right of **OK** and name it **Send Email**.
   - Drag a scripted element to the right of **Timeout 2** and name it **Send Email Failed**.
13. Drag an end element to the right of **Send Email**.
14. Click **Save** at the bottom of the workflow workbench’s **Schema** tab.
You have laid out the structure of the workflow.

**What to do next**
You must now link the workflow elements together.
Link the Simple Workflow Example Elements

You link a workflow's elements in the Schema tab of the workflow workbench. The linking defines the flow of data through the workflow.

Prerequisites

You must have created the Start VM and Send Email workflow, defined its parameters, and laid out its schema.

Procedure

1. Click the connector tool button in the toolbar at the top of the Schema tab in the workflow workbench.
2. Click the start element, and holding the left mouse button down, move the pointer to the VM Powered On? decision element.

   You have linked the start element to the decision element.

3. Link the remaining elements as described in the following table.

<table>
<thead>
<tr>
<th>Click</th>
<th>Link to</th>
<th>Type of Arrow</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left side of VM Powered On? decision element</td>
<td>Already Started scriptable task element</td>
<td>Green</td>
<td>Input matches decision statement</td>
</tr>
<tr>
<td>Right side of VM Powered On? decision element</td>
<td>startVM action element</td>
<td>Red dotted</td>
<td>Input does not match decision statement</td>
</tr>
<tr>
<td>Middle of startVM action element</td>
<td>vim3WaitTaskEnd action element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
<tr>
<td>Middle of vim3WaitTaskEnd action element</td>
<td>vim3WaitToolsStarted action element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
<tr>
<td>Middle of vim3WaitToolsStarted action element</td>
<td>OK scriptable task element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
<tr>
<td>Middle of Already Started scriptable task element</td>
<td>vim3WaitToolsStarted action element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
<tr>
<td>Right side of startVM action element</td>
<td>StartVM Failed scriptable task element</td>
<td>Thick red dotted</td>
<td>Exception handling</td>
</tr>
<tr>
<td>Right side of vim3WaitTaskEnd action element</td>
<td>Timeout 1 scriptable task element</td>
<td>Thick red dotted</td>
<td>Exception handling</td>
</tr>
<tr>
<td>Right side of vim3WaitToolsStarted action element</td>
<td>Timeout 2 scriptable task element</td>
<td>Thick red dotted</td>
<td>Exception handling</td>
</tr>
<tr>
<td>Middle of StartVM Failed scriptable task element</td>
<td>Send Email scriptable task element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
<tr>
<td>Middle of both Timeout scripted elements</td>
<td>Send Email scriptable task element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
<tr>
<td>Middle of OK scriptable task element</td>
<td>Send Email scriptable task element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
<tr>
<td>Right side of Send Email scriptable task element</td>
<td>Send Email Failed scriptable task element</td>
<td>Thick red dotted</td>
<td>Exception handling</td>
</tr>
<tr>
<td>Click</td>
<td>Link to</td>
<td>Type of Arrow</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Middle of Send Email scriptable task element</td>
<td>End element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
<tr>
<td>Middle of Send Email Failed scriptable task element</td>
<td>End element</td>
<td>Black</td>
<td>Normal workflow progression</td>
</tr>
</tbody>
</table>

4 Click Save at the bottom of the workflow workbench’s Schema tab.

You have linked the workflow elements together, defining the flow of the workflow.

What to do next

You can highlight different zones in the workflow.

Create Workflow Zones

You can emphasize different zones in workflow by adding workflow notes of different colors. Creating different workflow zones helps to make complicated workflow schema easier to read and understand.

Prerequisites

You must have created the Start VM and Send Email workflow, laid out its schema, and linked the schema elements together.

Procedure

1 Drag a workflow note element from the Generic menu into the workflow workbench.
2 Position the workflow note over the Already started scriptable task element.
3 Drag the edges of the workflow note to resize it so that it surrounds the Already started scriptable task element.
4 Double-click the text and add a description. For example, Path if virtual machine is already powered on.
5 Click Color in the Info tab at the bottom of the workflow workbench and select the background color.
6 Repeat the preceding steps to highlight other zones in the workflow.
   - Place a note over the vertical sequence of elements from the Is virtual machine on? decision element to the OK element. Add the description Start VM path.
   - Place a note over the startVM failed, both Timeout scriptable task elements and the Send Email Failed scriptable task element. Add the description Error handling.
   - Place a note over the Send Email scriptable task element.

Figure 2-3 shows what the example workflow schema should look like.
Define the Simple Workflow Example Decision Bindings

You bind a workflow’s elements together in the Schema tab of the workflow workbench. Decision bindings define how decision elements compare the input parameters received to the decision statement, and generate output parameters according to whether the input parameters match the decision statement.

Prerequisites
You must have created the Start VM and Send Email workflow, defined its parameters, laid out its schema, and linked the schema elements together.

Procedure
2. Click the Decision tab in the schema element properties pane at the bottom of the Schema tab.
3. Click the Not set (NULL) link and select vm as the decision element’s input parameter from the list of proposed parameters.
4. Select the state equals statement from the list of decision statements proposed in the drop-down menu.
   A Not set link appears in the value text box, which presents you with a limited choice of possible values.
5. Select poweredOn.
6. Click Save at the bottom of the workflow workbench’s Schema tab.

You have defined the true or false statement against which the decision element will compare the value of the input parameter it receives.

What to do next
You must define the bindings for the other elements in the workflow.
Bind the Simple Workflow Example Action Elements

You bind a workflow’s elements together in the Schema tab of the workflow workbench. Bindings define how the action elements process input parameters and generate output parameters.

Prerequisites

You must have created the Start VM and Send Email workflow, defined its parameters, laid out its schema, and linked the schema elements together.

Procedure

1. Click the startVM action element.
2. Set the following general information in the Info tab.
   - Interaction: No external interaction
   - Color: None
   - Business status: Check the check box and add the text Sending start VM.
   - Description: Leave the text Start / Resume a VM. Return the start task
3. Click the IN tab in the schema element properties pane at the bottom of the Schema tab.
   You will see the two possible input parameters available to the startVM action, vm and host.
   Orchestrator automatically binds the vm parameter to vm[in-parameter] because the startVM action can only take a VC:VirtualMachine as an input parameter. Orchestrator detected the vm parameter you defined when you set the workflow input parameters and so bound it to the action automatically.
4. Set host to NULL.
   This is an optional parameter, so you can set it to null. However, if you leave it set to Not set, the workflow will not validate.
5. Click the OUT tab in the schema element properties pane.
   The default output parameter that all actions generate, actionResult, appears.
6. Click the actionResult parameter’s Not set link.
7. Click the Create parameter/attribute in workflow link.
   The Parameter Information dialog box opens, where you can define the values for this output parameter.
   The output parameter type for the startVM action is a VC:Task object.
8. Name the parameter powerOnTask.
9. Provide a description for this parameter.
   For example, Contains the result of powering on a VM.
10. Click Create workflow ATTRIBUTE with the same name.
11. Click OK to exit the Parameter Information dialog box.
12 Repeat the preceding steps to bind the input and output parameters to the \texttt{vim3WaitTaskEnd} and \texttt{vim3WaitToolsStarted} action elements.

“Simple Workflow Example Action Element Bindings,” on page 66 lists the bindings for the \texttt{vim3WaitTaskEnd} and \texttt{vim3WaitToolsStarted} action elements.

13 Click \textbf{Save} at the bottom of the workflow workbench’s \textbf{Schema} tab.

The action elements’ input and output parameters are bound to the appropriate parameter types and values.

\textbf{What to do next}

Bind the scriptable task elements and define their functions.

\textbf{Simple Workflow Example Action Element Bindings}

Bindings define how the simple workflow example’s action elements process input and output parameters.

When defining bindings, Orchestrator presents parameters you have already defined in the workflow as candidates for binding. If you have not defined the required parameter in the workflow yet, the only parameter choice is \textbf{NULL}. Click \texttt{Create parameter/attribute in workflow} to create a new parameter.

\textbf{vim3WaitTaskEnd Action}

The \texttt{vim3WaitTaskEnd} action element declares constants to track the progress of a task and a polling rate. \textbf{Table 2-6} shows the input and output parameter bindings that the \texttt{vim3WaitTaskEnd} action requires.

\textbf{Table 2-6. Binding Values of the vim3WaitTaskEnd Action}

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| task           | IN           | Bind                                  | - Local Parameter: \texttt{task}  
|                |              |                                       | - Source parameter:  
|                |              |                                       | \texttt{task[attribute]}  
|                |              |                                       | - Type: \texttt{VC:Task}  
|                |              |                                       | - Description:  
|                |              |                                       | \texttt{The vCenter server task currently running.}  
| progress       | IN           | Create                                | - Local Parameter: \texttt{progress}  
|                |              |                                       | - Source parameter:  
|                |              |                                       | \texttt{progress[attribute]}  
|                |              |                                       | - Type: Boolean  
|                |              |                                       | - Value: No (false)  
|                |              |                                       | - Description:  
|                |              |                                       | \texttt{Log progress while waiting for the vCenter Server task to complete.}  

Table 2-6. Binding Values of the vim3WaitTaskEnd Action (Continued)

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| pollRate       | IN           | Create                               | Local Parameter: pollRate  
|                |              |                                      | Source parameter: pollRate[attribute]  
|                |              |                                      | Type: number  
|                |              |                                      | Value: 2  
|                |              |                                      | Description: The polling rate in seconds at which vim3WaitTaskEnd checks the advancement of the vCenter server task. |

| actionResult   | OUT          | Create                               | Local Parameter: actionResult  
|                |              |                                      | Source parameter: returnedManagedObject[attribute]  
|                |              |                                      | Type: Any  
|                |              |                                      | Description: The returned managed object from the waitTaskEnd action. |

vim3WaitToolsStarted Action

The vim3WaitToolsStarted action element waits until VMware Tools have installed on a virtual machine, and defines a polling rate and a timeout period. Table 2-7 shows the input parameter bindings the vim3WaitToolsStarted action requires.

The vim3WaitToolsStarted action element has no output, so requires no output binding.
### Table 2-7. Binding Values of the vim3WaitToolsStarted Action

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm</td>
<td>IN</td>
<td>Automatic binding</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local Parameter: vm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vm[in-parameter]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: VC:VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Value: Not editable, variable is not a workflow attribute.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The virtual machine to start.</td>
</tr>
<tr>
<td>pollingRate</td>
<td>IN</td>
<td>Bind</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local Parameter: pollRate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: pollRate[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The polling rate in seconds at which vim3WaitTaskEnd checks the advancement of the vCenter server task.</td>
</tr>
<tr>
<td>timeout</td>
<td>IN</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local Parameter: timeout</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: timeout[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Value: 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The timeout limit that vim3WaitToolsS started waits before throwing an exception.</td>
</tr>
</tbody>
</table>

### Bind the Simple Workflow Example Scripted Task Elements

You bind a workflow’s elements together in the Schema tab of the workflow workbench. Bindings define how the scripted task elements process input parameters and generate output parameters. You also bind the scriptable task elements to their JavaScript functions.

#### Prerequisites

You must have created the Start VM and Send Email workflow, defined its parameters, laid out its schema, and linked the schema elements.
Procedure

1. Click the **Already Started** scriptable task element.
2. Set the following general information in the **Info** tab.
   - **Interaction**: No external interaction
   - **Color**: None
   - **Business status**: Check the check box and add the text **VM already powered on**.
   - **Description**: Leave the text
     
     The VM is already powered on, bypassing startVM and waitTaskEnd, checking if the VM tools are up and running.

3. Click the **IN** tab in the schema element properties pane at the bottom of the **Schema** tab.
   Because this is a custom scriptable task element, no properties are predefined for you.
4. Right-click in the **IN** tab and select **Bind to workflow parameter/attribute**.
5. Select **vm** from the proposed list of parameters.
6. Leave the **OUT** and **Exception** tabs blank.
   This element does not generate an output parameter or exception.
7. Click the **Scripting** tab.
8. Add the following JavaScript function.

   ```
   // Writes the following event in the vCO database
   Server.log("VM "+vm.name +" already started");
   ```
9. Repeat the preceding steps to bind the remaining input parameters to the other scriptable task elements.
   “Simple Workflow Example Scriptable Task Element Bindings,” on page 69 lists the bindings for the **Start VM failed**, both **Timeout or Error**, and the **OK** scriptable task elements.
10. Click **Save** at the bottom of the workflow workbench’s **Schema** tab.

You have bound the scriptable task elements to their input and output parameters, and provided the scripting that defines their function.

**What to do next**

You must define the exception handling.

**Simple Workflow Example Scriptable Task Element Bindings**

Bindings define how the simple workflow example's scriptable task elements process input parameters. You also bind the scriptable task elements to their JavaScript functions.

When defining bindings, Orchestrator presents parameters you have already defined in the workflow as candidates for binding. If you have not defined the required parameter in the workflow yet, the only parameter choice is **NULL**. Click **Create parameter/attribute in workflow** to create a new parameter.

**Start VM Failed Scriptable Task**

The Start VM Failed scriptable task element handles any exceptions that the **startVM** action returns by setting the content of an email notification about the failure to start the virtual machine, and writing the event in the Orchestrator log.

Table 2-8 shows the input and output parameter bindings that the Start VM Failed scriptable task element requires.
Table 2-8. Bindings of the Start VM Failed Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorCode</td>
<td>IN</td>
<td>Create</td>
<td>Local Parameter: errorCode Source parameter: errorCode[attribute] Type: string Description: Catch any exceptions while powering on a VM.</td>
</tr>
<tr>
<td>body</td>
<td>OUT</td>
<td>Create</td>
<td>Local Parameter: body Source parameter: body[attribute] Type: string Description: The email body</td>
</tr>
</tbody>
</table>

The Start VM Failed scriptable task element performs the following scripted function.

body = "Unable to execute powerOnVM_Task() on VM '"+vm.name+"', exception found: "+errorCode; //Writes the following event in the vCO database Server.error("Unable to execute powerOnVM_Task() on VM '"+vm.name, "Exception found: "+errorCode);

Timeout 1 Scriptable Task Element

The Timeout 1 scriptable task element handles any exceptions that the vim3WaitTaskEnd action returns by setting the content of an email notification about the failure of the task, and writing the event in the Orchestrator log.

Table 2-9 shows the input and output parameter bindings that the Timeout 1 scriptable task element requires.
Table 2-9. Bindings of the Timeout 1 Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| vm             | IN           | Bind                                   | ■ Local Parameter: vm  
■ Source parameter: vm[in-parameter]  
■ Type: VC:VirtualMachine  
■ Description: The virtual machine to start. |
| errorCode      | IN           | Bind                                   | ■ Local Parameter: errorCode  
■ Source parameter: errorCode[attribute]  
■ Type: string  
■ Description: Catch any exceptions while powering on a VM. |
| body           | OUT          | Bind                                   | ■ Local Parameter: body  
■ Source parameter: body[attribute]  
■ Type: string  
■ Description: The email body |

The Timeout 1 scriptable task element requires the following scripted function.

```
body = "Error while waiting for poweredOnVM_Task() to complete on VM "'+vm.name+'", exception found: "'+errorCode+'";
//Writes the following event in the vCO database
Server.error("Error while waiting for poweredOnVM_Task() to complete on VM "'+vm.name, "Exception found: "+errorCode);
```

**Timeout 2 Scriptable Task Element**

The Timeout 2 scriptable task element handles any exceptions that the vim3WaitToolsStarted action returns by setting the content of an email notification about the failure of the task, and writing the event in the Orchestrator log.

**Table 2-10** shows the input and output parameter bindings that the Timeout 2 scriptable task element requires.
Table 2-10. Bindings of the Timeout 2 Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm</td>
<td>IN</td>
<td>Bind</td>
<td>Local Parameter: vm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vm[in-parameter]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: VC:VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The virtual machine to power on.</td>
</tr>
<tr>
<td>errorCode</td>
<td>IN</td>
<td>Bind</td>
<td>Local Parameter: errorCode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: errorCode[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: Catch any exceptions while powering on a VM.</td>
</tr>
<tr>
<td>body</td>
<td>OUT</td>
<td>Bind</td>
<td>Local Parameter: body</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: body[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The email body</td>
</tr>
</tbody>
</table>

The Timeout 2 scriptable task element requires the following scripted function.

```plaintext
body = "Error while waiting for VMware tools to be up on VM "'+vm.name+'", exception found: "+errorCode;
//Writes the following event in the vCO database
Server.error("Error while waiting for VMware tools to be up on VM "'+vm.name, ", "Exception found: "+errorCode);
```

**OK Scriptable Task Element**

The OK scriptable task element receives notice that the virtual machine has started successfully, sets the content of an email notification about the successful start of the virtual machine, and writes the event in the Orchestrator log.

*Table 2-11* shows the input and output parameter bindings that the OK scriptable task element requires.
Table 2-11. Bindings of the OK Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| vm             | IN           | Bind                                  | Local Parameter: vm  
Source parameter: vm[in-parameter]  
Type: VC:VirtualMachine  
Description: The virtual machine to power on. |
| body           | OUT          | Bind                                  | Local Parameter: body  
Source parameter: body[attribute]  
Type: string  
Description: The email body |

The OK scriptable task element requires the following scripted function.

```plaintext
body = "The VM \\
"+vm.name+"\" has started successfully and is ready for use"
//Writes the following event in the vCO database
Server.log(body);
```

**Send Email Failed Scriptable Task Element**

The Send Email Failed scriptable task element receives notice that the sending of the email failed, and writes the event in the Orchestrator log.

Table 2-12 shows the input and output parameter bindings that the OK scriptable task element requires.
Table 2-12. Bindings of the Send Email Failed Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm</td>
<td>IN</td>
<td>Bind</td>
<td>Local Parameter: vm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vm[in-parameter]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: VC:VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The virtual machine to power on.</td>
</tr>
<tr>
<td>toAddress</td>
<td>IN</td>
<td>Bind</td>
<td>Local Parameter: toAddress</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: toAddress[in-parameter]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The email address of the person to inform of the result of this workflow</td>
</tr>
<tr>
<td>emailErrorCode</td>
<td>IN</td>
<td>Create</td>
<td>Local Parameter: emailErrorCode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: emailErrorCode[attrbute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: Catch any exceptions while sending an email</td>
</tr>
</tbody>
</table>

The Send Email Failed scriptable task element requires the following scripted function.

```java
//Writes the following event in the vCO database
Server.error("Couldn't send result email to "+toAddress+" for VM "+vm.name, "Exception found: "+emailErrorCode);
```

Send Email Scriptable Task Element

The purpose of the Start VM and Send Email workflow is to inform an administrator when it starts a virtual machine. To do so, you must define the scriptable task that sends an email. To send the email, the Send Email scriptable task element needs an SMTP server, addresses for the sender and recipient of the email, the email subject, and the email content.

Table 2-13 shows the input and output parameter bindings that the Send Email scriptable task element requires.
Table 2-13. Bindings of the Send Email Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm</td>
<td>IN</td>
<td>Bind</td>
<td>Local Parameter: vm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description: The virtual machine to power on.</td>
<td></td>
</tr>
<tr>
<td>toAddress</td>
<td>IN</td>
<td>Bind</td>
<td>Local Parameter: toAddress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source parameter: toAddress[in-parameter]</td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description: The email address of the person to inform of the result of this workflow</td>
<td></td>
</tr>
<tr>
<td>body</td>
<td>IN</td>
<td>Bind</td>
<td>Local Parameter: body</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source parameter: body[attribute]</td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description: The email body</td>
<td></td>
</tr>
<tr>
<td>smtpHost</td>
<td>IN</td>
<td>Create</td>
<td>Local Parameter: smtpHost</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source parameter: smtpHost[attribute]</td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description: The email SMTP server</td>
<td></td>
</tr>
<tr>
<td>fromAddress</td>
<td>IN</td>
<td>Create</td>
<td>Local Parameter: fromAddress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source parameter: fromAddress[attribute]</td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description: The email address of the sender</td>
<td></td>
</tr>
<tr>
<td>subject</td>
<td>IN</td>
<td>Create</td>
<td>Local Parameter: subject</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source parameter: subject[attribute]</td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description: The email subject</td>
<td></td>
</tr>
</tbody>
</table>
The Send Email scriptable task element requires the following scripted function.

```javascript
//Create an instance of EmailMessage
var myEmailMessage = new EmailMessage();

//Apply methods on this instance that populate the email message
myEmailMessage.smtpHost = smtpHost;
myEmailMessage.fromAddress = fromAddress;
myEmailMessage.toAddress = toAddress;
myEmailMessage.subject = subject;
myEmailMessage.addMimePart(body, "text/html");

//Apply the method that sends the email message
myEmailMessage.sendMessage();
System.log("Sent email to "+toAddress);
```

Define the Simple Example Workflow Exception Bindings

You define exception bindings in the Schema tab in the workflow workbench. Exception bindings define how elements process errors.

The following elements in the workflow return exceptions: startVM, vim3WaitTaskEnd and vim3WaitToolsStarted.

Prerequisites

You must have created the Start VM and Send Email workflow, defined its parameters, and laid out its schema.

Procedure

1. Click the startVM action element.
2. Click the Exceptions tab at the bottom of the Schema tab.
3. Click the Not set link.
4. Select errorCode from the proposed list.
5. Repeat the preceding steps to set the exception binding to errorCode for both vim3WaitTaskEnd and vim3WaitToolsStarted.
6. Click the Send Email scriptable task element.
7. Click the Exceptions tab at the bottom of the Schema tab.
8. Click the Not set link.
9. Select emailErrorCode from the proposed list.
10. Click Save at the bottom of the workflow workbench’s Schema tab.

You have defined the exception binding for the elements that return exceptions.

What to do next

You must set the read and write properties on the attributes and parameters.

Set the Simple Workflow Example Attribute Read-Write Properties

You can define whether parameters and attributes are read-only constants or writeable variables. You can also set limitations on the values that users can provide for input parameters.

Setting certain parameters to read-only allows other developers to adapt the workflow or to modify it without breaking the workflow’s core function.
Prerequisites

You must have created a workflow, laid out and linked its schema, and defined the IN, OUT, and exception bindings for all elements.

Procedure

1. Click the General tab at the top of the workflow workbench.
   
   Under Attributes is a list of all the attributes you defined, with check boxes next to each attribute. When you check these checkboxes, you set attributes as read-only.

2. Check the check boxes to make the following attributes read-only constants:
   - progress
   - pollRate
   - timeout
   - smtpHost
   - fromAddress
   - subject

You have defined which of the workflow's attributes are constants and which are variables.

What to do next

You must set the parameter properties and place constraints on the possible values for that parameter.

Set the Simple Workflow Example Parameter Properties

You set the parameter properties in the Presentation tab in the workflow workbench. Setting the parameter properties affects the behavior of the parameter, and places constraints on the possible values for that parameter.

Prerequisites

You must have created a workflow, laid out and linked its schema, and defined the IN, OUT, and exception bindings for all elements.

Procedure

1. Click the Presentation tab.
   
   The two input parameters you defined for this workflow are listed.

2. Click the (VC:VirtualMachine)vm parameter.

3. Add a description in the Description tab in the bottom half of the screen.
   
   For example, type The virtual machine to start.

4. Click the Properties tab in the bottom half of the screen.
   
   This tab allows you to set the properties of the (VC:VirtualMachine)vm parameter.

5. Right-click the Properties tab and select Add Property.

6. Select Mandatory input from the list of proposed properties.
   
   When you select this property, users cannot run the Start VM and Send Email workflow without providing a virtual machine to start.

7. Set the value of the Mandatory input property to Yes.

8. Right-click on the Properties tab and select Add Property again.
Select **Select value** as from the list of proposed properties. When you set this property, you set how the user selects the value of the `(VC:VirtualMachine)vm` input parameter.

Select **list** from the list of possible values.

Click the `(string)toAddress` parameter in the top half of the **Presentation** tab.

Add a description in the **Description** tab in the bottom half of the screen. For example, type **The email address of the person to notify.**

Click the **Properties** tab for `(string)toAddress`.

Right-click the **Properties** tab and select **Add Property > Mandatory input**.

Set the value of the **Mandatory input** property to **Yes**.

Right-click the **Properties** tab and select **Add Property > Matching regular expression**.

This property allows you to set constraints on what users can provide as input.

Click the **Value** text box for **Matching regular expression** and set the constraints to

```
[a-zA-Z0-9-_.]+@[a-zA-Z0-9-_.]+\.[a-zA-Z]{2,4}
```

Setting these constraints limits user input to characters that are appropriate for email addresses. If the user tries to input any other character for the email address of the recipient when they start the workflow, the workflow will not start.

You have made both parameters mandatory, defined how the user can select the virtual machine to start, and limited the characters that can be input for the recipient's email address.

**What to do next**

You must create the layout, or presentation, of the input parameters dialog box in which users enter a workflow's input parameter values when they run it.

**Set the Layout of the Simple Workflow Example Input Parameters Dialog Box**

You create the layout, or presentation, of the input parameters dialog box in the **Presentation** tab of the workflow workbench. The input parameters dialog box opens when users run a workflow, and is the means by which users enter the input parameters with which the workflow runs.

The layout you define in the **Presentation** tab also defines the layout of the input parameter dialog boxes for workflows you run using a Web view.

**Prerequisites**

You must have created a workflow, laid out and linked its schema, defined the **IN**, **OUT**, and exception bindings for all elements, and set the attribute and parameter properties.

**Procedure**

1. Click the **Presentation** tab in the workflow workbench.
2. Right-click the **Presentation** node in the presentation hierarchical list and select **New Group**.
3. A **New Step** node and a **New Group** sub-node appear under the **Presentation** node.
4. Right-click **New Step** and select **Delete**.

Because this workflow only has two parameters, you do not need multiple layers of display sections in the input parameters dialog box.
Double-click **New Group** to edit the group name and press Enter.

For example, name the display group **Virtual Machine**.

The text you enter here appears as a heading in the input parameter dialog box when users start the workflow.

Provide a description of the Virtual Machine display group in the **Description** text box in the **General** tab at the bottom of the **Presentation** tab.

For example, type **Select the virtual machine to start.**

The text you enter here appears as a prompt in the input parameter dialog box when users start the workflow.

Drag the (VC:VirtualMachine)vm parameter under the **Virtual Machine** display group.

The text box in the input parameters dialog box in which the user enters the virtual machine to start will appear under the heading **Virtual Machine**.

Repeat the preceding steps to create a display group for the **toAddress** parameter, setting the following properties:

a. Create a display group named **Recipient's Email Address**.

b. Add a description for the display group, for example, **Enter the email address of the person to notify when this virtual machines is powered-on.**

c. Drag the **toAddress** property under the **Recipient's Email Address** display group.

You have set up the layout of the input parameters dialog box that appears when users run the workflow.

**What to do next**

You have completed the development of the simple workflow example. You can now validate and run the workflow.

**Validate and Run the Simple Workflow Example**

After you create a workflow, you can validate it to discover any possible errors. If the workflow contains no errors, you can run it.

**Prerequisites**

You must have created a workflow, laid out its schema, defined the links and bindings, defined the parameter properties, and created the presentation of the input parameters dialog box before you attempt to validate and run the workflow.

**Procedure**

1. Click **Validation** in the **Schema** tab of the workflow workbench.

   The validation tool locates any errors in the definition of the workflow.

2. When you have eliminated any errors, click **Save and Close** at the bottom of the workflow workbench.

   You return to the Orchestrator client.

3. Click the **Workflows** view.

4. Select **Workflow Examples > Start VM and Send Email** in the workflow hierarchical list.

5. Right-click the Start VM and Send Email workflow and select **Execute workflow**.

   The input parameters dialog box opens and prompts you for a virtual machine to start and an email address of a person to inform.
6 Select a virtual machine to start in the vCenter Server from the proposed list.
7 Enter an email address to which to send email notifications.
8 Click **Submit** to start the workflow.
   A workflow token appears under the Start VM and Send Email workflow.
9 Click the workflow token to follow the progress of the workflow as it runs.
If the workflow ran successfully the virtual machine you identified is in the powered-on state, and the email recipient you defined receives a confirmation email.

**Develop a Complex Workflow**

Developing a complex example workflow demonstrates the most common steps in the workflow development process, as well as more advanced scenarios, such as creating custom decisions and loops.

In this exercise, you develop a workflow that takes a snapshot of all the virtual machines contained in a given resource pool. The workflow you create will perform the following tasks:

1 Prompts the user for a resource pool that contains the virtual machines of which to take snapshots.
2 Determines whether the resource pool contains running virtual machines.
3 Determines how many running virtual machines the resource contains.
4 Verifies whether an individual virtual machine running in the pool meets specific criteria for a snapshot to be taken.
5 Takes the snapshot of the virtual machine.
6 Determines whether more virtual machines exist in the pool of which to take snapshots.
7 Repeats the verification and snapshot process until the workflow has taken snapshots of all eligible virtual machines in the resource pool.

**Prerequisites**

Before attempting to develop this complex workflow, follow the exercises in “**Develop a Simple Example Workflow,**” on page 57. The procedures in this section provide the broad steps of the development process, but are not as detailed as the simple workflow exercises.

**Procedure**

1 **Create the Complex Workflow Example** on page 81
   In this exercise you create a workflow called Take a Snapshot of All Virtual Machines in a Resource Pool.
2 **Define the Complex Workflow Example Input Parameters** on page 82
   You define workflow input parameters in the workflow workbench. The input parameters provide data for the workflow to process.
3 **Create a Custom Action For the Complex Workflow Example** on page 82
   The Check VM scriptable element calls on an actions that does not exist in the Orchestrator API. You must create the **getVMDiskModes** action.
4 **Create the Complex Workflow Example Schema** on page 83
   You create a workflow’s schema in the **Schema** tab of the workflow workbench. The workflow schema contains the elements that the workflow runs.
5 **Link the Complex Workflow Example Schema Elements** on page 84
   You link a workflow’s elements together in the **Schema** tab of the workflow workbench. The linking defines the logical flow the workflow.
6. **Create the Complex Workflow Example Zones** on page 85

Optionally, you can highlight different zones of the workflow by adding workflow notes. Creating different workflow zones helps to make complicated workflow schema easier to read and understand.

7. **Define the Complex Workflow Example Bindings** on page 86

You bind a workflow’s elements together in the **Schema** tab of the workflow workbench. Bindings define how the data flow of the workflow. You also bind the scriptable task elements to their JavaScript functions.

8. **Set the Complex Workflow Example Attribute Properties** on page 97

You set the attribute properties in the **General** tab in the workflow workbench.

9. **Create the Layout of the Complex Workflow Example Input Parameters** on page 98

You create the layout, or presentation, of the input parameters dialog box in the **Presentation** tab of the workflow workbench. The input parameters dialog box opens when users run a workflow, and is the means by which users enter the input parameters with which the workflow runs.

10. **Validate and Run the Complex Workflow Example** on page 98

After you create a workflow, you can validate it to discover any possible errors. If the workflow contains no errors, you can run it.

---

### Create the Complex Workflow Example

In this exercise you create a workflow called Take a Snapshot of All Virtual Machines in a Resource Pool.

**Prerequisites**

To create this more complex workflow example, you must have the following components installed and configured on the system.

- Orchestrator 4.0
- vCenter 4.0, controlling a resource pool that contains some virtual machines
- The **Workflow Examples** category in the workflows hierarchical list, that you created in “Create the Simple Workflow Example,” on page 59.

For details about how to install and configure vCenter, see the ESX and vCenter Server Installation Guide. For details about how to configure Orchestrator, see the Orchestrator 4.0 Administration Guide.

**Procedure**

1. Select **Workflows > Workflow Examples**.
2. Create a workflow called Take a Snapshot of All Virtual Machines in a Resource Pool.
3. Open the workflow workbench by right-clicking the new workflow and selecting **Edit**.
4. In the **General** tab of the workflow workbench, click the version number digits to increment the version number.
   
   Because this is the initial creation of the workflow, set the version to **0.0.1**.
5. Check the **Allowed operations** check boxes in the **General** tab to set the operations the users can perform on the workflow.
6. Click the **Server restart behavior** value in the **General** tab to set whether the workflow resumes after a server restart.
7. Provide a description of what the workflow does in the **Description** text box in the **General** tab.
8. Click **Save** at the bottom of the **General** tab.

You have created the Take a Snapshot of All Virtual Machines in a Resource Pool workflow.
What to do next
You can now continue editing the Take a Snapshot of All Virtual Machines in a Resource Pool workflow.

Define the Complex Workflow Example Input Parameters
You define workflow input parameters in the workflow workbench. The input parameters provide data for the workflow to process.

Prerequisites
You must have created the Take a Snapshot of All Virtual Machines in a Resource Pool workflow, and opened it for editing in the workflow workbench.

Procedure
1. Click the Inputs tab in the workflow workbench.
2. Define the following input parameter.
   - Name: resourcePool
   - Type: VC:ResourcePool
   - Description: The resource pool containing the virtual machines of which to take snapshots.
3. Click the Outputs tab in the workflow workbench.
4. Define the following input parameter.
   - Name: snapshotVmArrayOut
   - Type: Array/VC:ResourcePool
   - Description: The Array of virtual machines of which snapshots have been taken.

You have defined the workflow input parameter.

What to do next
You can create a workflow schema.

Create a Custom Action For the Complex Workflow Example
The Check VM scriptable element calls on an actions that does not exist in the Orchestrator API. You must create the getVMDiskModes action.

For more detail about creating actions, see Chapter 3, “Developing Actions,” on page 101.

Prerequisites
You must have created the Take a Snapshot of All Virtual Machines in a Resource Pool workflow.

Procedure
1. Close the workflow workbench by clicking Save and Close.
2. Click the Actions view in the Orchestrator client.
3. Right-click the root of the actions hierarchical list and select New Module.
4. Name the new module com.vmware.example.
5. Right-click the com.vmware.example module and select Add Action.
6. Create an action called getVMDiskModes.
7. Right-click getVMDiskModes and select Edit.
8 Increment the version number in the General tab in the actions workbench by clicking the version digits.

9 Check all the Allowed Operations check boxes in the General tab.

10 Add the following description of the action in the General tab.

This action returns an array containing the disk modes of all disks on a VM. The array contains the following string values:
- persistent
- independent-persistent
- nonpersistent
- independent-nonpersistent

Legacy values:
- undoable
- append

11 Click the Scripting tab.

12 Right-click in the top pane of the Scripting tab and select Add Parameter to create the following input parameter.

- Name: vm
- Value: VC:VirtualMachine
- Description: The virtual machine for which to return the Disk Modes

13 Add the following scripting in the bottom of the Scripting tab.

The following code returns an array of disk modes for the disks of the virtual machine.

```javascript
var devicesArray = vm.config.hardware.device;
var retArray = new Array();
if (devicesArray!=null && devicesArray.length!=0) {
    for (i in devicesArray) {
        if (devicesArray[i] instanceof VcVirtualDisk) {
            retArray.push(devicesArray[i].backing.diskMode);
        }
    }
}
return retArray;
```

14 Click Save and Close to exit the Actions palette.

You have defined the custom action the Take a Snapshot of All Virtual Machines in a Resource Pool workflow requires.

What to do next
You must create the workflow schema.

Create the Complex Workflow Example Schema
You create a workflow’s schema in the Schema tab of the workflow workbench. The workflow schema contains the elements that the workflow runs.

Prerequisites
You must have created the Take a Snapshot of All Virtual Machines in a Resource Pool workflow, defined its input parameter, and created the getVMDiskModes action.
Procedure

1. Right-click the Take a Snapshot of All Virtual Machines in a Resource Pool workflow to open the workflow workbench.
2. Click the Schema tab in the workflow workbench.
3. Add the following schema elements to the workflow schema.

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Element Name</th>
<th>Position in Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scriptable task</td>
<td>Initializing</td>
<td>Below start element</td>
</tr>
<tr>
<td>Decision</td>
<td>VMs to Process?</td>
<td>Below Initializing scriptable task</td>
</tr>
<tr>
<td>Scriptable task</td>
<td>Pool Has No VMs</td>
<td>Below Virtual Machines to Process? custom decision</td>
</tr>
<tr>
<td>Custom decision</td>
<td>Remaining VMs?</td>
<td>Right of Virtual Machines to Process? custom decision</td>
</tr>
<tr>
<td>Action</td>
<td>getVMDiskModes</td>
<td>Right of Virtual Machines Remaining? custom decision</td>
</tr>
<tr>
<td>Custom decision</td>
<td>Create snapshot?</td>
<td>Right of getVMDiskModes action</td>
</tr>
<tr>
<td>Workflow</td>
<td>Create a Snapshot (vCenter Server 4.0)</td>
<td>Above Create snapshot? decision</td>
</tr>
<tr>
<td>Scriptable task</td>
<td>VM Snapshots</td>
<td>Left of Create a Snapshot workflow</td>
</tr>
<tr>
<td>Scriptable task</td>
<td>Increment</td>
<td>Left of VM Snapshots scriptable task</td>
</tr>
<tr>
<td>Scriptable task</td>
<td>Log Exception</td>
<td>Above VM Snapshots scriptable task</td>
</tr>
<tr>
<td>Scriptable task</td>
<td>Set Output</td>
<td>Below Remaining VMs? custom decision</td>
</tr>
<tr>
<td>End element</td>
<td>No name</td>
<td>Right of Set Output scriptable task</td>
</tr>
</tbody>
</table>

4. Click Save at the bottom of the Schema tab.

You have created the structure of the workflow.

What to do next

You can now link the workflow elements together.

Link the Complex Workflow Example Schema Elements

You link a workflow’s elements together in the Schema tab of the workflow workbench. The linking defines the logical flow the workflow.

Prerequisites

You must have created the Take a Snapshot of All Virtual Machines in a Resource Pool workflow, defined its input parameter, and created its schema.

Procedure

1. Click the connector tool button in the toolbar at the top of the Schema tab in the workflow workbench.
2. Create the following links between the elements in the schema.

<table>
<thead>
<tr>
<th>Starting Element</th>
<th>Target Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start element</td>
<td>Initializing scriptable element</td>
</tr>
<tr>
<td>Initializing scriptable element</td>
<td>VMs to Process? custom decision</td>
</tr>
</tbody>
</table>
Starting Element | Target Element
---|---
VMs to Process? decision’s true result | VMs Remaining? custom decision
VMs to Process? decision’s false result | Has No VMs scriptable task
Has No VMs scriptable task | Set Output scriptable task
VMs Remaining? custom decision’s true result | getVMDisksModes action
VMs Remaining? custom decision’s false result | Set Output scriptable task
getVMDisksModes action | Create Snapshot? decision
getVMDisksModes action exception link | Log Exception scriptable task
Create Snapshot? custom decision’s true result | Create a Snapshot workflow
Create Snapshot? custom decision’s false result | Increment scriptable task
Create a Snapshot workflow | VM Snapshots scriptable task
Create a Snapshot workflow exception link | Log Exception scriptable task
VM Snapshots scriptable task | Increment scriptable task
Increment scriptable task | VMs Remaining? custom decision
Log Exception scriptable task | Increment scriptable task
Set Output scriptable task | End element

3. Click Save at the bottom of the workflow workbench’s Schema tab.

You have linked the workflow elements, defining the logical flow of the workflow.

**What to do next**

You can optionally define workflow zones by using workflow notes.

**Create the Complex Workflow Example Zones**

Optionally, you can highlight different zones of the workflow by adding workflow notes. Creating different workflow zones helps to make complicated workflow schema easier to read and understand.

**Prerequisites**

You must have created the workflow, created its schema, and linked the schema elements.

**Procedure**

1. Create the following workflow zones by using workflow notes.

<table>
<thead>
<tr>
<th>Elements in Zone</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start element; Initialize scriptable task; VMs to Process? custom decision</td>
<td>Get an array of virtual machines from a resource pool, initialize the counter of the Array and set the first virtual machine to be treated, if any.</td>
</tr>
<tr>
<td>Pool has no VMs scriptable task.</td>
<td>Resource pool contains no virtual machines of which to take snapshots.</td>
</tr>
<tr>
<td>Elements in Zone</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VMs remaining? custom decision; <code>getVMDiskModes</code> action, Create Snapshot? decision; Create a Snapshot workflow; VM Snapshots scriptable task; Increment scriptable task; Log Exception scriptable task</td>
<td>Check whether any virtual machines remain in the resource pool, check that a virtual machine meets the snapshot criteria, take a snapshot, then loop until a snapshot has been taken of all the virtual machines.</td>
</tr>
<tr>
<td>Set Output scriptable task; End element</td>
<td>Generates the resulting array of virtual machines of which snapshots have been taken.</td>
</tr>
</tbody>
</table>

2. Click **Save** at the bottom of the workflow workbench’s **Schema** tab.

Your workflow schema should look like the following diagram.

**Figure 2-4. Schema Diagram for Take Snapshot of all Virtual Machines in a Resource Pool Example Workflow**

---

**What to do next**

You must define the bindings between the element parameters.

**Define the Complex Workflow Example Bindings**

You bind a workflow’s elements together in the **Schema** tab of the workflow workbench. Bindings define how the data flow of the workflow. You also bind the scriptable task elements to their JavaScript functions.

**Prerequisites**

You must already have created the Take a Snapshot of All Virtual Machines in a Resource Pool workflow, defined its input parameter, created its schema, and linked the schema elements together.

**Procedure**

1. Click the **Schema** tab in the workflow workbench.
2. Define the bindings shown in “Complex Workflow Example Bindings,” on page 87.
3. Click **Save** at the bottom of the workflow workbench’s **Schema** tab.

All the elements’ input and output parameters are bound to the appropriate parameter types and values.

**What to do next**

You must set the attribute properties.
Complex Workflow Example Bindings

Bindings define how the simple workflow example's action elements process input and output parameters.

The Take Snapshots of All Virtual Machines in a Resource Pool workflow requires the following input and output parameter bindings. You also define the JavaScript functions for the scriptable task elements.

Initializing Scriptable Task

The Initializing scriptable task element initializes the attributes of the workflow. Table 2-14 shows the input and output parameter bindings that the Initializing scriptable task element requires.

Table 2-14. Bindings of the Initializing Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>resourcePool</td>
<td>IN</td>
<td>Bind</td>
<td>Local parameter: resourcePool</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: resourcePool[in-parameter]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: VC:ResourcePool</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The resource pool containing the virtual machines of which to take snapshots</td>
</tr>
<tr>
<td>allVMs</td>
<td>OUT</td>
<td>Create</td>
<td>Local parameter: allVMs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: allVMs[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: Array/VC:VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The virtual machines in the resource pool.</td>
</tr>
<tr>
<td>numberOfVMs</td>
<td>OUT</td>
<td>Create</td>
<td>Local parameter: numberOfVMs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: numberOfVMs[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The number of virtual machines found in the resourcePool</td>
</tr>
<tr>
<td>vmCounter</td>
<td>OUT</td>
<td>Create</td>
<td>Local parameter: vmCounter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vmCounter[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The counter of the virtual machines inside the array</td>
</tr>
</tbody>
</table>
### Table 2-14. Bindings of the Initializing Scriptable Task Element (Continued)

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm</td>
<td>OUT</td>
<td>Create</td>
<td>▪ Local parameter: vm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▪ Source parameter: vm[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▪ Type: VC:VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▪ Description: The current virtual machine having a snapshot taken</td>
</tr>
<tr>
<td>snapshotVmArray</td>
<td>OUT</td>
<td>Create</td>
<td>▪ Local parameter: snapshotVmArray</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▪ Source parameter: snapshotVmArray[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▪ Type: Array/VC:VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▪ Description: The Array of virtual machines of which snapshots have been taken</td>
</tr>
</tbody>
</table>

The Initialize scriptable task element performs the following scripted function.

```java
//Retrieve an array of virtual machines contained in the specified Resource Pool
allVMs = resourcePool.vm;
//Initialize the size of the Array and the first VM to snapshot
if (allVMs!=null & allVMs.length!=0) {
    numberOfVms = allVMs.length;
    vm = allVMs[0];
} else {
    numberOfVms = 0;
}
//Initialize the VM counter
vmCounter = 0;
//Initializing the array of VM snapshots
snapshotVmArray = new Array();
```

### VMs to Process? Decision Element

The VMs to Process? decision element determines whether any virtual machines of which to take snapshots exist in the resource pool. Table 2-15 shows the bindings that the VMs to Process? decision element requires.
Table 2-15. Bindings of the VMs to Process? Decision Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>numberOfVMs</td>
<td>Decision</td>
<td>Bind</td>
<td>Source parameter: numberOfVMs[attribute ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decision statement: Greater than</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Value: 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The number of virtual machines found in the resourcePool</td>
</tr>
</tbody>
</table>

Pool Has No VMs Scriptable Task Element

The Pool Has No VMs scriptable task element logs the fact that the resource pool contains no eligible virtual machines in the Orchestrator database. Table 2-16 shows the bindings that the Pool Has No VMs scriptable task element requires.

Table 2-16. Bindings of the Pool Has No VMs Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>resourcePool</td>
<td>IN</td>
<td>Bind</td>
<td>Local parameter: resourcePool</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: resourcePool[in-parameter]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: VC:ResourcePool</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The resource pool containing the virtual machines of which to take snapshots.</td>
</tr>
</tbody>
</table>

The Pool Has No VMs scriptable task element performs the following scripted function.

//Writes the following event in the vCO database
Server.warn("The specified ResourcePool " +resourcePool.name+" does not contain any VMs.");

Remaining VMs? Custom Decision Element

The Remaining VMs? custom decision element determines whether any virtual machines of which to take snapshots remain in the resource pool. Table 2-17 shows the bindings that the Remaining VMs? custom decision element requires.
### Table 2-17. Bindings of the Remaining VMs? Custom Decision Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>numberOfVMs</td>
<td>IN</td>
<td>Bind</td>
<td>Source parameter: numberOfVMs[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decision statement: Greater than</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Value: 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The number of virtual machines found in the resourcePool</td>
</tr>
<tr>
<td>vmCounter</td>
<td>IN</td>
<td>Bind</td>
<td>Local parameter: vmCounter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vmCounter[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: The counter of the virtual machines inside the array</td>
</tr>
</tbody>
</table>

The Remaining VMs? custom decision element performs the following scripted function.

```java
//Checks if the workflow has reached the end of the array of VMs
if (vmCounter < numberOfVMs) {
    return true;
} else {
    return false;
}
```

**getVMDisksModes Action Element**

The `getVMDisksModes` action element obtains the modes of the disks running in a virtual machine. 

*Table 2-18* shows the bindings that the `getVMDisksModes` action element requires.
Table 2-18. Bindings of the getVMDisksModes Action Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm</td>
<td>IN</td>
<td>Bind</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local parameter: vm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vm[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: VC:VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: <em>The current virtual machine having a snapshot taken</em></td>
</tr>
<tr>
<td>actionResult</td>
<td>OUT</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local parameter: actionResult</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vmDisksModes[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: Array/String</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: <em>The current Disks Modes of the virtual machine</em></td>
</tr>
<tr>
<td>errorCode</td>
<td>Exception</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local parameter: errorCode</td>
</tr>
</tbody>
</table>

Create Snapshot? Custom Decision Element

The Create Snapshot? custom decision element determines whether to take snapshots of virtual machines, depending on the disk modes of the virtual machines. Table 2-19 shows the bindings that the Create Snapshot? custom decision element requires.

Table 2-19. Bindings of the Create Snapshot? Decision Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmDisksMode</td>
<td>IN</td>
<td>Bind</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local parameter: vmDisksMode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vmDisksMode[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: Array/String</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: <em>The current Disks Modes of the virtual machine</em></td>
</tr>
<tr>
<td>vm</td>
<td>IN</td>
<td>Bind</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local parameter: vm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source parameter: vm[attribute]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: VC:VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description: <em>The current virtual machine having a snapshot taken</em></td>
</tr>
</tbody>
</table>
The Create Snapshot? custom decision element performs the following scripted function.

// A snapshot cannot be taken if one of its disks is in independent mode
// (independent-persistent or independent-nonpersistent)
var containsIndependentDisks = false;
if (vmDisksModes!=null & & vmDisksModes.length>0) {
    for (i in vmDisksModes) {
        if (vmDisksModes[i].charAt(0)=="i") {
            containsIndependentDisks = true;
        }
    }
} else {
    // if no disk found no need to try to snapshot the VM
    System.warn("Won't snapshot '"+vm.name+"', no disks found");
    return false;
}
if (containsIndependentDisks) {
    System.warn("Won't snapshot '"+vm.name+"', independent disk(s) found");
    return false;
} else {
    System.log("Snapshoting '"+vm.name+"'");
    return true;
}

Create Snapshot Workflow Element

The Create Snapshot workflow element takes snapshots of virtual machines. Table 2-20 shows the bindings that the Create Snapshot workflow element requires.

Table 2-20. Bindings of the Create Snapshot Workflow Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| vm             | IN           | Bind                                  | ■ Local parameter: vm
|                |              |                                       | ■ Source parameter: ActiveVM[attribute]
|                |              |                                       | ■ Type: VC:VirtualMachine
| name           | IN           | Create                                | ■ Local parameter: name
|                |              |                                       | ■ Source parameter: snapshotName[attribute]
|                |              |                                       | ■ Type: string
|                |              |                                       | ■ Description: The name for this snapshot. The name does not need to be unique for this virtual machine.
### Table 2-20. Bindings of the Create Snapshot Workflow Element (Continued)

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| description    | IN           | Create                                | - Local parameter: description  
|                |              |                                       | - Source parameter: snapshotDescription[attribute]  
|                |              |                                       | - Type: string  
|                |              |                                       | - Description: A description for this snapshot. |
| memory         | IN           | Create                                | - Local parameter: memory  
|                |              |                                       | - Source parameter: snapshotMemory[attribute]  
|                |              |                                       | - Type: Boolean  
|                |              |                                       | - Value: no  
|                |              |                                       | - Description: If TRUE, a dump of the internal state of the virtual machine (a memory dump) is included in the snapshot. |
| quiesce        | IN           | Create                                | - Local parameter: quiesce  
|                |              |                                       | - Source parameter: snapshotQuiesce[attribute]  
|                |              |                                       | - Type: Boolean  
|                |              |                                       | - Value: yes  
|                |              |                                       | - Description: If TRUE and the virtual machine is powered on when the snapshot is taken, the VMware Tools are used to quiesce the file system in the virtual machine. |
| snapshot       | OUT          | Create                                | - Local parameter: snapshot  
|                |              |                                       | - Source parameter: NULL  
|                |              |                                       | - Type: VC:VirtualMachineSnapshot  
|                |              |                                       | - Description: The snapshot taken. |
| errorCode      | Exception    | Create                                | - Local parameter: errorCode |

### VM Snapshots Scriptable Task Element

The VM Snapshots scriptable task element adds the snapshots to an array. Table 2-21 shows the bindings that the VM Snapshots scriptable task element requires.
Table 2-21. Bindings of the VM Snapshots Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| vm               | IN           | Bind                                   | - Local parameter: vm  
|                  |              |                                        | - Source parameter: ActiveVM[attribute]                                                      |
|                  |              |                                        | - Type: VC:VirtualMachine                                                                  |
|                  |              |                                        | - Description: An active virtual machine of which to take a snapshot.                                  |
| snapshotVmArray  | IN           | Bind                                   | - Local parameter: snapshotVmArray  
|                  |              |                                        | - Source parameter: snapshotVmArray[attri bute]                                                      |
|                  |              |                                        | - Type: Array/VC:VirtualMachine                                                                  |
|                  |              |                                        | - Description: The Array of virtual machines of which snapshots have been taken                              |
| snapshotVmArray  | OUT          | Bind                                   | - Local parameter: snapshotVmArray  
|                  |              |                                        | - Source parameter: snapshotVmArray[attri bute]                                                      |
|                  |              |                                        | - Type: Array/VC:VirtualMachine                                                                  |
|                  |              |                                        | - Description: The Array of virtual machines of which snapshots have been taken                              |

The VM Snapshots scriptable task element performs the following scripted function.

// Writes the following event in the vCO database  
Server.log("Successfully took snapshot of the VM "+vm.name);  
// Inserts the VM snapshot in an array  
snapshotVmArray.push(vm);

**Increment Scriptable Task Element**

The Increment scriptable task element increments the counter that counts the number of virtual machines in the array. Table 2-22 shows the bindings that the Increment scriptable task element requires.
### Table 2-22. Bindings of the Increment Scriptable Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| vmCounter      | IN           | Bind                                   | - Local parameter: vmCounter  
|                |              |                                         | - Source parameter: vmCounter[attribute]  
|                |              |                                         | - Type: number  
|                |              |                                         | - Description: The counter of the virtual machines inside the array |
| allVMs         | IN           | Bind                                   | - Local parameter: allVMs  
|                |              |                                         | - Source parameter: allVMs[attribute]  
|                |              |                                         | - Type: Array/VC:VirtualMachine  
|                |              |                                         | - Description: The virtual machines in the resource pool. |
| vmCounter      | OUT          | Bind                                   | - Local parameter: vmCounter  
|                |              |                                         | - Source parameter: vmCounter[attribute]  
|                |              |                                         | - Type: number  
|                |              |                                         | - Description: The counter of the virtual machines inside the array |
| vm             | OUT          | Bind                                   | - Local parameter: vm  
|                |              |                                         | - Source parameter: vm[attribute]  
|                |              |                                         | - Type: VC:VirtualMachine  
|                |              |                                         | - Description: The current virtual machine having a snapshot taken |

The Increment scriptable task element performs the following scripted function.

```java
// Increases the array VM counter
vmCounter++;  
// Sets the next VM to be snapshot in the attribute vm
vm = allVMs[vmCounter];
```

### Log Exception Scriptable Task Element

The Log Exception scriptable task element handles exceptions from the workflow and action elements. Table 2-23 shows the bindings that the Log Exception scriptable task element requires.
Table 2-23. Bindings of the Log Exception Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| vm             | IN           | Bind                                   | - Local parameter: vm  
|                |              |                                        | - Source parameter: vm[attribute]  
|                |              |                                        | - Type: VC:VirtualMachine  
|                |              |                                        | - Description: The current virtual machine having a snapshot taken |
| errorCode      | IN           | Bind                                   | - Local parameter: errorCode  
|                |              |                                        | - Source parameter: errorCode[attribute]  
|                |              |                                        | - Type: string  
|                |              |                                        | - Description: An exception caught while taking a snapshot of a virtual machine |

The Log Exception scriptable task element performs the following scripted function.

//Writes the following event in the vCO database
Server.error("Couldn't snapshot the VM '"+vm.name+'', exception: "+errorCode);

Set Output Scriptable Task Element

The Set Output scriptable generates the workflow’s output parameter, that contains the array of virtual machines of which snapshots have been taken. Table 2-24 shows the bindings that the Set Output scriptable task element requires.
### Table 2-24. Bindings of the Set Output Task Element

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Binding Type</th>
<th>Bind to Existing or Create Parameter?</th>
<th>Binding Values</th>
</tr>
</thead>
</table>
| snapshotVmArray       | IN           | Bind                                   | - Local parameter: snapshotVmArray  
- Source parameter: snapshotVmArray[attribute]  
- Type: Array/VC:VirtualMachine  
- Description: The Array of virtual machines of which snapshots have been taken |
| snapshotVmArrayOut    | OUT          | Bind                                   | - Local parameter: snapshotVmArrayOut  
- Source parameter: snapshotVmArrayOut[out-parameter]  
- Type: Array/VC:VirtualMachine  
- Description: The Array of virtual machines of which snapshots have been taken |

The Set Output scriptable task element performs the following scripted function.

```java
//Passes the value of the internal attribute to a workflow output parameter
snapshotVmArrayOut = snapshotVmArray;
```

### Set the Complex Workflow Example Attribute Properties

You set the attribute properties in the General tab in the workflow workbench.

#### Prerequisites

You must have created the workflow, created and linked its schema, and defined the IN and OUT bindings for all elements.

#### Procedure

1. Click the **General** tab.
2. Check the read-only check box of the following attributes to make them read-only constants:
   - snapshotName
   - snapshotDescription
   - snapshotMemory
   - snapshotQuiesce

You have defined which of the workflow's attributes are constants and which are variables.

#### What to do next

You must create the workflow presentation, which creates the layout of the input parameters dialog box in which users enter a workflow's input parameter values when they run it.
Create the Layout of the Complex Workflow Example Input Parameters

You create the layout, or presentation, of the input parameters dialog box in the Presentation tab of the workflow workbench. The input parameters dialog box opens when users run a workflow, and is the means by which users enter the input parameters with which the workflow runs.

Prerequisites

You must have created the workflow, created and linked its schema, defined the IN, OUT, and exception bindings for all elements, and set the attribute and parameter properties.

Procedure

1. Click the Presentation tab in the workflow workbench.
   This workflow has only one input parameter, so creating the presentation is straightforward.

2. Right-click the Presentation node in the presentation hierarchical list and select New Group.

3. Delete the New Step that appears above the New Group.

4. Double-click the New Group and change the group name to Resource Pool.

5. Provide a description of the Resource Pool display group in the Description text box in the General tab at the bottom of the Presentation tab.
   For example, Enter the name of the resource pool that contains the virtual machines of which to take a snapshot.

6. Click the (VC:ResourcePool)resourcePool parameter.

7. Click the Properties tab for (VC:ResourcePool)resourcePool.

8. Right-click the Properties tab and select Add Property > Mandatory input.

9. Right-click the Properties tab again and select Select value as from the list of proposed properties.
   When you set this property, you set how the user selects the value of the (VC:ResourcePool)resourcePool input parameter.


You have created the layout of the input parameters dialog box that appears when users run the workflow.

What to do next

You have completed the development of the more complex workflow example. You can now validate and run the workflow.

Validate and Run the Complex Workflow Example

After you create a workflow, you can validate it to discover any possible errors. If the workflow contains no errors, you can run it.

Prerequisites

You must have created a workflow, laid out its schema, defined the links and bindings, defined the parameter properties, and created the presentation of the input parameters dialog box before you attempt to validate and run the workflow.
**Procedure**

1. Click **Validation** in the Schema tab of the workflow workbench.
   
   The validation tool detects any errors in the definition of the workflow.

2. When you have eliminated any errors, click **Save and Close** at the bottom of the workflow workbench.
   
   You return to the Orchestrator client.

3. Click the **Workflows** view.

4. In the workflow hierarchical list, select **Workflow Examples > Take a Snapshot of All Virtual Machines in a Resource Pool**.

5. Right-click the Take a Snapshot of All Virtual Machines in a Resource Pool workflow and select **Execute**.
   
   The input parameters dialog box opens and prompts you for a resource pool that contains the virtual machines of which to take a snapshot.

6. Click **Submit** to run the workflow.
   
   A workflow token appears under the Take a Snapshot of All Virtual Machines in a Resource Pool workflow.

7. Click the workflow token to follow the progress of the workflow as it runs.

If the workflow ran successfully, the workflow takes a snapshot of all the virtual machines in the chosen resource pool.
Developing Actions

Orchestrator provides libraries of predefined actions. Actions represent individual functions that you use as building blocks in workflows, Web views, and scripts.

Actions are JavaScript functions. They take multiple input parameters and have a single return value. They can call on any object in the Orchestrator API, or on objects in any API that you import into Orchestrator by using a plug-in.

When a workflow runs, an action takes its input parameters from the workflow’s attributes. These attributes can be either the workflow’s initial input parameters, or attributes that other elements in the workflow set when they run.

The Orchestrator client uses the action icon ( ) to identify actions.

This chapter includes the following topics:

- “Reusing Actions,” on page 101
- “Access the Actions View,” on page 101
- “Components of the Actions View,” on page 102
- “Creating Actions,” on page 102

Reusing Actions

When you define an individual function as an action instead of coding it directly into a scriptable task workflow element, you expose it in the library. When an action is visible in the library, other workflows can use it.

When you define actions independently from the workflows that call upon them, you can update or optimize the actions more easily. Defining individual actions also allows other workflows to reuse actions. When a workflow runs, Orchestrator caches each action only the first time the workflow runs it. Orchestrator can then reuse the cached action. Caching actions is useful for recursive calls in a workflow, or fast loops.

You can duplicate actions, export them to other workflows or packages, or move them to a different category in the actions hierarchical list.

Access the Actions View

The Orchestrator client interface features an Actions view that provides access to the Orchestrator server’s libraries of actions.

The Actions view on the left side of the Orchestrator client interface presents you with a hierarchical list of all the actions available in the Orchestrator server.
Procedure

1. Click Actions on the left side of the client interface.
2. Browse the libraries of actions by expanding the nodes of the actions hierarchical list.

You can use the Actions view to view information about the actions in the libraries and create and edit actions.

Components of the Actions View

When you click an action in the actions hierarchical list, information about that action appears in the Orchestrator client's right pane.

The Actions view presents four tabs.

- **General**: Displays general information about the action, including its name, its version number, the permissions, and a description.
- **Scripting**: Shows the action's return types, input parameters, and the JavaScript code that defines the action's function.
- **Events**: Shows all the events that this action encountered or triggered.
- **Permissions**: Shows which users and user groups have permission to access this action.

Creating Actions

You can define individual functions as actions that other elements, such as workflows, can use. Actions are JavaScript functions with defined input and output parameters and permissions.

- **Create an Action** on page 102
  When you define an individual function as an action, instead of coding it directly into a scriptable task workflow element, you can expose it in the library for other workflows to use.

- **Find Elements That Implement an Action** on page 103
  If you edit an action and change its behavior, you might inadvertently break a workflow or application that implements that action. Orchestrator provides a function to find all of the actions, workflows, or packages that implement a given element. You can check whether modifying the element affects the operation of other elements.

- **Action Coding Guidelines** on page 103
  To optimize the performance of workflows and to maximize the potential to reuse actions, you should follow some basic coding guidelines when creating actions.

Create an Action

When you define an individual function as an action, instead of coding it directly into a scriptable task workflow element, you can expose it in the library for other workflows to use.

**Procedure**

1. Click the Actions view in the Orchestrator client.
2. Expand the root of the actions hierarchical list and navigate to the module in which you want to create the action.
3. Right-click the module and select Add action.
4. Enter the name of the action in the text box and click OK.
5. Right-click the action and select Edit.
6 Click the **Scripting** tab.
7 To change the default return type, click the **void** link.
8 Add the action input parameters by clicking the arrow icon.
9 Write the action script.
10 Click **Save and close**.

Your custom action is added to the library of actions.

**What to do next**

You can use the new custom action in a workflow.

## Find Elements That Implement an Action

If you edit an action and change its behavior, you might inadvertently break a workflow or application that implements that action. Orchestrator provides a function to find all of the actions, workflows, or packages that implement a given element. You can check whether modifying the element affects the operation of other elements.

**IMPORTANT** The **Find Elements that Use this Element** function checks all packages, workflows, and policies, but it does not check in scripts. Consequently, modifying an action might affect an element that calls this action in a script that the **Find Elements that Use this Element** function did not identify.

**Procedure**

1 Click the **Actions** view in the Orchestrator client.
2 Expand the nodes of the actions hierarchical list to navigate to a given action.
3 Right-click the action and select **Find Elements that Use this Element**.
   A dialog box shows all of the elements, such as workflows or packages, that implement this action.
4 Double-click an element in the list of results to show that element in the Orchestrator client.

You located all of the elements that implement a given action.

**What to do next**

You can check whether modifying this element affects any other elements.

## Action Coding Guidelines

To optimize the performance of workflows and to maximize the potential to reuse actions, you should follow some basic coding guidelines when creating actions.

**Basic Action Guidelines**

When you create an action, you must use basic guidelines.

- Every action must include a description of its role and function.
- Write short, elementary actions and combine them in a workflow.
- Avoid writing actions that perform multiple functions, because this limits the potential for reusing the action.
- Avoid actions that run for long periods of time. Instead, create a loop in the workflow and include a Waiting Event or Waiting Timer element after the action element.
- Do not write check points in actions. Workflows set a check point at the start and end of each element’s run.

- Avoid writing loops in an action. Create loops in the workflow instead. If the server restarts, a running workflow resumes at its last check point, at the start of an element. If you write a loop inside an action and the server restarts while the workflow is running that action, the workflow resumes at the check point at the beginning of that action, and the loop starts again from the beginning.

**Action Naming Guidelines**

Use basic guidelines when you name actions.

- Write action names in English.

- Start action names with a lowercase letter. Use an uppercase letter at the beginning of each conjoined word in the name. For example, `myAction`.

- Make action names as explicit as possible, so that the function of the action is clear. For example, `backupAllVMsInPool`.

- Make module names as explicit as possible.

- Make module names unique.

- Use the inverse Internet address format for module names. For example, `com.vmware.myactions.myAction`.

**Action Parameter and Attribute Guidelines**

Use basic guidelines when you write action parameter and attribute definitions.

- Write parameter and attribute names in English.

- Start parameter and attribute names with a lowercase letter.

- Make parameter and attribute names as explicit as possible.

- Preferably limit parameter and attribute names to a single word. If a name must contain more than one word, use an uppercase letter at the beginning of each conjoined word in the name. For example, `myParameter`.

- Use the plural form for parameters and attributes that represent an array of objects.

- Make variable names unambiguous, for example, `displayName`.

- Include a description for each attribute to describe its purpose.

- Do not use an excessive number of parameters in a single action.
Orchestrator uses JavaScript to create building blocks from which you create actions, workflow elements, and policies.

Orchestrator implements the Mozilla Rhino JavaScript engine to provide a way to write new scripts. The scripting engine provides version control, variable type checking, name space management, automatic completion, and exception handling.

The Mozilla Rhino JavaScript engine uses only basic JavaScript language features, such as if, loops, arrays, and strings. The other objects that you can use in scripts are the objects that the Orchestrator API provides, or objects from any other API that you import into Orchestrator through a plug-in and that you map to JavaScript objects.

This chapter includes the following topics:

- “Orchestrator Elements that Require Scripting,” on page 105
- “Using the Orchestrator API,” on page 106
- “Exception Handling Guidelines,” on page 111
- “Orchestrator JavaScript Examples,” on page 112

**Orchestrator Elements that Require Scripting**

Not all Orchestrator elements require you to write scripts. To provide maximum flexibility to your applications, you can customize certain elements by adding JavaScript functions.

You can add scripts in the following Orchestrator elements.

**Actions**

Actions are scripted functions. Ideally, you should limit the scripting you write for an action to a single operation, to maximize the potential for action reuse by other elements, such as other workflows.

**Policies**

You set policies using scripts that watch for trigger events. Policies launch defined orchestration operations when the trigger events occur.

**Workflows**

The Scriptable Task workflow element allows you to write a custom scripted operation or sequence of operations that you can use in the workflows. You also define the Boolean decision statement for custom decision elements in scripts that return either true or false.
Using the Orchestrator API

The Orchestrator API exposes as JavaScript objects and methods all of the objects that Orchestrator can access through its plug-ins.

For example, you can access JavaScript implementations of the vCenter Server 4.0 API through the Orchestrator API, for use in scripted elements that you create. You can also access JavaScript implementations of the objects from all of the other plug-ins you installed in the Orchestrator server. When you create a plug-in to a third-party application, you map the objects from its API to JavaScript objects that the Orchestrator API then exposes.

You can use the features of the Orchestrator implementation of the Mozilla Rhino JavaScript engine to help you write scripts.

Procedure

1. **Access the Scripting Engine from the Workflow Workbench** on page 106
   The Orchestrator scripting engine implements the Mozilla Rhino JavaScript engine to help you write scripts for scripted elements in workflows. You access the scripting engine for scripted workflow elements from the Scripting tab in the workflow workbench.

2. **Access the Scripting Engine from the Action or Policy Workbench** on page 107
   The Orchestrator scripting engine implements the Mozilla Rhino JavaScript engine to help you write scripts for actions or policies. You access the scripting engine for actions and policies from the Scripting tabs in the action and policy workbenches.

3. **Access the Orchestrator API Explorer** on page 107
   Orchestrator provides an API Explorer to allow you to search the Orchestrator API and add JavaScript objects to scripted elements.

4. **Use the Orchestrator API Explorer to Find Objects** on page 107
   The Orchestrator API exposes the API of all plugged-in technologies, including the entire vCenter Server API. The Orchestrator API Explorer helps you find the objects you need to add to scripts.

5. **Add JavaScript Objects to Scripts** on page 108
   The Orchestrator scripting engine helps you add objects and functions when you write scripts. Automatic insertion of objects and functions and automatic completion of lines of scripting accelerates the scripting process and minimizes the potential for writing errors in scripts.

6. **Add Parameters to Scripts** on page 109
   The Orchestrator scripting engine helps you to import available parameters into scripts.

7. **Access Java Classes in JavaScript** on page 110
   By default, Orchestrator restricts JavaScript access to a limited set of Java classes. If you require JavaScript access to a wider range of Java classes, you must set an Orchestrator system property to grant this access.

Access the Scripting Engine from the Workflow Workbench

The Orchestrator scripting engine implements the Mozilla Rhino JavaScript engine to help you write scripts for scripted elements in workflows. You access the scripting engine for scripted workflow elements from the Scripting tab in the workflow workbench.

Procedure

1. Right-click a workflow in the **Workflows** view of the Orchestrator client and select **Edit**.
2. Click the **Schema** tab in the workflows workbench.
3  Add a Scriptable Task element or a Custom Decision element to the workflow schema.

4  Click on the scriptable element’s Scripting tab.

You accessed the scripting engine to define the scripted functions of workflow elements. The Scripting tab allows you to navigate through the API, consult documentation about the objects, search for objects, and write JavaScript.

What to do next
Search the Orchestrator API using the API Explorer.

Access the Scripting Engine from the Action or Policy Workbench

The Orchestrator scripting engine implements the Mozilla Rhino JavaScript engine to help you write scripts for actions or policies. You access the scripting engine for actions and policies from the Scripting tabs in the action and policy workbenches.

Procedure

1  Right-click an action or policy in the Actions or Policies views of the Orchestrator client and select Edit.
2  Click the Scripting tab in the action or policy workbench.

You accessed the scripting engine to define the scripted functions of action or policy elements. The Scripting tab allows you to navigate through the API, consult documentation about the objects, search for objects, and write JavaScript.

What to do next
Search the Orchestrator API using the API Explorer.

Access the Orchestrator API Explorer

Orchestrator provides an API Explorer to allow you to search the Orchestrator API and add JavaScript objects to scripted elements.

Procedure

◆  You can access the API Explorer from either the Orchestrator client or from the Scripting tabs of the workflow, policy, and action workbenches
  ◆  To access the API Explorer from the Orchestrator client, click Tools > API Explorer in the Orchestrator client tool bar.
  ◆  To access the API Explorer from the Scripting tabs of the workflow, policy, and action workbenches, click Search API on the left.

The API Explorer appears, allowing you to search all the objects and functions of the Orchestrator API.

What to do next
Use the API Explorer to write scripts for scriptable elements.

Use the Orchestrator API Explorer to Find Objects

The Orchestrator API exposes the API of all plugged-in technologies, including the entire vCenter Server API. The Orchestrator API Explorer helps you find the objects you need to add to scripts.

Prerequisites

The API Explorer is open.
Procedure

1. Enter the name or part of a name of an object in the API Explorer Search text box and click Search.
   To limit your search to a particular object type, uncheck or check the Scripting Class, Attributes & Methods and Types & Enumerations check boxes.

2. Double-click the element in the proposed list.
   The object is highlighted in the hierarchical list on the left. A documentation pane under the hierarchical list presents information about the object.

You found the object you were looking for.

What to do next

Use the objects you find in scripts.

JavaScript Object Types in the API Explorer

The API Explorer identifies and groups together the different types of JavaScript objects in the hierarchical tree on the left of the Scripting tab or API Explorer dialog box. The API Explorer uses different icons to help you identify the different object types.

Table 4-1 describes the different object types of the Orchestrator API and shows their icon.

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Icon in Hierarchical List</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td></td>
<td>Types</td>
</tr>
<tr>
<td>Function set</td>
<td></td>
<td>Internal type containing a set of static methods</td>
</tr>
<tr>
<td>Primitive</td>
<td></td>
<td>Primitive types</td>
</tr>
<tr>
<td>Object</td>
<td></td>
<td>Standard Orchestrator scripting objects</td>
</tr>
<tr>
<td>Attribute</td>
<td></td>
<td>JavaScript attributes</td>
</tr>
<tr>
<td>Method</td>
<td></td>
<td>JavaScript methods</td>
</tr>
<tr>
<td>Constructor</td>
<td>C</td>
<td>JavaScript constructors</td>
</tr>
<tr>
<td>Enumeration</td>
<td></td>
<td>JavaScript enumerations</td>
</tr>
<tr>
<td>String set</td>
<td>()</td>
<td>String set, default values</td>
</tr>
<tr>
<td>Module</td>
<td></td>
<td>A category of actions</td>
</tr>
<tr>
<td>Plug-in</td>
<td>Image defined by plug-in</td>
<td>The APIs exposed to Orchestrator though plug-ins</td>
</tr>
</tbody>
</table>

Add JavaScript Objects to Scripts

The Orchestrator scripting engine helps you add objects and functions when you write scripts. Automatic insertion of objects and functions and automatic completion of lines of scripting accelerates the scripting process and minimizes the potential for writing errors in scripts.

Prerequisites

A scripted element is open for editing and its Scripting tab is open.
Procedure

1. Navigate through the hierarchical list of objects on the left of the Scripting tab, or use the API Explorer search function, to select an object to add to the script.

2. Right-click the object and select Copy.
   If the scripting engine does not allow you to copy the object, this object is not possible in the context of the script.

3. Right-click in the scripting pad, and paste the object you copied into the appropriate place in the script.
   The scripting engine enters the object into the script, complete with its constructor and an instance name.
   For example, if you copied the Date object, the scripting engine pastes the following code into the script.
   ```javascript
   var myDate = new Date();
   ```

4. Copy and paste a method to add to the script.
   The scripting engine completes the method call, adding the required attributes.
   For example, if you copied the cloneVM() method from the com.vmware.library.vc.vm module, the scripting engine pastes the following code into the script.
   ```javascript
   System.getModule("com.vmware.library.vc.vm").cloneVM(vm,folder,name,spec)
   ```
   The scripting engine highlights those parameters that you already defined in the element. Any undefined parameters remain unhighlighted.

5. Place the cursor at the end of an Object you pasted into the script and press Ctrl+space to select from a contextual list of possible methods and attributes that the object can call.

   NOTE The automatic completion feature is currently experimental.

6. Continue copying and pasting from the hierarchical list on the left until you have added all the required objects and functions to the script.

You added object and functions to the script.

What to do next

Add parameters to the script.

Add Parameters to Scripts

The Orchestrator scripting engine helps you to import available parameters into scripts.

If you have already defined parameters for the element you are editing, they appear as links in the Scripting tab toolbar.

Prerequisites

A scripted element is open for editing and its Scripting tab is open.
Procedure
1  Move the cursor to the appropriate position in a script in the scripting pad of the Scripting tab.
2  Click the parameter link in the Scripting tab toolbar.
   Orchestrator inserts the parameter at the position of the cursor.
3  Insert a parameter with a null value into the script.
   If you pass null values to primitive types such as integers, Booleans, and Strings, the Orchestrator scripting API automatically sets the default value for this argument.
You added parameters to the script.

What to do next
Add access to Java classes in scripts.

Access Java Classes in JavaScript
By default, Orchestrator restricts JavaScript access to a limited set of Java classes. If you require JavaScript access to a wider range of Java classes, you must set an Orchestrator system property to grant this access.

Allowing the JavaScript engine full access to the Java virtual machine (JVM) presents potential security issues. Malformed or malicious scripts might have access to all of the system components to which the user who runs the Orchestrator server has access. Consequently, by default the Orchestrator JavaScript engine can access only the classes in the java.util.* package.

If you require JavaScript access to classes outside of the java.util.* package, you can list in a configuration file the Java packages to which to grant JavaScript access. You then set the com.vmware.scripting.rhino-class-shutter-file system property to point to this file.

Procedure
1  Create a text configuration file to store the list of Java packages to which to grant JavaScript access.
   For example, to grant JavaScript access to all the classes in the java.net package and to the java.lang.Object class, you add the following content to the file.
   java.net.*
   java.lang.Object
2  Save the configuration file with an appropriate name and in an appropriate place.
3  Open the vmo.properties system properties file.
   The vmo.properties file is in the following location:
   - <install-directory>\VMware\Orchestrator\app-server\server\vmo\conf if you installed the standalone version of Orchestrator.
   - <install-directory>\VMware\Infrastructure\Orchestrator\app-server\server\vmo\conf if the vCenter Server installer installed Orchestrator.
4  Set the com.vmware.scripting.rhino-class-shutter-file system property by adding the following line to the vmo.properties file.
   com.vmware.scripting.rhino-class-shutter-file=path_to_your_configuration_file
5  Save the vmo.properties file.
6  Restart the Orchestrator server.
   The JavaScript engine has access to the Java classes that you specified.
Exception Handling Guidelines

The Orchestrator implementation of the Mozilla Rhina JavaScript Engine supports exception handling, to allow you to process errors. You must use the following guidelines when writing exception handlers in scripts.

- Use the following European Computer Manufacturers Association (ECMA) error types. Use Error as a generic exception that plug-in functions return, and the following specific error types.
  - TypeError
  - RangeError
  - EvalError
  - ReferenceError
  - URIError
  - SyntaxError

The following example shows a URIError definition.

```javascript
try {
  ...
  throw new URIError("VirtualMachine with ID 'vm-0056'
                     not found on 'vcenter-test-1'") ;
  ...
} catch ( e if e instanceof URIError ) {
  ...
}
```

- Avoid creating local error types in scripts, but rather use predefined error types. Do not use the following type of error handling within scripts.

```javascript
try {
  ...
  throw new MyError("this is a message") ;
  ...
} catch ( e if e instanceof MyError ) {
  ...
}
```

- All exceptions that scripts do not catch must be simple string objects of the form `<type>:SPACE<human readable message>`, as the following example shows.

```
throw "ValidationError: The input parameter 'myParam' of type 'string' is too short."
```

- Write human readable messages as clearly as possible.
Simple string exception type checking must use the following pattern.

```
try {
    throw "VMwareNoSpaceLeftOnDatastore: Datastore 'myDatastore' has no space left" ;
} catch ( e if (typeof(e)="$string" && e.indexOf("VMwareNoSpaceLeftOnDatastore:")) == 0) {
    System.log("No space left on device") ;
    // Do something useful here
}
```

Simple string exception type checking, must use the following pattern in scripted elements in workflows.

```
if (typeof(errorCode)="string" && errorCode.indexOf("VMwareNoSpaceLeftOnDatastore:")) == 0) {
    // Do something useful here
}
```

**Orchestrator JavaScript Examples**

You can cut, paste, and adapt the Orchestrator JavaScript examples to help you write JavaScripts for common orchestration tasks.

- **Basic Scripting Examples** on page 113
  Workflow scripted elements, actions, and policies require basic scripting of common tasks. You can cut, paste, and adapt these examples into your scripted elements.

- **Email Scripting Examples** on page 114
  Workflow scripted elements, actions, and policies require scripting of common email-related tasks. You can cut, paste, and adapt these examples into your scripted elements.

- **File System Scripting Examples** on page 115
  Workflow scripted elements, actions, and policies require scripting of common file system tasks. You can cut, paste, and adapt these examples into your scripted elements.

- **LDAP Scripting Examples** on page 115
  Workflow scripted elements, actions, and policies require scripting of common LDAP tasks. You can cut, paste, and adapt these examples into your scripted elements.

- **Logging Scripting Examples** on page 115
  Workflow scripted elements, actions, and policies require scripting of common logging tasks. You can cut, paste, and adapt these examples into your scripted elements.

- **Networking Scripting Examples** on page 116
  Workflow scripted elements, actions, and policies require scripting of common networking tasks. You can cut, paste, and adapt these examples into your scripted elements.

- **vCenter Server Scripting Examples** on page 116
  Workflow scripted elements, actions, and policies require scripting of common vCenter Server tasks. You can cut, paste, and adapt these examples into your scripted elements.

- **Workflow Scripting Examples** on page 118
  Workflow scripted elements, actions, and policies require scripting examples of common workflow tasks. You can cut, paste, and adapt these examples into your scripted elements.
Basic Scripting Examples

Workflow scripted elements, actions, and policies require basic scripting of common tasks. You can cut, paste, and adapt these examples into your scripted elements.

Check if a Variable Exists

The following JavaScript example checks whether a variable exists in a workflow.

```javascript
function isDefined(variable) {
    try {
        eval(""+variable+"");
        return true;
    } catch (ex) {
        return false;
    }
}
var a = 10;
if (isDefined('a')) {
    System.log("is define");
} else {
    System.log("Not defined");
}
```

Setting and Obtaining Properties from a Hashtable

The following JavaScript example sets properties in a hashtable and obtains the properties from the hashtable. In the following example, the key is always a String and the value is an object, a number, a Boolean, or a String.

```javascript
var table = new Properties();
table.put("myKey",new Date());
// get the object back
var myDate= table.get("myKey");
System.log("Date is : "+myDate);
```

Replace the Contents of a String

The following JavaScript example replaces the content of a String and replaces it with new content.

```javascript
var str1 = '\"hello\"';
var reg = new RegExp("\\(\")\", "g");
var str2 = str1.replace(reg,"]\")
System.log("\"+str2); // result : \\hello\"
```

Access XML Documents

The following JavaScript example allows you to access XML documents from JavaScript by using ECMAScript for XML (E4X).

```javascript
var people = <people>
    <person id="1">
        <name>Moe</name>
    </person>
    <person id="2">
        <name>Larry</name>
    </person>
</people>
System.log("Native XML datatype : " + typeof(people));
System.log("which is a list : " + people.person.length());
System.log("whose elements are indexable : " + people.person[0]);
people.person[0].@id='47';
System.log("and mutable! : " + people.person.(name=='Moe'));
delete people.person[0];
people.person[1] = new XML("<person id="+"3">James</name></person>");
for each(var person in people.person{  
  System.log("- " + person.name);
})

Compare Types
The following JavaScript example checks whether an object matches a given object type.

var path = 'myurl/test';
if(typeof(path, String)){  
  throw("string");
} else {  
  throw("other");
}

Run a Command in the Orchestrator Server
The following JavaScript example allows you to run a command line on the Orchestrator server. Use the same credentials as those used to start the server.

var cmd = new Command("ls -al") ;
cmd.execute(true) ;
System.log(cmd.output) ;

Email Scripting Examples
Workflow scripted elements, actions, and policies require scripting of common email-related tasks. You can cut, paste, and adapt these examples into your scripted elements.

Obtain an Email Address
The following JavaScript example obtains the email address of the current owner of a running script.

var emailAddress = Server.getRunningUser().emailAddress ;

Send an Email
The following JavaScript example sends an email to the defined recipient, through an SMTP server, with the defined content.

var message = new EmailMessage() ;
message.smtpHost = "smtpHost" ;
message.subject= "my subject" ;
message.toAddress = "receiver@vmware.com" ;
message.fromAddress = "sender@vmware.com" ;
message.addMimePart("This is a simple message","text/html") ;
message.sendMessage() ;
File System Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common file system tasks. You can cut, paste, and adapt these examples into your scripted elements.

Obtain the Contents of a File

The following JavaScript example obtains the contents of a file from the Orchestrator server host machine.

```javascript
var fileReader = new FileReader("/home/vmware/readme.txt") ;
fileReader.open() ;
var fileContentAsString = fileReader.readAll();
fileReader.close() ;
```

Add Content to a Simple Text File

The following JavaScript example adds content to a text file.

```javascript
var fileWriter = new FileWriter("/home/vmware/readme.txt") ;
fileWriter.open() ;
fileWriter.writeLine("File written at : "+new Date());
fileWriter.writeLine("Another line");
fileWriter.close() ;
```

LDAP Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common LDAP tasks. You can cut, paste, and adapt these examples into your scripted elements.

Convert LDAP Objects to Active Directory Objects

The following JavaScript example converts LDAP group elements to Active Directory user group objects, and the reverse.

```javascript
var ldapGroup ;
// convert from ldap element to Microsoft:UserGroup object
var adGroup = ActiveDirectory.search("UserGroup",ldapGroup.commonName) ;
// convert back to LdapGroup element
var ldapElement = Server.getLdapElement(adGroup.distinguishedName) ;
```

Logging Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common logging tasks. You can cut, paste, and adapt these examples into your scripted elements.

Persistent Logging

The following JavaScript example creates persistent log entries.

```javascript
Server.log("This is a persistant message", "enter a long description here");
Server.warn("This is a persistant warning", "enter a long description here");
Server.error("This is a persistant error", "enter a long description here");
```

Non-Persistent Logging

The following JavaScript example creates non-persistent log entries.

```javascript
System.log("This is a non-persistant log message");
System.warn("This is a non-persistant log warning");
System.error("This is a non-persistant log error");
```
Networking Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common networking tasks. You can cut, paste, and adapt these examples into your scripted elements.

Obtain Text from a URL

The following JavaScript example accesses a URL, obtains text, and converts it to a String.

```javascript
var url = new URL("http://www.vmware.com")
var htmlContentAsString = url.getContent()
```

vCenter Server Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common vCenter Server tasks. You can cut, paste, and adapt these examples into your scripted elements.

Access Managed Object Types

The following JavaScript example allows Orchestrator to use scripting to access vCenter Server managed objects through the vCenter Server 4.0 plug-in.

```javascript
var vm = ...
// Get the property 'name'
var name = vm.name;  // returns a string
// return a VcEnvironmentBrowser managed object
var environmentBrowser = vm.environmentBrowser;
```

Access Data Object Types

The following JavaScript example allows Orchestrator to use scripting to access vCenter Server data objects through the vCenter Server 4.0 plug-in.

```javascript
var vimHost = ...  // vCenter Server host connection
var virtualMachineSnapshotInfo = ...  // VcVirtualMachineSnapshotInfo data object

// There no automatic conversion between ManagedObjectReference and VimManagedObject
// in a 'Data Object Type'. virtualMachineSnapshotRef is only the reference to the
// 'Managed Object Type' not the object itself
var virtualMachineSnapshotRef = virtualMachineSnapshotInfo.currentSnapshot;

// Convert from ManagedObjectReference to a VimManagedObject.
// The concrete class is VcVirtualMachineSnapshot.
var virtualMachineSnapshot = VcPlugin.convertToVimManagedObject(vimHost,
virtualMachineSnapshotRef);

// The reverse operation, if required.
```

Handle Enumeration Types

The following JavaScript example allows Orchestrator to use scripting to handle vCenter Server enumerations through the vCenter Server 4.0 plug-in.

```javascript
// a VcSharesLevel FINDER ENUMERATION TYPE, for example
// received from an input parameter
var sharesLevel = ...
```
// get the String value of the FINDER ENUMERATION TYPE
var sharesLevelString = sharesLevel.name;

// Convert from the String value to a static value of VcSharesLevel
// SCRIPTING TYPE
var level = VcSharesLevel.fromString(sharesLevel.name);

// Get the String value of the VcSharesLevel SCRIPTING TYPE
System.log("Shares Level : " + level.value);

// Get a FINDER ENUMERATION
finder = Server.findForType("VC:SharesLevel", "normal");

Discover Host Machines and Virtual Machines

The following JavaScript example allows Orchestrator to use scripting to find host machines and virtual machines through the vCenter Server 4.0 plug-in.

var vimHosts = VcPlugin.getVimHosts();
System.log(vimHosts.length + " Vim hosts found");
for (var i = 0; i < vimHosts.length; i++) {
    var vimHost = vimHosts[i];
    System.log("Vim host '" + vimHost.id + ":'"');
}

// Hierarchy entry point
var rootFolder = vimHost.rootFolder;

// Get the property 'name'
var name = rootFolder.name;
System.log("--- Root folder '" + name + ":'"');

// Get the folder's data centers
var datacenters = rootFolder.datacenter;
if (datacenters != null) {
    for (var j = 0; j < datacenters.length; j++) {
        var datacenter = datacenters[j];
        System.log("--- Datacenter '" + datacenter.id + ":'"');
    }
}

// Method to get all the host systems in a vCenter Server host
var hostSystems = vimHost.getAllHostSystems();
if (hostSystems != null) {
    for (var j = 0; j < hostSystems.length; j++) {
        var hostSystem = hostSystems[j];
        System.log("--- HostSystem '" + hostSystem.id + ":'"');
    }
}

// Method to get all the virtual machines in a vCenter Server host
var vms = vimHost.getAllVirtualMachines();
if (vms != null) {
    for (var j = 0; j < vms.length; j++) {
        var vm = vms[j];
        System.log("--- VM '" + vm.id + ":'"');
        System.log("--- VM '" + vm.getName() + ":'"');
    }
}
Workflow Scripting Examples

Workflow scripted elements, actions, and policies require scripting examples of common workflow tasks. You can cut, paste, and adapt these examples into your scripted elements.

Return All Workflows Run by the Current User

The following JavaScript example obtains all workflow runs from the server and checks whether they belong to the current user. You can use this scripting with Webview components, for example.

```javascript
var allTokens = Server.findAllForType('WorkflowToken');
var currentUser = Server.getCredential().username;
var res = [];
for(var i = 0; i<res.length; i++){
    if(allTokens[i].runningUserName == currentUser){
        res.push(allTokens[i]);
    }
}
return res;
```

Schedule a Workflow

The following JavaScript example starts a workflow with a given set of properties, then schedules it to start one hour later.

```javascript
var workflowToLaunch = myWorkflow ;
// create parameters
var workflowParameters = new Properties() ;
workflowParameters.put("name","John Doe") ;
// change the task name
workflowParameters.put("__taskName","Workflow for John Doe") ;

// create scheduling date one hour in the future
var workflowScheduleDate = new Date() ;
var time = workflowScheduleDate.getTime() + (60*60*1000) ;
workflowScheduleDate.setTime(time) ; var scheduledTask = workflowToLaunch.schedule(workflowParameters,workflowScheduleDate);
```
Creating Packages

Packages are the vehicle for transporting content from one Orchestrator server to another. Packages can contain workflows, actions, policies, Web views, configurations, or resources.

When you add an element to a package, Orchestrator checks for dependencies and adds any dependent elements to the package. For example, if you add a workflow that uses actions or other workflows, Orchestrator adds those actions and workflows to the package.

When you import a package, the server compares the versions of the different elements of its contents to matching local elements. The comparison shows the differences in versions between the local and imported elements. The administrator can decide whether to import the package, or choose specific elements to import.

Packages feature digital rights management to control how the receiving server can use the content of the package. Orchestrator signs packages and encrypts the packages for data protection. Packages can track which users export and redistribute elements by using X509 certificates.

**IMPORTANT** Packages generated by Orchestrator 3.2 are upwardly compatible with Orchestrator 4.0. You can import a package from an Orchestrator 3.2 server into an Orchestrator 4.0 server. Packages from Orchestrator 4.0 are not backwards compatible with Orchestrator 3.2. You cannot import a package that an Orchestrator 4.0 server generates into an Orchestrator 3.2 server.

- **Create a Package** on page 119
  You export workflows, policies, actions, plug-ins, resources, Web views, and configuration elements in packages.

- **Set User Permissions on a Package** on page 120
  You set different levels of permission on a package to limit the access that different users or user groups can have to that package.

Create a Package

You export workflows, policies, actions, plug-ins, resources, Web views, and configuration elements in packages.

**Prerequisites**

You must have elements such as workflows, actions, and policies to add to a package.

**Procedure**

1. Click the Packages view in the Orchestrator client.
2. Click the menu button in the title bar of the Packages hierarchical list and select Add package.
3. Provide a name for the package in the dialog box that opens and click OK.
   The naming convention for packages is `<domain.your_company>.category.<package_name>`. For example, `com.vmware.mycategory.mypackage`.

4. Right-click the package and select Edit.
   The package editor opens.

5. Add a description for the package in the General tab.

6. Click the Workflows tab to add workflows to the package.
   - Click Insert Workflows (list search) to search for and select workflows in a selection dialog box.
   - Click Insert Workflows (tree browsing) to browse and select workflows in a hierarchical list.

7. Click the Policies, Actions, Configurations, and Resources tabs to add policy templates, actions, configuration elements, and resource elements to the package.

8. Click Insert Webview in the Web View tab to add Web views to the package.

9. Click Insert used plug-in in the Used plug-ins tab to add plug-ins to the package.

   You added the required elements to the package.

**What to do next**

You must set the user permissions for this package.

---

### Set User Permissions on a Package

You set different levels of permission on a package to limit the access that different users or user groups can have to that package.

You select the different users and user groups for which to set permissions from the users and user groups in the Orchestrator LDAP server. Orchestrator defines levels of permissions that you can apply to users or groups.

<table>
<thead>
<tr>
<th>View</th>
<th>The user can view the elements in the package, but cannot view the schemas or scripting.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspect</td>
<td>The user can view the elements in the package, including the schemas and scripting.</td>
</tr>
<tr>
<td>Execute</td>
<td>The user can run the elements in the package.</td>
</tr>
<tr>
<td>Edit</td>
<td>The user can edit the elements in the package.</td>
</tr>
<tr>
<td>Admin</td>
<td>The user can set permissions on the elements in the package.</td>
</tr>
</tbody>
</table>

**Prerequisites**

You must have created a package, opened it for editing in the package editor, and added to it the necessary elements.

**Procedure**

1. Click the Permissions tab in the package editor.
2. Click the Add access rights link to define permissions for a new user or user group.
3. Search for a user or user group by entering text in the Search text box.
   The search results show all the users and user groups from the Orchestrator LDAP server that match the search.
4. Select a user or user group and click OK.
5 Right-click the user and select **Add access rights**.

6 Check the appropriate check boxes to set the level of permissions for this user and click **OK**.

   The levels of permissions are not cumulative. To grant a user permission to view the elements, inspect the schema and scripting, run and edit the elements, and change the permissions, you must check all check boxes.

7 Click **Save and Close** to exit the package editor.

You created a package and set the appropriate user permissions.
Plug-ins allow you to use Orchestrator to access and control external technologies and applications. The external technologies that you can access by using plug-ins can include virtualization management tools, email systems, databases, directory services, and remote control interfaces.

Orchestrator provides a set of standard plug-ins to allow you to incorporate such technologies as the VMware vCenter Server API and email capabilities into workflows. In addition, the Orchestrator open plug-in architecture allows you to develop plug-ins to access other applications. Orchestrator implements open standards, to simplify integration with external systems.

Each plug-in exposes an external product API to the Orchestrator platform. Plug-ins provide an inventory of objects, extend the scripting engine with new object types and publish notification events from the external system to Orchestrator triggers. Each plug-in can also provide one or more packages that contain elements that represent the typical use cases of the integrated product in an automated fashion.

The Orchestrator open plug-in architecture and the related software development kit (SDK) allow you to open third-party products and tools for orchestration.

This chapter includes the following topics:

- “Components and Architecture of a Plug-In,” on page 123
- “Create an Orchestrator Plug-In,” on page 142
- “Orchestrator Plug-In API Reference,” on page 157

## Components and Architecture of a Plug-In

Orchestrator plug-ins must include a standard set of components and must adhere to a standard architecture. These practices help you to create plug-ins for as wide a variety of external technologies as possible.

- **Plug-In Components** on page 124
  - Plug-ins comprise a standard set of components that expose objects in the plugged-in technology to the Orchestrator platform. These components allow you to perform orchestration operations on the objects in the plugged-in technology.

- **Accessing the Orchestrator Plug-In API** on page 125
  - The Orchestrator plug-in API provides Java interfaces that you implement to create the plug-in adapter and plug-in factory. The plug-in adapter and factory expose the objects and operations of the plugged-in technology to the Orchestrator server.

- **Roles of the Plug-In Adaptor** on page 126
  - The plug-in adapter is the entry point of the plug-in to the Orchestrator server. To create the plug-in adapter you must implement and extend the IPluginAdaptor interface from the Orchestrator plug-in API.
The plug-in factory defines how Orchestrator finds objects in the plugged-in technology and performs operations on them. To create the plug-in factory you must use the Orchestrator plug-in API to implement and extend the IPluginFactory interface.

To create a plug-in you must define how Orchestrator accesses and interacts with the objects in the plugged-in technology. You define these interactions in the vso.xml file.

The vso.xml file defines how the Orchestrator server interacts with the plugged-in technology. You must include a reference to every type of object or operation to expose to Orchestrator in the vso.xml file.

The vso.xml file contains a set of standard elements. Some of the elements are mandatory while others are optional.

You must provide a unique identifier for every object the plug-in finds in the plugged-in technology. You define the object names in the <finder> elements in the vso.xml file and in scripting objects.

You deliver a completed plug-in as a standard Java archive (JAR) or ZIP file, that you must rename to *.dar. You import this *.dar file into the Orchestrator server.

Plug-In Components

Plug-ins comprise a standard set of components that expose objects in the plugged-in technology to the Orchestrator platform. These components allow you to perform orchestration operations on the objects in the plugged-in technology.

Plug-ins must feature standard components.

**Adapter**

Defines the interface between the plugged-in technology and the Orchestrator server. The adapter is the point of entry of the plug-in to the orchestration platform. The adapter creates the plug-in factory, manages the loading and unloading of the plug-in, and manages the events that occur on the objects in the plugged-in technology.

**Factory**

Defines how Orchestrator finds objects in the plugged-in technology and performs operations on them. The adapter creates a factory for every client session that opens between Orchestrator and a plugged-in technology.

**Module**

The plug-in itself, as defined by a collection of Java classes, a vso.xml file, and an Orchestrator package of the workflows and actions that interact with the objects you access through the plug-in.

**Finders**

Interaction rules that define how Orchestrator locates and represents objects in the plugged-in technology. You define finders in the vso.xml file.

**Inventory**

Objects that Orchestrator accesses through finders can appear in the Inventory view in the Orchestrator client. You can perform operations on the objects in the inventory by defining scripting objects.
Scripting Objects
JavaScript object types that provide access to the objects, operations, and attributes in the plugged-in technology. Scripting objects define how Orchestrator accesses the object model of the plugged-in technology through JavaScript. You can access the scripted objects in the Orchestrator scripting API and integrate them into Orchestrator scripted tasks, actions, and policies.

Events
Monitor the state of a given object in the plugged-in technology, that an Orchestrator policy locates through a finder. Orchestrator defines two types of policy events you can monitor through a plug-in: triggers and gauges.

Triggers
Launch a certain operation through the plug-in if a defined event occurs in the plugged-in technology.

Watchers
Watch for a certain event to complete in the plugged-in technology, on behalf of a Waiting Event element in a long-running workflow. You define the trigger event that the watcher watches for in a scriptable task element, the output of which you pass to the Waiting Event element.

You define all of the objects in these components in the plug-in adapter and factory implementations. You map the objects and operations defined in the adapter and factory implementations to Orchestrator objects in an XML definition file named vso.xml.

Accessing the Orchestrator Plug-In API
The Orchestrator plug-in API provides Java interfaces that you implement to create the plug-in adapter and plug-in factory. The plug-in adapter and factory expose the objects and operations of the plugged-in technology to the Orchestrator server.

- The plug-in adapter is defined by the IPluginAdaptor interface.
- The plug-in factory is defined by the IPluginFactory interface.

The plug-in API includes other interfaces, classes, and annotations that you can call on when you create the adapter and factory implementations. For the full list of the classes in the Orchestrator plug-in API, see “Orchestrator Plug-In API Reference,” on page 157.

Locating the Plug-In API Java Archive
Orchestrator provides the classes of the plug-in API in the Orchestrator plug-in API Java archive (JAR) file, vmware-vmo-sdkapi.jar. To develop the plug-in adapter and factory implementations, you must include the vmware-vmo-sdkapi.jar file in your classpath.

You find the vmware-vmo-sdkapi.jar file in the following locations:

- `<install-directory>\VMware\Orchestrator\app-server\server\vmo\lib` if you installed the standalone version of Orchestrator.
- `<install-directory>\VMware\Infrastructure\Orchestrator\app-server\server\vmo\lib` if the vCenter Server installer installed Orchestrator.

Importing the Plug-In API Package
When you write Java classes that implement the Orchestrator plug-in API, you must import the following package using a Java import statement.

```
import ch.dunes.vso.sdk.api.*;
```
Roles of the Plug-In Adaptor

The plug-in adapter is the entry point of the plug-in to the Orchestrator server. To create the plug-in adapter you must implement and extend the IPluginAdaptor interface from the Orchestrator plug-in API.

To create a plug-in adapter, you create a Java class that implements the IPluginAdaptor interface. The plug-in adapter class that you create manages the plug-in factory, events, and triggers in the plugged-in technology. The IPluginAdaptor interface provides methods that you use to perform these tasks.

The plug-in adapter performs the following principal roles.

- **Creates a factory**  
  The most important role of the plug-in adapter is to create the plug-in factory. The plug-in adapter class calls the IPluginAdaptor.createPluginFactory method to create an instance of a class that implements the IPluginFactory interface.

- **Manages events**  
  The plug-in adapter is the interface between the Orchestrator server and the plugged-in technology. The plug-in adapter manages the events that Orchestrator performs or watches for on the objects in the plugged-in technology. The adapter manages events through event publishers. Event publishers are instances of the IPluginEventPublisher interface that the adapter creates by calling the IPluginAdaptor.registerEventPublisher method. Event publishers set triggers and gauges on objects in the plugged-in technology, to allow Orchestrator policies to launch defined actions if certain events occur on the object, or if the object’s values pass certain thresholds. Similarly, you can define PluginTrigger and PluginWatcher instances that define events that Wait Event elements in long-running workflows await.

- **Obtains the plug-in name**  
  You provide a name for the plug-in in the vso.xml file. The plug-in adapter gets this name from the vso.xml file and publishes it in the Orchestrator client Inventory view.

For full details of the IPluginAdaptor interface, all of its methods, and all of the other classes of the plug-in API, see “Orchestrator Plug-In API Reference,” on page 157.

Roles of the Plug-In Factory

The plug-in factory defines how Orchestrator finds objects in the plugged-in technology and performs operations on them. To create the plug-in factory you must use the Orchestrator plug-in API to implement and extend the IPluginFactory interface.

To create a plug-in factory, you create a Java class that implements the IPluginFactory interface. The plug-in factory class that you create defines the finder rules that Orchestrator uses to access objects through the plug-in. The factory allows the Orchestrator server to locate objects by their ID, by their relation to other objects, or by searching for a query string.
The plug-in factory performs the following principal roles.

**Finds objects**
You can create object finders that find objects according to their name and type. You find objects by name and type using the `IPluginFactory.find` method.

**Finds objects related to other objects**
You can create object finders to find objects that relate to a given object by a given relation type. You define relations in the `vso.xml` file. You can also create finders to find dependent child objects that relate to all parents by a given relation type. You implement the `IPluginFactory.findRelation` method to find any objects that are related to a given parent object by a given relation type. You implement the `IPluginFactory.hasChildrenInRelation` method to discover whether at least one child object exists for a parent instance.

**Define queries to find objects according to your own criteria**
You can create object finders that implement query rules of your own choice. You implement the `IPluginFactory.findAll` method to find all objects that satisfy query rules you define when the factory calls this method. You obtain the results of the `findAll` method in a `QueryResult` object that contains a list of all the objects found that match the query rules you define.

**Run commands on objects**
You can create functions to run custom commands by implementing the `IPluginFactory.executePluginCommand` method. You define the command to run when the factory calls this method.

For more information about the `IPluginFactory` interface, all of its methods, and all of the other classes of the plug-in API, see "Orchestrator Plug-In API Reference," on page 157.

**Defining the Application Mapping in the vso.xml File**

To create a plug-in you must define how Orchestrator accesses and interacts with the objects in the plugged-in technology. You define these interactions in the `vso.xml` file.

The `vso.xml` file provides the following information to the Orchestrator server:

- A version, name, and description for the plug-in
- One or more data sources, with the associated plug-in adapter Java class names
- The behavior of the plug-in when the Orchestrator server starts
- Finders that map to Java classes in the plugged-in technology
- JavaScript object types that map to the Java objects and operations in the plugged-in technology in the Orchestrator scripting API
- Enumerations to define a list of values with IDs, names, and description
- Events that Orchestrator policies monitor

The `vso.xml` file must conform to the Orchestrator plug-in XML schema definition. You can access the schema definition at the VMware support site.

http://www.vmware.com/support/orchestrator/plugin-4-0.xsd

**Format of the vso.xml Plug-In Definition File**

The `vso.xml` file defines how the Orchestrator server interacts with the plugged-in technology. You must include a reference to every type of object or operation to expose to Orchestrator in the `vso.xml` file.

As part of the open architecture and standardized implementation of plug-ins, the `vso.xml` file must adhere to a standard format.

Figure 6-1 shows the format of the `vso.xml` plug-in definition file and how the elements nest within each other.
Elements of the vso.xml Plug-In Definition File

The vso.xml file contains a set of standard elements. Some of the elements are mandatory while others are optional.

Each element has attributes that define values for the objects and operations you map to Orchestrator objects and operations. This information describes all of the elements of the vso.xml file and the possible attribute values for each element.

**<module> Element**

A module describes a set of plug-in objects to make available to Orchestrator.

The module contains information about how data from the plugged-in technology maps to Java classes, versioning, how to deploy the module, and how the plug-in appears in the Orchestrator inventory.

The `<module>` element has the following attributes:

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Defines the type of all the <code>&lt;finder&gt;</code> elements in the plug-in</td>
</tr>
<tr>
<td>version</td>
<td>Number</td>
<td>The plug-in version number, for use when reloading packages in a new version of the plug-in</td>
</tr>
<tr>
<td>build-number</td>
<td>Number</td>
<td>The plug-in build number, for use when reloading packages in a new version of the plug-in</td>
</tr>
<tr>
<td>image</td>
<td>Image file</td>
<td>The icon to display in the Orchestrator Inventory</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display-name</td>
<td>String</td>
<td>The name that appears in the Orchestrator Inventory</td>
</tr>
<tr>
<td>interface-mapping-allowed</td>
<td>true or false</td>
<td>VMware strongly discourages interface mapping</td>
</tr>
</tbody>
</table>

### Parent Element

None

### Child Elements

- `<description>`
- `<installation>`
- `<configuration>`
- `<webview-component-library>`
- `<finder-datasources>`
- `<inventory>`
- `<finders>`
- `<scripting-objects>`
- `<enumerations>`

### `<description>` Element

The `<description>` element provides a description of the plug-in.

The `<description>` element has no attributes or child elements.

### `<installation>` Element

The `<installation>` element allows you to install a package or run a script when the server starts.

The `<installation>` element has the following attributes:

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| mode       | always, never, or version | Setting the mode value results in the following behavior when the Orchestrator server starts:
  - The action always runs
  - The action never runs
  - The action runs when the server detects a newer version of the plug-in |

### Parent Element

- `<module>`

### Child Element

- `<action>`

### `<action>` Element

The `<action>` element specifies the action that runs when the Orchestrator server starts.

The `<action>` element attributes provide the path to the Orchestrator package or script that defines the plug-in's behavior when it starts. The `<action>` element has the following attributes.
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>String</td>
<td>The path to the Java package or script from the root of the *.dar file.</td>
</tr>
<tr>
<td>type</td>
<td>install-package or execute-script</td>
<td>Either installs the specified Orchestrator package in the Orchestrator server, or runs the specified script.</td>
</tr>
</tbody>
</table>

### Parent Element

<installation>

### Child Elements

None

---

### <webview-component-library> Element

The `<webview-component-library>` element points to a JAR file containing custom Web view tapestry components that extend Web view capabilities.

The `<webview-component-library>` element has the following attributes.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jar</td>
<td>String</td>
<td>A JAR file containing Web view components</td>
</tr>
<tr>
<td>specification-path</td>
<td>String</td>
<td>The path in the JAR file to the Tapestry component definition file, in the lib folder of the *.dar file. The path must begin with a forward slash (/).</td>
</tr>
</tbody>
</table>

### Parent Element

<module>

### Child Elements

None

---

### <finder-datasources> Element

The `<finder-datasources>` points to the Java class files of the IPluginAdaptor implementation that you create for the plug-in.

You set how users access the plug-in and set timeouts for the various finder calls that Orchestrator performs through the plug-in. Different timeouts apply to the following finder methods from the IPluginFactory interface.

The `<finder-datasources>` element has the following attributes.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Identifies the data source in the <code>&lt;finder&gt;</code> element's datasource attributes. Equivalent to an XML id.</td>
</tr>
<tr>
<td>adaptor-class</td>
<td>Java class</td>
<td>Points to the IPluginAdaptor implementation you define to create the plug-in adapter, for example, com.vmware.plugins.sample.Adaptor.</td>
</tr>
<tr>
<td>concurrent-call</td>
<td>true (default) or false</td>
<td>Allows multiple users to access the adapter at the same time. You must set concurrent-call to false if the plug-in does not support concurrent calls.</td>
</tr>
<tr>
<td>anonymous-login-mode</td>
<td>never (default) or always</td>
<td>Passes or does not pass the user's username and password to the plug-in.</td>
</tr>
<tr>
<td>timeout-fetch-relation</td>
<td>Number; default 30 seconds</td>
<td>Applies to calls from findRelation()</td>
</tr>
<tr>
<td>timeout-find-all</td>
<td>Number; default 60 seconds</td>
<td>Applies to calls from findAll()</td>
</tr>
<tr>
<td>timeout-find</td>
<td>Number; default 60 seconds</td>
<td>Applies to calls from find()</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeout-has-children-in-relation</td>
<td>Number; default 2 seconds</td>
<td>Applies to calls from findChildrenInRelation()</td>
</tr>
<tr>
<td>timeout-execute-plugin-command</td>
<td>Number; default 30 seconds</td>
<td>Applies to calls from executePluginCommand()</td>
</tr>
</tbody>
</table>

**Parent Element**  
<module>

**Child Elements**  
No children

### <inventory> Element

The `<inventory>` element defines the root of the hierarchical list for the plug-in that appears in the Orchestrator client Inventory view and object selection dialog boxes.

The `<inventory>` element does not represent an object in the plugged-in application, but rather represents the plug-in itself as an object in the Orchestrator scripting API. The `<inventory>` element has the following attribute.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>An Orchestrator object type</td>
<td>The type of the <code>&lt;finder&gt;</code> element that represents the root of the hierarchy of objects</td>
</tr>
</tbody>
</table>

**Parent Element**  
<module>

**Child Elements**  
None

### <finders> Element

The `<finders>` element is the container for all the `<finder>` elements.

The `<finders>` element has no attributes.

**Parent Element**  
<module>

**Child Element**  
<finder>

### <finder> Element

The `<finder>` element represents in the Orchestrator client a type of object found through the plug-in.

The `<finder>` element identifies the Java class that defines the object the object finder represents. The `<finder>` element defines how the object appears in the Orchestrator client interface. It also identifies the scripting object that the Orchestrator scripting API defines to represent this object.

Finders act as an interface between object formats used by different types of plugged-in technologies.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>An Orchestrator object type</td>
<td>Type of object represented by the finder</td>
</tr>
<tr>
<td>datasource</td>
<td>&lt;finder-datasource name&gt; attribute</td>
<td>Identifies the Java class that defines the object by using the datasource refid</td>
</tr>
<tr>
<td>dynamic-finder</td>
<td>Java method</td>
<td>Defines a custom finder method you implement in an IDynamicFinder instance, to return the ID and properties of a finder programmatically, instead defining it in the vso.xml file</td>
</tr>
<tr>
<td>hidden</td>
<td>true or false (default)</td>
<td>If true, hides the finder in the Orchestrator client</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>Path to a graphic file</td>
<td>A 16x16 icon to represent the finder in hierarchical lists in the Orchestrator client</td>
</tr>
<tr>
<td>java-class</td>
<td>Name of a Java class</td>
<td>The Java class that defines the object the finder finds and maps to a scripting object</td>
</tr>
<tr>
<td>script-object</td>
<td>&lt;scripting-object type&gt; attribute</td>
<td>The &lt;scripting-object type&gt; attribute, if any, to which to map this finder</td>
</tr>
</tbody>
</table>

### Parent Element

- `<finders>`

### Child Elements

- `<properties>`
- `<relations>`
- `<id>`
- `<inventory-children>`

#### `<properties>` Element

The `<properties>` element is the container for `<finder>` `<property>` elements.

The `<properties>` element has no attributes.

**Parent Element**

- `<finder>`

**Child Element**

- `<property>`

#### `<property>` Element

The `<property>` element maps the found object's properties to Java properties or method calls by using OGNL expressions.

You can call on the methods of the `SDKFinderProperty` class when you implement the plug-in factory to obtain properties for the plug-in factory implementation to process.

You can show or hide object properties in the views in the Orchestrator client. You can also use enumerations to define object properties.

The `<property>` element has the following attributes.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Finder name</td>
<td>The name the <code>FinderResult</code> uses to store the element.</td>
</tr>
<tr>
<td>display-name</td>
<td>Finder name</td>
<td>The displayed property name</td>
</tr>
<tr>
<td>bean-property</td>
<td>Property name</td>
<td>You use the <code>bean-property</code> attribute to identify a property to obtain using <code>get</code> and <code>set</code> operations. If you identify a property named <code>MyProperty</code>, the plug-in defines <code>getMyProperty</code> and <code>setMyProperty</code> operations. You set one or the other of <code>bean-property</code> or <code>property-accessor</code>, but not both.</td>
</tr>
<tr>
<td>property-accessor</td>
<td>The method that obtains a property value from an object</td>
<td>The <code>property-accessor</code> attribute allows you to define an OGNL expression to validate an object's properties. You set one or the other of <code>bean-property</code> or <code>property-accessor</code>, but not both.</td>
</tr>
<tr>
<td>show-in-column</td>
<td>true (default) or false</td>
<td>If true, this property shows in the Orchestrator client results table</td>
</tr>
<tr>
<td>show-in-description</td>
<td>true (default) or false</td>
<td>If true, this property shows in the object description</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hidden</td>
<td>true or false (default)</td>
<td>If true, this property is hidden in all cases</td>
</tr>
<tr>
<td>linked-configuration</td>
<td>Enumeration name</td>
<td>Links a finder property to an enumeration</td>
</tr>
</tbody>
</table>

### Parent Element

| <properties>     |                           |                                                   |

### Child Elements

None

### <relations> Element

The `<relations>` element is the container for `<finder>` `<relation>` elements.

The `<relations>` element has no attributes.

### Parent Element

| <finder>         |                           |                                                   |

### Child Element

| <relation>       |                           |                                                   |

### <relation> Element

The `<relation>` element defines how objects relate to other objects.

You define the relation name in the `<inventory-children>` element.

The `<relation>` element has the following attributes.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Relation name</td>
<td>A name for this relation</td>
</tr>
<tr>
<td>type</td>
<td>Orchestrator object type</td>
<td>The type of the object that relates to another object by this relation</td>
</tr>
</tbody>
</table>

### Parent Element

| <relations>      |                           |                                                   |

### Child Elements

None

### <id> Element

The `<id>` element defines a method to obtain the ID of the object that the finder identifies.

The `<id>` element has the following attributes.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessor</td>
<td>Method name</td>
<td>The accessor attribute allows you to define an OGNL expression to validate an object's ID.</td>
</tr>
</tbody>
</table>

### Parent Element

| <finder>         |                           |                                                   |

### <inventory-children> Element

The `<inventory-children>` element defines the hierarchy of the lists that show the objects in the Orchestrator client `<inventory>` view and object selection boxes.

The `<inventory-children>` element has no attributes.

### Parent Element

| <finder>         |                           |                                                   |

### Child Element

| <relation-link>  |                           |                                                   |

VMware, Inc.  133
**<relation-link> Element**

The `<relation-link>` element defines the hierarchies between parent and child objects.

The `<relation-link>` element has the following attribute.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Relation name</td>
<td>A refid to a relation name</td>
</tr>
</tbody>
</table>

**Parent Element**  
`<inventory-children>`

**Child Elements**  
None

**<events> Element**

The `<events>` element is the container for the `<trigger>` and `<gauge>` elements.

The `<events>` element has no attributes.

**Parent Element**  
`<module>`

**Child Elements**  
- `<description>`
- `<trigger-properties>`

**<trigger> Element**

The `<trigger>` element declares the triggers you can use for this finder. You must implement the `registerEventPublisher()` and `unregisterEventPublisher()` methods of `IPluginAdaptor` to set triggers.

The `<trigger>` element has the following attribute.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Relation name</td>
<td>A name for this relation</td>
</tr>
</tbody>
</table>

**Parent Element**  
`<events>`

**Child Elements**  
None

**<trigger-properties> Element**

The `<trigger-properties>` element is the container for the `<trigger-property>` elements.

The `<trigger-properties>` element has no attributes.

**Parent Element**  
`<trigger>`

**Child Element**  
`<trigger-property>`

**<trigger-property> Element**

The `<trigger-property>` element defines the properties that identify a trigger object.

The `<trigger-property>` element has the following attributes.
### <gauge> Element

The `<gauge>` element defines the gauges you can use for this finder. You must implement the `registerEventPublisher()` and `unregisterEventPublisher()` methods of `IPluginAdaptor` to set gauges.

The `<gauge>` element has the following attributes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Trigger name</td>
<td>A name for the trigger</td>
</tr>
<tr>
<td>display-name</td>
<td>Trigger name</td>
<td>The name that displays in the Orchestrator client</td>
</tr>
<tr>
<td>type</td>
<td>Trigger type</td>
<td>The object type that defines the trigger</td>
</tr>
</tbody>
</table>

**Parent Element**  
`<events>`

**Child Element**  
`<gauge-properties>`

### <gauge-properties> Element

The `<gauge-properties>` element is the container for the `<gauge-property>` elements.

The `<gauge-properties>` element has no attributes.

**Parent Element**  
`<gauge>`

**Child Element**  
`<gauge-property>`

### <gauge-property> Element

The `<gauge-property>` element defines the properties that identify the gauge object.

The `<gauge-property>` element has the following attributes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Gauge name</td>
<td>A name for the gauge</td>
</tr>
<tr>
<td>display-name</td>
<td>Gauge name</td>
<td>The name that displays in the Orchestrator client</td>
</tr>
<tr>
<td>type</td>
<td>Gauge type</td>
<td>The object type that defines the gauge</td>
</tr>
<tr>
<td>min-value</td>
<td>Number</td>
<td>The minimum threshold the gauge watches</td>
</tr>
<tr>
<td>unit</td>
<td>Measurement type</td>
<td>The unit of measurement</td>
</tr>
<tr>
<td>max-value</td>
<td>Number</td>
<td>The maximum threshold the gauge watches</td>
</tr>
</tbody>
</table>

**Parent Element**  
`<gauge-properties>`

**Child Elements**  
None
<scripting-objects> Element
The <scripting-objects> element is the container for the <object> elements.
The <scripting-objects> element has no attributes.

Parent Element <module>
Child Element <object>

<object> Element
The <object> element maps the plugged-in technology’s constructors, attributes, and methods to JavaScript object types that the Orchestrator scripting API exposes.
See “Naming Plug-In Objects,” on page 140 for object naming conventions.
The <object> element has the following attributes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>script-name</td>
<td>JavaScript name</td>
<td>Scripting name of the class. Must be globally unique</td>
</tr>
<tr>
<td>java-class</td>
<td>Java class</td>
<td>The Java class wrapped by this JavaScript class</td>
</tr>
<tr>
<td>create</td>
<td>true (default) or false</td>
<td>If true, you can create a new instance of this class</td>
</tr>
<tr>
<td>strict</td>
<td>true or false (default)</td>
<td>If true, you can only call methods you annotate or declare in the vso.xml file</td>
</tr>
<tr>
<td>description</td>
<td>Text</td>
<td>A description of the object</td>
</tr>
<tr>
<td>url</td>
<td>Number</td>
<td>A link to more information, for example a link to official API documentation</td>
</tr>
</tbody>
</table>

Parent Element <scripting-objects>
Child Elements
- <constructors>
- <attributes>
- <methods>
- <singleton>

<constructors> Element
The <constructors> element is the container for the <object><constructor> elements.
The <constructors> element has no attributes.

Parent Element <constructors>
Child Element <constructor>

<constructor> Element
The <constructor> element defines a constructor method. Used for API documentation.
The <constructor> element has no attributes.

Parent Element <constructors>
Child Elements
- <description>
- <parameters>
Constructor <parameters> Element

The <parameters> element is the container for the <constructor><parameter> elements.

The <parameters> element has no attributes.

Parent Element  <constructor>
Child Element  <parameter>

Constructor <parameter> Element

The <parameter> element defines the constructor’s parameters.

The <parameter> element has the following attributes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Parameter name to use in API documentation</td>
</tr>
<tr>
<td>type</td>
<td>Orchestrator parameter type</td>
<td>Parameter type to use in API documentation</td>
</tr>
</tbody>
</table>

Parent Element  <parameters>
Child Elements  None

<attributes> Element

The <attributes> element is the container for the <object><attribute> elements.

The <attributes> element has no attributes.

Parent Element  <object>
Child Element  <attribute>

<attribute> Element

The <attribute> element maps the attributes of a Java class from the plugged-in technology to JavaScript attributes that the Orchestrator JavaScript engine exposes.

The <attribute> element has the following attributes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java-name</td>
<td>Java attribute</td>
<td>Name of the Java attribute</td>
</tr>
<tr>
<td>script-name</td>
<td>JavaScript object</td>
<td>Name of the corresponding JavaScript object</td>
</tr>
<tr>
<td>return-type</td>
<td>Java object type</td>
<td>The type this attribute returns</td>
</tr>
<tr>
<td>read-only</td>
<td>true or false</td>
<td>If true, you cannot modify this attribute</td>
</tr>
<tr>
<td>is-optional</td>
<td>true or false</td>
<td>If true, this field can be null</td>
</tr>
<tr>
<td>show-in-api</td>
<td>true or false</td>
<td>If false, this attribute does not appear in API documentation</td>
</tr>
</tbody>
</table>

Parent Element  <attributes>
Child Elements  None

<methods> Element

The <methods> element is the container for the <object><method> elements.

The <methods> element has no attributes.
<method> Element

The `<method>` element maps a Java method from the plugged-in technology to a JavaScript method that the Orchestrator JavaScript engine exposes.

The `<method>` element has the following attributes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java-name</td>
<td>Java method</td>
<td>Name of the Java method signature with argument types in parenthesis, for example, <code>getVms(DataStore)</code></td>
</tr>
<tr>
<td>script-name</td>
<td>JavaScript method</td>
<td>Name of the corresponding JavaScript method</td>
</tr>
<tr>
<td>return-type</td>
<td>Java object type</td>
<td>The type this method obtains</td>
</tr>
<tr>
<td>show-in-api</td>
<td>true or false</td>
<td>If false, this method does not appear in API documentation</td>
</tr>
</tbody>
</table>

Method `<parameters>` Element

The `<parameters>` element is the container for the `<method>`-`<parameter>` elements.

The `<parameters>` element has no attributes.

Method `<parameter>` Element

The `<parameter>` element defines the method's input parameters.

The `<parameter>` element has the following attributes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Parameter name</td>
</tr>
<tr>
<td>type</td>
<td>Orchestrator parameter type</td>
<td>Parameter type</td>
</tr>
<tr>
<td>is-optional</td>
<td>true or false</td>
<td>If true, value can be null.</td>
</tr>
</tbody>
</table>

<singleton> Element

The `<singleton>` element creates a singleton instance of a `<script-object>`. You cannot instantiate singleton objects from JavaScript scripts.

The `<singleton>` element has the following attributes.
### <enumerations> Element

The `<enumerations>` element is the container for the `<enumeration>` elements.

The `<enumerations>` element has no attributes.

**Parent Element**  
`<object>`  

**Child Element**  
None

### <enumeration> Element

The `<enumeration>` element defines common values that apply to all objects of a certain type.

If all objects of a certain type require a certain attribute, and if the range of values for that attribute is limited, you can define the different values as enumeration entries. For example, if a type of object requires a `color` attribute, and if the only available colors are red, blue, and green, you can define three enumeration entries to define these three color values. You define entries as child elements of the enumeration element.

The `<enumeration>` element has the following attribute.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Orchestrator object type</td>
<td>Enumeration type</td>
</tr>
</tbody>
</table>

**Parent Element**  
`<enumerations>`  

**Child Elements**  
- `<url>`  
- `<description>`  
- `<entries>`

### <url> Element

The `<url>` element provides a URL that points to documentation about the enumeration.

The `<url>` element has no attributes. You provide the URL between the `<url>` and `</url>` tags.

**Parent Element**  
`<enumeration>`  

**Child Elements**  
None

### <entries> Element

The `<entries>` element is the container for the `<enumeration><entry>` elements.

The `<entries>` element has no attributes.

**Parent Element**  
`<enumeration>`  

**Child Element**  
`<entry>`
<entry> Element

The <entry> element provides a value for an enumeration attribute. The <entry> element has the following attributes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Text</td>
<td>The identifier that objects use to set the enumeration entry as an attribute</td>
</tr>
<tr>
<td>name</td>
<td>Text</td>
<td>The entry name</td>
</tr>
</tbody>
</table>

Parent Element <enumeration>
Child Elements None

Naming Plug-In Objects

You must provide a unique identifier for every object the plug-in finds in the plugged-in technology. You define the object names in the <finder> elements in the vso.xml file and in scripting objects.

The plug-in finder operations that you define in the factory implementation find objects in the plugged-in technology. When the plug-in finds objects, you can use them in Orchestrator workflows and pass them from one workflow element to another. To allow the objects to pass between the elements in a workflow, you must provide a unique identifier for each object.

The Orchestrator server stores only the type and identifier of each object it processes, and stores no information about where or how Orchestrator obtained the object. You must name objects consistently in the plug-in implementation so that you can track the objects you obtain from plug-ins.

If the Orchestrator server shuts down while workflows are running, when you restart the server the workflows resume at the workflow element that was running when the server stopped. The workflow retrieves any objects that the element was processing when the server stopped by using their identifier. This is why you must provide a unique identifier for all objects you find through a plug-in.

Plug-In Object Naming Conventions

You must adhere to Java class naming conventions when naming all objects in plug-ins.

- Use an initial uppercase letter for each word in the name
- Do not use spaces to separate words
- For letters, only use the standard characters A to Z and _a to z
- Do not use special characters, such as accents
- Do not use a number as the first character of a name
- Where possible, use fewer than 10 characters

Table 6-1 shows the additional rules that apply to the different types of object.
Table 6-1. Plug-In Object Naming Rules

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Naming Rules</th>
</tr>
</thead>
</table>
| Plug-In     | • Defined in the `<module>` element in the `vso.xml` file.  
• Must adhere to Java class naming conventions.  
• Plug-in names must be unique. You cannot run two plug-ins with the same name in an Orchestrator server. |
| Finder object | • Defined in the `<finder>` elements in the `vso.xml` file.  
• Must adhere to Java class naming conventions.  
• Finder names must be unique within the plug-in |
| Scripting object | • Defined in the `<scripting-object>` elements in the `vso.xml` file.  
• Must adhere to Java class naming conventions.  
• Scripting object names must be unique within the Orchestrator server  
• To avoid name clashes with scripting objects from other plug-ins, always prefix the scripting object name with the name of the plug-in |

**IMPORTANT** Because of the way in which the workflow engine handles data serialization, do not use the following string sequences in object names.

- `;#`
- `##`
- `#=`  

Using these character sequences in object identifiers causes the workflow engine to parse workflows incorrectly, leading to unexpected behavior when you run the workflows.

### Constructing Object Names

Orchestrator constructs unique object names from the plug-in and finder names you define in the `vso.xml` file.

For example, if you define a plug-in called `SolarSystem` and a finder called `Planet` that finds objects of the `Planet` type, the unique name of the `Planet` finder object is `SolarSystem:Planet`.

The unique name of the scripting object that maps the Java class of the `Planet` object to JavaScript is `SolarSystemPlanet`.

### Structure of the *.dar File

You deliver a completed plug-in as a standard Java archive (JAR) or ZIP file, that you must rename to `*.dar`. You import this `*.dar` file into the Orchestrator server.

The contents of the `*.dar` archive must adhere to the following folder structure and naming conventions:

Table 6-2. Structure of the `vso.xml` File

<table>
<thead>
<tr>
<th>Folders and Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/VSO-INF/vso.xml</code></td>
<td>Defines the mapping of the objects in the plugged-in technology to Orchestrator objects.</td>
</tr>
<tr>
<td><code>/lib/</code></td>
<td>Contains JAR files for the plugged-in technology and for the implementations of the plug-in adapter and factory interfaces.</td>
</tr>
<tr>
<td><code>/resources/</code></td>
<td>Contains resource files needed by the plug-in adapter and factory implementations, such as images, scripts, HTML fragments, and so on. You can organize resources into subfolders. For example, <code>/resources/images/</code> or <code>/resources/scripts/</code>.</td>
</tr>
</tbody>
</table>
To import the plug-in into the Orchestrator server, use the Orchestrator configuration interface. See the *VMware vCenter Orchestrator Administration Guide* for instructions about importing plug-ins using the configuration interface.

**Create an Orchestrator Plug-In**

To create a plug-in to manage an external application using Orchestrator, you must create a plug-in adaptor and a plug-in factory, and map the objects from the plugged-in application to Orchestrator objects in the *vso.xml* file.

The procedure to create a plug-in consists of several sub-procedures. The following procedures demonstrate the plug-in creation process by dissecting the Java classes and *vso.xml* file for an example Java application. The application these procedures examine represents the solar system, containing Java objects to represent the Sun, the planets, and their various moons, as well as defining different operations that you can perform on the objects. The Orchestrator plug-in for this application allows you to manage the solar system application using Orchestrator. Through the plug-in, you can view the objects in the **Inventory** view, access them via a Web view, perform operations on them, and run workflows that automate solar system operations.

**Procedure**

1. **Obtain a Java Application to Plug into Orchestrator** on page 142
   To create a plug-in, you must have a Java application to expose for management by Orchestrator.

2. **Examine an Example Plug-in Adapter** on page 144
   To create a plug-in adapter, you create a Java class that implements the *IPluginAdaptor* interface from the Orchestrator plug-in API.

3. **Examine an Example Plug-In Factory** on page 146
   To create a plug-in factory, you create a Java class that implements the *IPluginFactory* interface from the Orchestrator plug-in API.

4. **Map the Application in the *vso.xml* File** on page 149
   The *vso.xml* file defines how Orchestrator access and interacts with the plugged-in technology. It maps objects and operations in the plugged-in technology to Orchestrator objects and operations.

5. **Create the Plug-In *.dar Archive** on page 155
   The final stage in the creation of a plug-in is to create the *.dar that you import into Orchestrator.

**Obtain a Java Application to Plug into Orchestrator**

To create a plug-in, you must have a Java application to expose for management by Orchestrator.

**Prerequisites**

The following procedures use the solar system example application to demonstrate how to create a plug-in. The ZIP file of Orchestrator examples you can download from the Orchestrator documentation homepage contains the *.dar file for the plug-in, the source files for the solar system example application, and the source files for its plug-in implementation. See “Example Applications” for details of where to download the Orchestrator examples bundle.

**Procedure**

1. Download the ZIP file of Orchestrator examples from the Orchestrator documentation home page.
2. Unzip the bundle to an appropriate location.
3. Navigate to the following location to view the solar system plug-in files.

   /install-directory/vcenter_orchestrator_examples/Plug-Ins
The /Plug-Ins folder contains the following file and folders.

- /vmware-vmosdk-solarsystem.dar, which is the *.dar archive that contains the finished plug-in.
- /SolarSystem, contains the source files for the solar system application for you to inspect.
- /VSOSDK-SolarSystem, contains the implementations of the Orchestrator plug-in API interfaces to define the plug-in adapter and factory for the solar system application.

4 (Optional) Unpack the vmware-vmosdk-solarsystem.dar archive to view the contents of the finished plug-in.

Components of the Solar System Example Application

The solar system example application replicates a solar system, containing objects to represent stars, planets, moons, and operations that you can perform on these objects.

The /SolarSystem/src/com/vmware/solarsystem folder contains the following files.

Table 6-3. Source Files for the Solar System Plug-In Example Application

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CelestialBody.java</td>
<td>A serializable class that defines a generic celestial body, with get and set methods to obtain and set object name and id values.</td>
</tr>
<tr>
<td>Moon.java</td>
<td>Extends CelestialBody and defines methods to obtain and set its volume, and to obtain and set the identifier of the planet that it orbits.</td>
</tr>
<tr>
<td>Planet.java</td>
<td>Extends CelestialBody and defines methods to obtain and set its circumference and gravity level, to obtain, set, add, and remove moons, and to obtain and set the identifier of the star that it orbits.</td>
</tr>
<tr>
<td>Star.java</td>
<td>Extends CelestialBody and defines methods to obtain and set its circumference and surface temperature, and to obtain, set, add, and remove the planets in its orbit.</td>
</tr>
<tr>
<td>SolarSystemRepository.java</td>
<td>Creates an instance of a solar system repository that contains instances of the Star, Planet, and Moon classes. These instances represent the Sun and the planets and their respective moons of Earth's solar system.</td>
</tr>
</tbody>
</table>

The /VSOSDK-SolarSystem/src/com/vmware/orchestrator/api/sample/solarsystem folder contains the following files.

Table 6-4. Source Files for the Solar System Plug-In Implementation

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolarSystemAdapter.java</td>
<td>An implementation of the IPluginAdaptor interface that defines the entry point of the solar system application to Orchestrator.</td>
</tr>
<tr>
<td>SolarSystemEventGenerator.java</td>
<td>A Java class that creates IPluginEventPublisher instances and defines methods to publish events to the Orchestrator policy engine.</td>
</tr>
<tr>
<td>SolarSystemFactory.java</td>
<td>An implementation of the IPluginFactory interface that defines how Orchestrator finds objects through the plug-in, and how to perform operations on those objects.</td>
</tr>
</tbody>
</table>

Note: The source files of the solar system example application are provided for reference purposes. If you import the vmware-vmosdk-solarsystem.dar file into Orchestrator, you do not need to build solar system Java code or create the plug-in from the sources.
**Install the Solar System Plug-In JAR File**

To run the Solar System example plug-in, you must download and install a third-party Java archive file in the plug-in.

The JAR file that the Solar System plug-in requires is available as a free download.

**Procedure**

   
   The `xml-apis.jar` archive is in the distribution.

2. Unpack the Solar System `dar` archive.

3. Copy the `xml-apis.jar` archive to the following location in the Solar System `dar` archive.

   `<solar_system_install_directory>/lib/`

4. Repack the Solar System `dar` archive.

You installed the Java archive file that the Solar System example plug-in requires.

**What to do next**

You can examine the classes of the Solar System example plug-in.

**Examine an Example Plug-in Adapter**

To create a plug-in adapter, you create a Java class that implements the `IPluginAdaptor` interface from the Orchestrator plug-in API.

The following procedure presents the steps involved in creating a plug-in adapter. To illustrate the process, it presents code from the `SolarSystemAdapter` class from the solar system example application. You can download the Orchestrator examples ZIP file from the Orchestrator documentation home page to obtain the sources of the solar system example application.

**Prerequisites**

The following procedure outlines the broad steps in the process to develop a plug-in adapter class. For a description of the role of the plug-in adapter and the other components of a plug-in, you must read “Components and Architecture of a Plug-In,” on page 123.

**Procedure**

1. Create and save a Java file for the plug-in adapter implementation called `<YourApplicationName>Adaptor.java`.

   In the solar system example, the adapter class is called `SolarSystemAdapter.java`.

2. Declare a package to contain the plug-in adapter and factory implementations.

   The solar system example declares the following package to contain the adapter and factory implementations:

   ```java
   package com.vmware.orchestrator.api.sample.solarsystem;
   ```

3. Import the Orchestrator plug-in API interfaces, classes, and enumerations with a Java `import` statement.

   ```java
   import ch.dunes.vso.sdk.api.*;
   ```
4 Import any other classes the adapter implementation requires.
In the solar system example, the adapter implementation requires the following class:
import javax.security.auth.login.LoginException;

5 Declare a public constructor that implements the IPluginAdaptor interface from the Orchestrator plug-in API.
The solar system example adapter declares the SolarSystemAdapter constructor.
public class SolarSystemAdapter implements IPluginAdaptor {
}

6 Create an instance of the plug-in factory class that implements the IPluginFactory interface.
You create a factory instance by calling the createPluginFactory() method.
The solar system example adapter calls the createPluginFactory() method to create an instance of the
SolarSystemFactory class named factory.
public class SolarSystemAdapter implements IPluginAdaptor {
    private SolarSystemFactory factory;

    public IPluginFactory createPluginFactory(String sessionID, String username, String password, IPluginNotificationHandler notificationHandler)
        throws SecurityException, LoginException, PluginLicenseException {
        if (factory == null) {
            factory = new SolarSystemFactory();
        }
        return factory;
    }
}

7 Define the events that the plug-in adapter manages.
You can define the events to manage directly in the adapter implementation. However, the solar system
plug-in implementation defines the events in a separate class, named SolarSystemEventGenerator.
The SolarSystemEventGenerator class defines the following three methods:
• addPolicyElement(), to register an event with the Orchestrator policy engine.
• removePolicyElement(), to remove an event from the Orchestrator policy engine.
• generateFlareEvent(), to generate solar flare events, the magnitude of which Orchestrator
  monitors using a gauge.
SolarSystemAdapter defines the following getEventGenerator() method obtain an instance of
SolarSystemEventGenerator to provide access these methods and the events they manage.
private SolarSystemEventGenerator getEventGenerator() { return
    SolarSystemEventGenerator._solarSystemEventGenerator; }
If Orchestrator monitors the plugged-in application for events, you must register an instance of the IPluginEventPublisher interface by calling the IPluginAdaptor interface's registerEventPublisher() method.

The solar system example adapter creates the following IPluginEventPublisher instance and adds it to an Orchestrator policy by calling the SolarSystemEventGenerator class's addPolicyElement() method.

```java
public void registerEventPublisher(String type, String id,
        IPluginEventPublisher publisher) {
    getEventGenerator().addPolicyElement(type, id, publisher); }
```

Install any licenses that Orchestrator requires to access the plugged-in technology by calling the installLicenses() method to instantiate an array of PluginLicense objects.

```java
public void installLicenses(PluginLicense[] licenses) throws PluginLicenseException {
}
```

Pass the plug-in name to the plug-in.

The IPluginAdaptor interface's setPluginName() method gets the name from the vsbro.xml file, if necessary.

```java
public void setPluginName(String pluginName) {
}
```

Unregister an IPluginEventPublisher from the Orchestrator policy engine by calling.

The solar system example adapter unregisters the IPluginEventPublisher instance created in Step 8 and removes it from the Orchestrator policy engine by calling the unregisterEventPublisher() method.

```java
public void unregisterEventPublisher(String type, String id,
        IPluginEventPublisher publisher) {
    getEventGenerator().removePolicyElement(type, id, publisher); }
```

You have a plug-in adapter implementation that creates a plug-in factory, obtains licenses, passes the plug-in name to Orchestrator, and defines how to manage events that occur in the plugged-in technology.

**What to do next**

Create the plug-in factory implementation to find objects through the plug-in and define how to perform operations on them.

### Examine an Example Plug-In Factory

To create a plug-in factory, you create a Java class that implements the IPluginFactory interface from the Orchestrator plug-in API.

**Prerequisites**

The following procedure outlines the broad steps in the process to develop a plug-in factory class. For a description of the role of the plug-in factory and the other components of a plug-in, you must read "Components and Architecture of a Plug-In," on page 123.
Procedure

1. Create and save a Java file for the plug-in factory implementation called <YourApplicationName>Factory.java.
   In the solar system example, the factory class is called SolarSystemFactory.java.

2. Declare a package to contain the plug-in adapter and factory implementations.
   The solar system example declares the following package to contain the adapter and factory implementations:
   ```java
   package com.vmware.orchestrator.api.sample.solarsystem;
   ```

3. Import the Orchestrator plug-in API classes into the factory implementation with a Java `import` statement.
   ```java
   import ch.dunes.vso.sdk.api.*;
   ```

4. Import the solar system application with a Java `import` statement.
   ```java
   import com.vmware.solarsystem.*;
   ```

5. Import any other classes the adapter implementation requires.
   In the solar system example, the factory implementation requires the following classes:
   ```java
   import java.util.*;
   import org.apache.log4j.Logger;
   ```

6. Declare a public constructor that implements the `IPluginFactory` interface from the Orchestrator plug-in API.
   The solar system example adapter declares the `SolarSystemFactory` constructor.
   ```java
   public class SolarSystemFactory implements IPluginFactory {
   }
   ```

7. Set up logging so that Orchestrator can track the events that occur in the plugged-in technology.
   ```java
   private Logger log = Logger.getLogger(this.getClass());
   ```

8. Define how Orchestrator performs operations on the objects it finds through the plug-in by calling the
   `IPluginFactory` interface's `executePluginCommand()` method and defining the command behavior.
   The `SolarSystemFactory` example implementation does not support any commands, so does not
   implement the `executePluginCommand()` method.
   ```java
   public void executePluginCommand(String cmd) throws PluginExecutionException {
   }
   ```

9. Define how Orchestrator finds objects through the plug-in by their name and type by implementing the
   `IPluginFactory` interface's `find()` method.
   The `SolarSystemFactory` example implementation calls on methods defined by the
   `SolarSystemRepository` class to obtain Star, Planet, and Moon objects by their type and identifier.
   ```java
   public Object find(String type, String id) {
       log.debug("find: " + type + ", " + id);
       if (type.equals("Star")) {
           return SolarSystemRepository.getUniqueInstance().getStar();
       }
       else if (type.equals("Planet")) {
           return SolarSystemRepository.getUniqueInstance().getPlanetById(id);
       }
       else if (type.equals("Moon")) {
           return SolarSystemRepository.getUniqueInstance().getMoonById(id);
       }
   }
   ```
else if (type.equals("Galaxy")) {
    return null; // No object for galaxy defined yet
} else {
    throw new IndexOutOfBoundsException("Type " + type + " unknown for plugin SolarSystem");
}

10 Define query rules to find objects through the plug-in according to your own query criteria by implementing IPluginFactory interface's findAll() method.

The SolarSystemFactory example implementation returns a list of all the objects of a given type. For example, it can return a list of all objects of the Planet type.

    public QueryResult findAll(String type, String query) {
        log.debug("findAll: " + type + ", " + query);
        List list; // The list can contain any element from the plug-in
        if (type.equals("Star")) {
            list = new Vector();
            list.add(SolarSystemRepository.getUniqueInstance().getStar());
        } else if (type.equals("Planet")) {
            list = SolarSystemRepository.getUniqueInstance().getAllPlanets();
        } else if (type.equals("Moon")) {
            list = SolarSystemRepository.getUniqueInstance().getAllMoons();
        } else if (type.equals("Galaxy")) {
            list = new Vector();
        } else {
            throw new IndexOutOfBoundsException("Type " + type + " unknown for SolarSystem plug-in ");
        }
        return new QueryResult(list);
    }

The SolarSystemFactory class's implementation of the findAll() method does not define a custom query, so it returns a list of all the objects of the given type in a QueryResult object.

11 Define how Orchestrator finds objects through the plug-in by their relations to other objects by implementing the IPluginFactory interface's findRelation() method.

The SolarSystemFactory example implements findRelation() to return all the planets that orbit a given star, or all the moons that orbit a given planet.

    public List findRelation(String parentType, String parentId, String relationName) {
        log.debug("findRelation: " + parentType + ", " + parentId + ", " + relationName);
        if (parentId == null) {
            List<Star> list = new Vector<Star>();
            list.add(SolarSystemRepository.getUniqueInstance().getStar());
            return list;
        } else if (parentType.equals("Star")) {
            if (relationName.equals("OrbitingPlanets")) {
                return SolarSystemRepository.getUniqueInstance().getAllPlanets();
            }
        }
        return null;
    }

The findRelation() method returns a list of all the objects that match the given criteria.
else {
    throw new IndexOutOfBoundsException("Unknown relation name: " + relationName);
}
}

if (parentType.equals("Planet")) {
    if (relationName.equals("OrbitingMoons")) {
        Planet parentPlanet = SolarSystemRepository.getUniqueInstance().getPlanetById(parentId);
        if (parentPlanet != null) {
            return parentPlanet.getMoons();
        }
        return null;
    }
    else {
        throw new IndexOutOfBoundsException("Unknown relation name: " + relationName);
    }
} else {
    return null;
}

The SolarSystemFactory example's findRelation() method returns a list of objects that relate to the parent by the OrbitingPlanets and OrbitingMoons relation types defined in the vso.xml file.

12 Discover whether an object has children of a certain relation type by implementing the IPluginFactory interface's method.

The SolarSystemFactory example implements hasChildrenInRelation() to determine whether an object has children that relate to it by a given type.

public HasChildrenResult hasChildrenInRelation(String parentType, String parentId, String relationName) {
    return HasChildrenResult.Unknown;
}

The possible return values are Yes, No and Unknown.

You have a plug-in factory implementation that defines how Orchestrator finds objects through the plug-in and how it performs operations on those objects.

What to do next

Map the application objects to Orchestrator objects in the vso.xml file.

Map the Application in the vso.xml File

The vso.xml file defines how Orchestrator access and interacts with the plugged-in technology. It maps objects and operations in the plugged-in technology to Orchestrator objects and operations.

Prerequisites

The following procedures dissect the main sections of the vso.xml file for the solar system example application. You find the vso.xml file for the solar system example in the following location in the Orchestrator examples bundle:
For full descriptions of all the elements of a vso.xml file and all their attributes, see “Elements of the vso.xml Plug-In Definition File,” on page 128.

Procedure

1 Set Up the Plug-In on page 150
To create a plug-in you must point Orchestrator to the relevant XML schema definition, source files for the application and plug-in, define the behavior of the plug-in when Orchestrator starts, and provide a root object for the hierarchy of objects the plug-in exposes.

2 Define Finder Elements to Find Objects on page 151
To allow a Orchestrator to access objects in a plugged-in application, you must define how and where the plug-in finds those objects.

3 Define Enumerations on page 153
You can define enumerations to set values that apply to all objects of different types.

4 Map Events to Orchestrator API Methods on page 153
To allow Orchestrator to monitor objects in the plugged-in application and perform operations on them, you map the events the application defines to methods you add to the Orchestrator scripting API.

5 Map Objects to JavaScript Objects on page 154
To allow Orchestrator to call on objects and their methods in the plugged-in application, you map the objects and methods the application defines to methods you add to the Orchestrator scripting API.

Set Up the Plug-In
To create a plug-in you must point Orchestrator to the relevant XML schema definition, source files for the application and plug-in, define the behavior of the plug-in when Orchestrator starts, and provide a root object for the hierarchy of objects the plug-in exposes.

Procedure

1 Create and save a file named vso.xml.

2 Set up the <module> element to provide basic information about the plug-in, including a pointer the Orchestrator plug-in XML schema definition.

The <module> element in the vso.xml file for the solar system example sets the plug-in name to SolarSystem, sets the version number, and provides the path in the *.dar archive to the icon that represents this plug-in in the Orchestrator Inventory view and selection dialog boxes.

3 Provide a description of the plug-in in the <description> element.

The solar system example description is as follows.

<description>Example plug-in to a solar system application.</description>
4 Set the behavior of the plug-in when the Orchestrator server starts in the <installation> and <action> elements.

The solar system example sets the version mode to start the plug-in whenever a new version is detected, and provides the path to the package of Orchestrator components in the *.dar archive.

```xml
<installation mode="version">
  <action type="install-package" resource="packages/com.vmware.solarsystem.package" />
</installation>
```

5 Set the root of the hierarchy of object types in the <inventory> element.

The solar system plug-in defines the root of the hierarchy that represents the plug-in in the Orchestrator scripting API as an object of the type Galaxy. All the other solar system objects now relate to the Galaxy object.

```xml
<inventory type="Galaxy"/>
```

You defined setup the elements that identify the plug-in to Orchestrator.

**What to do next**

Define how Orchestrator finds objects through the plug-in using <finder> elements.

**Define Finder Elements to Find Objects**

To allow a Orchestrator to access objects in a plugged-in application, you must define how and where the plug-in finds those objects.

**Prerequisites**

You must have created up the vso.xml file and defined how Orchestrator identifies the plug-in.

**Procedure**

1 Set the data sources for the plug-in <finder> elements in the <finder-datasources> element.

The solar system plug-in vso.xml file sets the name of the data source to solar-datasource and points the <finder> elements to the SolarSystemAdapter class that creates the SolarSystemFactory instance.

```xml
<finder-datasources>
  <finder-datasource name="solar-datasource"
    adaptor-class="com.vmware.orchestrator.api.sample.solarsystem.SolarSystemAdapter"
    anonymous-login-mode="internal"/>
</finder-datasources>
```

2 Define how the plug-in finds objects in the plugged-in technology in <finder> elements.

The following extract from the solar system vso.xml file shows the <finder> element for objects of the type Star.

```xml
<finders>
  <finder type="Star" datasource="solar-datasource"
    java-class="com.vmware.solarsystem.Star"
    script-object="Star" image="images/sun_16x16.png">
    [...]
  </finder>
  [...]
</finders>
```
The `<finder>` element for Star objects obtain their data from the data source that the `<finder-datasource>` element defines. The Star object type represents instances of the `com.vmware.solarsystem.Star` class.

Plug-in `<finder>` elements feature sub-elements to map the relations, properties, and operations of the object.

3 Obtain or set the identifier of the object in the `<id>` element.

The solar system example obtains the identifier of the object by calling the `getId()` method that the solar system application's `CelestialBody` class defines.

```xml
/id accessor="getId()" />
```

4 Define the object's relations in the `<relations>` element.

The solar system example defines a relation named OrbitingPlanets to relate objects of the type Planet to the Star object this `<finder>` element finds.

```xml
<relations>
  <relation type="Planet" name="OrbitingPlanets"/>
</relations>
```

5 Set the hierarchy of objects in the `Inventory` according to their relation to the parent.

The solar system example places all objects related to Star objects type by the OrbitingPlanets relation immediately beneath it in the inventory hierarchy.

```xml
<inventory-children>
  <relation-link name="OrbitingPlanets"/>
</inventory-children>
```

6 Set the object's properties in the `<properties>` element.

The solar system example defines `name`, `circumference`, and `surfaceTemp` properties for all Star objects. The `bean-property` property allows Orchestrator to create get and set methods in the scripting API to obtain and set these properties. In this example, the scripting API will define `getCircumference`, `setCircumference`, `getSurfaceTemp`, and `setSurfaceTemp` methods.

```xml
<properties>
  <property display-name="Name" name="name"
    bean-property="name"/>
  <property display-name="Circumference" name="circumference"
    bean-property="circumference"/>
  <property display-name="Surface Temperature" name="surfaceTemp"
    bean-property="surfaceTemp"/>
</properties>
```

7 Set the object's events in the `<events>` element.

Events can be either gauges or triggers.

In the solar system example, the Star object defines a method to generate solar flares. The `<gauge>` element monitors the values of the flare events that Star objects generate.

```xml
<events>
  <gauge min-value="0" name="Flare" unit="number">
    <description>Magnitude of the flare</description>
  </gauge>
</events>
```

You defined a `<finder>` element to find objects in the plugged-in application.
What to do next
Define enumerations.

Define Enumerations
You can define enumerations to set values that apply to all objects of different types.

Prerequisites
You must have set up the plug-in and defined <finder> elements.

Procedure
1 Define an enumerations for object types in the <enumerations> element.
   The solar system example defines enumerations to set a PlanetCategory enumeration on Planet objects.
   
   <enumerations>
   <enumeration type="PlanetCategory">
   <description>Define the category of a Planet</description>
   [...]
   </enumeration>
   </enumerations>

2 Define entries for the enumerations, that apply values to objects in the given object category.
   The solar system example defines values that represent different types of planet.
   
   <entries>
   <entry id="gaz" name="Huge Gaz">Huge planet with only gaz atmosphere.
   No Physical core.</entry>
   <entry id="earth" name="Earth">You could live on this planet.</entry>
   <entry id="desert" name="Desert">Planet without water.</entry>
   <entry id="ice" name="Ice">Planet with water but completely frozen.</entry>
   <entry id="other" name="Other">Does not fit into any category.</entry>
   </entries>

You defined enumerations that can apply to all objects in a certain category.

What to do next
Map the application events to methods in the Orchestrator scripting API.

Map Events to Orchestrator API Methods
To allow Orchestrator to monitor objects in the plugged-in application and perform operations on them, you map the events the application defines to methods you add to the Orchestrator scripting API.

You identify the events to map in <scripting-objects><object> elements.

Prerequisites
You must have set up the plug-in, defined <finder> elements and enumerations.
Procedure

1. Map the Java classes that define the application events to a JavaScript object.

   The `SolarSystemEventGenerator` class defines the solar system application's events for the `SolarSystemAdapter` plug-in adapter implementation. The following code extract maps these events to a JavaScript object named `_SolarSystemEventGenerator`. By setting the `strict` attribute to `true`, Orchestrator can only call methods from the `_SolarSystemEventGenerator` JavaScript object.

   ```xml
   <scripting-objects>
     <object script-name="_SolarSystemEventGenerator"
             java-class="com.vmware.orchestrator.api.sample.solarsystem.SolarSystemEventGenerator"
             strict="true">
       <description>The entry point to generate events</description>
       [...]
     </object>
     [...]
   </scripting-objects>
   ```

2. (Optional) If necessary, denote the JavaScript object as a singleton object.

   ```xml
   <singleton script-name="SolarSystemEventGenerator" datasource="solar-datasource"/>
   ```

3. Map the methods in the plugged-in application to JavaScript methods you can call in the Orchestrator API in the `<object>` `<methods>` element.

   The solar system `Star` object defines a method to generate solar flare events. The code extract below maps this method to a JavaScript method of the same name, and sets its parameters in the JavaScript method.

   ```xml
   <methods>
     <method script-name="generateFlareEvent"
             java-name="generateFlareEvent">
       <description>Start a Solar Flare</description>
       <parameters>
         <parameter name="star" type="Star">The star which generates the event</parameter>
         <parameter name="magnitude" type="number">The magnitude of the flare</parameter>
       </parameters>
     </method>
   </methods>
   ```

You mapped the application’s events to methods in the Orchestrator scripting API.

**What to do next**

Map the application’s objects to JavaScript objects in the Orchestrator scripting API.

**Map Objects to JavaScript Objects**

To allow Orchestrator to call on objects and their methods in the plugged-in application, you map the objects and methods the application defines to methods you add to the Orchestrator scripting API.

You identify the objects to map in `<scripting-objects>` `<object>` elements.

**Prerequisites**

You must have set up the plug-in, defined `<finder>` elements and enumerations, and mapped the application’s events to methods in the Orchestrator API.
**Procedure**

1. **Map the Java classes that define the objects to JavaScript objects in `<object>` elements.**

   The `Star` objects in the solar system example defines methods to perform different operations. The following code extract from the solar system example `vso.xml` file maps the `Star` Java class from the application to a JavaScript object of the same name. By setting the `create` attribute to `false`, Orchestrator cannot create instances of this object. By setting the `strict` attribute to `true`, Orchestrator can only call methods from the `Star` JavaScript object.

   ```xml
   <object script-name="Star" java-class="com.vmware.solarsystem.Star"
     create="false" strict="true">
   <description>A star, center of a solar system</description>
   [...]
   </object>
   ``

2. **Map the object's attributes to JavaScript attributes in `<object>` elements.**

   The following code extract from the solar system example `vso.xml` file maps the `Star` object's Java attributes to JavaScript attributes of the same name.

   ```xml
   <attributes>
   <attribute script-name="id" java-name="id"
     return-type="string">The unique Id of the star</attribute>
   <attribute script-name="name" java-name="name"
     return-type="string">The name of the star</attribute>
   <attribute script-name="circumference" java-name="circumference"
     return-type="number">Circumference of the star</attribute>
   <attribute script-name="temperature" java-name="surfaceTemp"
     return-type="number">The temperature on the star's surface</attribute>
   </attributes>
   ``

3. **Map the object's methods to JavaScript methods in `<object>` elements.**

   The following code extract from the solar system example `vso.xml` file maps one of the `Star` object's Java methods to JavaScript methods of the same name.

   ```xml
   <methods>
   <method script-name="addPlanet" java-name="addPlanet">
   <description>Add new planet to the solar system</description>
   <parameters>
   <parameter type="Planet" name="planet">The planet to add</parameter>
   </parameters>
   </method>
   </methods>
   ``

You mapped the application's objects to JavaScript objects in the Orchestrator scripting API.

**What to do next**

Create the `*.dar` archive that contains the plug-in components.

**Create the Plug-In `*.dar` Archive**

The final stage in the creation of a plug-in is to create the `*.dar` that you import into Orchestrator.

The `*.dar` archive is a standard `*.jar` Java archive that you rename to `*.dar`. The `*.dar` archive must adhere to a standard file and folder structure.
Prerequisites
You have an application to plug into Orchestrator, have created the adaptor and factory implementations, and have mapped the application to Orchestrator objects in the vso.xml file.

Procedure
1. Create a working directory in which to create the *.dar archive.
2. Create a named called VSO-INF at the root of the working directory.
3. Copy the vso.xml file into VSO-INF.
4. Create a folder named lib at the root of the working directory.
5. Copy the JAR files containing the classes of the application to plug in and the classes of the plug-in adaptor and factory implementations into lib.
6. Create a folder named resources at the root of the working directory.
7. Create a folder named images in the resources folder.
8. Copy icons into resources/images.
   These icons represent the different objects of the plugged-in application in the Orchestrator Inventory view and selection dialog boxes.
9. Create a folder named packages in the resources folder.
10. Copy Orchestrator packages into resources/packages.
    These packages can contain contain workflows, actions, policies, and so on, that interact with the plugged-in application.
11. Create a folder named web-content at the root of the working directory.
12. Copy the components of any Web views you have created for the application into the web-content folder.
13. Create a Java archive that contains all the preceding folders and files.
    For example, at the command line you run the following jar command.
    
    ```bash
    jar -cf myDarFile VSO.INF lib resources web-content
    ```
15. Import the plug-in into the Orchestrator server.
    You can import the plug-in in two ways:
    - Copy the *.dar archive into the Orchestrator plug-ins folder, at the following location.
      `install-directory\app-server\server\vmo\plugins`
    - Use the Orchestrator configuration interface. See the VMware vCenter Orchestrator Administration Guide for information about using the configuration interface to import a plug-in.

You created the *.dar archive that contains a plug-in, and imported it into Orchestrator.
Example 6-1. Contents of the Solar System *.dar Archive

To illustrate the contents and structure of a *.dar archive, the solar system example vmware-vmosdk-solarsystem.dar archive contains the following folders and files.

- /lib, containing the following JAR archives:
  - vmware-solarsystem.jar, containing the solar system application.
  - vmware-vmosdk-solarsystem.jar, containing the classes of the plug-in adapter and factory implementations for the solar system application.

- /resources
  - /images, containing icons that represent the different objects of the solar system application in the Orchestrator Inventory view.

- /packages, containing an Orchestrator package named com.vmware.solarsystem.package. The package contains workflows, policies, actions and the Web view that allow Orchestrator to interact with the solar system application.

- /VSO-INF/vso.xml, the XML file that maps the solar system application to Orchestrator objects.

What to do next

You can access the objects of the plugged-in application in the Inventory to perform operations upon them. You can also use the objects and methods that you mapped to the Orchestrator scripting API to create workflows, actions, policies, Web views, and so on, to interact with the objects through the plug-in. When you have created the workflows, actions, policies, and Web views, you can add them as packages to the plug-in *.dar file.

Orchestrator Plug-In API Reference

The Orchestrator plug-in API defines Java interfaces and classes to implement and extend when you develop the IPluginAdaptor and IPluginFactory implementations to create a plug-in.

IDynamicFinder Interface

The IDynamicFinder interface returns the ID and properties of a finder programmatically, instead defining the ID and properties in the vso.xml file.

The IDynamicFinder Interface defines the following methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getIdAccessor(java.lang.String type)</td>
<td>java.lang.String</td>
<td>Provides an OGNL expression to obtain an object ID programmatically.</td>
</tr>
<tr>
<td>getProperties(java.lang.String type)</td>
<td>java.util.List&lt;SDKFinderProperty&gt;</td>
<td>Provides a list of object properties programmatically.</td>
</tr>
</tbody>
</table>

IPluginAdaptor Interface

You implement the IPluginAdaptor interface to manage plug-in factories, events and watchers. The IPluginAdaptor interface defines an adapter between a plug-in and the Orchestrator server.

IPluginAdaptor instances are responsible for session management. The IPluginAdaptor Interface defines the following methods.
### Method

**addWatcher(PluginWatcher watcher)**

**Returns**: Void

**Description**: Adds a watcher to monitor for a specific event


**Returns**: IPluginFactory

**Description**: Creates an IPluginFactory instance. The Orchestrator server uses the factory to obtain objects from the plugged-in technology by their ID, by their relation to other objects, and so on. The session ID allows you to identify a running session. For example, a user could log into two different Orchestrator clients and run two sessions simultaneously. Similarly, starting a workflow creates a session that is independent from the client in which the workflow started. A workflow continues to run even if you close the Orchestrator client.

**installLicenses(PluginLicense[] licenses)**

**Returns**: Void

**Description**: Installs the license information for standard plug-ins that VMware provides

**registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)**

**Returns**: Void

**Description**: Sets triggers and gauges on an element in the inventory

**removeWatcher(java.lang.String watcherId)**

**Returns**: Void

**Description**: Removes a watcher

**setPluginName(java.lang.String pluginName)**

**Returns**: Void

**Description**: Gets the plug-in name from the vso.xml file

**setPluginPublisher(IPluginPublisher pluginPublisher)**

**Returns**: Void

**Description**: Sets the publisher of the plug-in

**uninstallPluginFactory(IPluginFactory plugin)**

**Returns**: Void

**Description**: Uninstalls a plug-in factory.

**unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)**

**Returns**: Void

**Description**: Removes triggers and gauges from an element in the inventory

### IPluginEventPublisher Interface

The IPluginEventPublisher interface publishes gauges and triggers on an event notification bus for Orchestrator policies to monitor.

The IPluginEventPublisher Interface defines the following methods.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)</td>
<td>Void</td>
<td>Publish a gauge on the event notification bus</td>
</tr>
<tr>
<td>pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)</td>
<td>Void</td>
<td>Publish a trigger on event notification bus</td>
</tr>
</tbody>
</table>

### IPluginFactory Interface

The IPluginAdaptor returns IPluginFactory instances. IPluginFactory instances run commands in the plugged-in application, and finds objects through the plug-in, upon which to perform Orchestrator operations.

The IPluginFactory Interface defines the following methods.
### IPluginNotificationHandler Interface

The IPluginNotificationHandler defines methods to notify Orchestrator of different types of event that occur on the objects Orchestrator accesses through the plug-in.

The IPluginNotificationHandler Interface defines the following methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSessionID()</td>
<td>java.lang.String</td>
<td>Returns the current session ID</td>
</tr>
<tr>
<td>notifyElementDeleted(java.lang.String type, java.lang.String id)</td>
<td>Void</td>
<td>Notifies the system that an object with the given type and ID has been deleted</td>
</tr>
<tr>
<td>notifyElementInvalidate(java.lang.String type, java.lang.String id)</td>
<td>Void</td>
<td>Notifies the system that an object's relations have changed</td>
</tr>
<tr>
<td>notifyElementUpdated(java.lang.String type, java.lang.String id)</td>
<td>Void</td>
<td>Notifies the system that an object's attributes have been modified</td>
</tr>
</tbody>
</table>

### IPluginPublisher Interface

The IPluginPublisher interface publishes a watcher event on an event notification bus for long-running workflow Wait Event elements to monitor.

The IPluginPublisher Interface defines the following method.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushWatcherEvent(java.lang.String id, java.util.Properties properties)</td>
<td>Void</td>
<td>Publish a watcher event on event notification bus</td>
</tr>
</tbody>
</table>
PluginExecutionException Class

The PluginExecutionException class returns an error message if the plug-in encounters an exception when it runs.

```java
public class PluginExecutionException
extends java.lang.Exception
implements java.io.Serializable
```

- Constructor: `PluginExecutionException(java.lang.String message)`
- The PluginExecutionException class inherits the following methods from class `java.lang.Throwable`:
  - `fillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`, `printStackTrace`, `printStackTrace`, `setStackTrace`, `toString`, `fillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`
- The PluginExecutionException class inherits the following methods from class `java.lang.Object`:
  - `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

PluginOperationException Class

The PluginOperationException class handles errors encountered during a plug-in operation.

```java
public class PluginOperationException
extends java.lang.RuntimeException
implements java.io.Serializable
```

- Constructor: `PluginOperationException(java.lang.String message)`
- The PluginOperationException class inherits the following methods from class `java.lang.Throwable`:
  - `fillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`, `printStackTrace`, `printStackTrace`, `setStackTrace`, `toString`
- The PluginOperationException class inherits the following methods from class `java.lang.Object`:
  - `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

PluginTrigger Class

The PluginTrigger class defines a trigger module that obtains information about event triggers in the plugged-in technology, on behalf of a long-running workflow Wait Event element.

```java
public class PluginTrigger
extends java.lang.Object
implements java.io.Serializable
```

The PluginTrigger class defines the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getModuleName()</code></td>
<td><code>java.lang.String</code></td>
<td>Obtains the trigger module</td>
</tr>
<tr>
<td><code>getProperties()</code></td>
<td><code>java.util.Properties</code></td>
<td>Obtains the trigger properties</td>
</tr>
</tbody>
</table>
### PluginTrigger Class

The `PluginTrigger` class inherits the following methods from class `java.lang.Object`:
- `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getSdkId()</code></td>
<td><code>java.lang.String</code></td>
<td>Obtains the ID of the SDK that built the module.</td>
</tr>
<tr>
<td><code>getSdkType()</code></td>
<td><code>java.lang.String</code></td>
<td>Obtains the type of the SDK that built the module.</td>
</tr>
<tr>
<td><code>getTimeout()</code></td>
<td><code>Long</code></td>
<td>Obtains the trigger timeout period.</td>
</tr>
<tr>
<td><code>setModuleName(java.lang.String moduleName)</code></td>
<td><code>Void</code></td>
<td>Sets a trigger module name.</td>
</tr>
<tr>
<td><code>setProperties(java.util.Properties properties)</code></td>
<td><code>Void</code></td>
<td>Sets trigger module properties.</td>
</tr>
<tr>
<td><code>setSdkId(java.lang.String sdkId)</code></td>
<td><code>Void</code></td>
<td>Sets the ID of the trigger SDK.</td>
</tr>
<tr>
<td><code>setSdkType(java.lang.String sdkType)</code></td>
<td><code>Void</code></td>
<td>Sets the type of the trigger SDK.</td>
</tr>
<tr>
<td><code>setTimeout(long timeout)</code></td>
<td><code>Void</code></td>
<td>Sets a trigger module timeout.</td>
</tr>
</tbody>
</table>

- **Constructors:**
  - `PluginTrigger()`
  - `PluginTrigger(java.lang.String moduleName, long timeout, java.lang.String sdkType, java.lang.String sdkId)`

### PluginWatcher Class

The `PluginWatcher` class watches a trigger module for a defined event in the plugged-in technology on behalf of a long-running workflow Wait Event element.

```java
class PluginWatcher extends java.lang.Object implements java.io.Serializable {
  // The PluginWatcher class defines the following methods:

  Method | Returns               | Description                                      |
  -----------------------------------------------|
  `getId()` | `java.lang.String`    | Obtains the ID of the trigger.                   |
  `getModuleName()` | `java.lang.String`    | Obtains the trigger module name.                 |
  `getTimeoutDate()` | `Long`                | Obtains the trigger timeout date.                |
  `getTrigger()` | `Void`                | Obtains a trigger.                               |
  `setId(java.lang.String id)` | `Void`                | Sets the ID of the trigger.                      |
  `setTimeoutDate()` | `Void`                | Sets the trigger timeout date.                   |
```

- **Constructor:** `PluginWatcher(PluginTrigger trigger)`

- The `PluginWatcher` class inherits the following methods from class `java.lang.Object`:
  - `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`
QueryResult Class

The QueryResult class contains the results of a find query made on the objects Orchestrator accesses through the plug-in.

```java
public class QueryResult
    extends java.lang.Object
    implements java.io.Serializable
```

The `totalCount` value can be greater than the number of elements the QueryResult returns, if the total number of results found exceeds the number of results the query returns. The number of results the query returns is defined in the query syntax in the vso.xml file.

The QueryResult class defines the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addElement(java.lang.Object element)</td>
<td>Void</td>
<td>Adds an element to the QueryResult</td>
</tr>
<tr>
<td>addElements(java.util.List elements)</td>
<td>Void</td>
<td>Adds a list of elements to the QueryResult</td>
</tr>
<tr>
<td>getElements()</td>
<td>java.util.List</td>
<td>Obtains elements from the plugged in application</td>
</tr>
<tr>
<td>getTotalCount()</td>
<td>Long</td>
<td>Obtains a count of all the elements available in the plugged in technology</td>
</tr>
<tr>
<td>isPartialResult()</td>
<td>Boolean</td>
<td>Determines whether the result obtained is complete</td>
</tr>
<tr>
<td>removeElement(java.lang.Object element)</td>
<td>Void</td>
<td>Removes an element from the plugged in technology</td>
</tr>
<tr>
<td>setElements(java.util.List elements)</td>
<td>Void</td>
<td>Sets elements in the plugged in technology</td>
</tr>
<tr>
<td>setTotalCount(long totalCount)</td>
<td>Void</td>
<td>Sets the total number of elements available in the plugged in technology</td>
</tr>
</tbody>
</table>

- Constructors:
  - QueryResult()
  - QueryResult(java.util.List ret)
    ```java
totalCount = ret.size()
```
  - QueryResult(java.util.List elements, long totalCount)

  The QueryResult class inherits the following methods from class java.lang.Object:
  ```java
close, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait
```

SDKFinderProperty Class

The SDKFinderProperty class defines methods to obtain and set properties in the objects found in the plugged in technology by the Orchestrator finder objects. The IDynamicFinder.getProperties method returns SDKFinderProperty objects.

```java
public class SDKFinderProperty
    extends java.lang.Object
```

The SDKFinderProperty class defines the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAttributeName()</td>
<td>java.lang.String</td>
<td>Obtains an object attribute name</td>
</tr>
<tr>
<td>getBeanProperty()</td>
<td>java.lang.String</td>
<td>Obtains properties from a Java bean</td>
</tr>
</tbody>
</table>
**Method** | **Returns** | **Description**  
--- | --- | ---  
getDescription() | java.lang.String | Obtains an object description  
getDisplayName() | java.lang.String | Obtains an object display name  
getPossibleResultType() | java.lang.String | Obtains the possible types of result the finder returns  
getPropertyAccessor() | java.lang.String | Obtains an object property accessor  
getPropertyAccessorTree() | java.lang.Object | Obtains an object property accessor tree  
isHidden() | Boolean | Shows or hides the object  
isShowInColumn() | Boolean | Shows or hides the object in the database column  
isShowInDescription() | Boolean | Shows or hides the object description  
setAttributeName(java.lang.String attributeName) | Void | Sets an object attribute name  
setBeanProperty(java.lang.String beanProperty) | Void | Sets properties in a Java bean  
setDescription(java.lang.String description) | Void | Sets an object description  
setDisplayValue(java.lang.String displayName) | Void | Sets an object display name  
 setHidden(boolean hidden) | Void | Show or hide the object  
setPossibleResultType(java.lang.String possibleResultType) | Void | Sets the possible types of result the finder returns  
setPropertyAccessor(java.lang.String propertyAccessor) | Void | Sets an object property accessor  
setPropertyAccessorTree(java.lang.Object propertyAccessorTree) | Void | Sets an object property accessor tree  
 setShowInColumn(boolean showInTable) | Void | Show or hide the object in the database column  
 setShowInDescription(boolean showInDescription) | Void | Show or hide the object description

- **Constructor:** SDKFinderProperty(java.lang.String attributeName, java.lang.String displayName, java.lang.String beanProperty, java.lang.String propertyAccessor)
- The SDKFinderProperty class inherits the following methods from class java.lang.Object: clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### HasChildrenResult Enumeration

The HasChildrenResult Enumeration declares whether a given parent has children. The IPluginFactory.hasChildrenInRelation method returns HasChildrenResult objects.

```java
default enum HasChildrenResult extends java.lang.Enum<HasChildrenResult> implements java.io.Serializable
```
The `HasChildrenResult` Enumeration defines the following constants:

- `public static final HasChildrenResult Yes`
- `public static final HasChildrenResult No`
- `public static final HasChildrenResult Unknown`

The `HasChildrenResult` Enumeration defines the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getValue()</td>
<td>int</td>
<td>Returns one of the following values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>HasChildrenResult</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Parent has children</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1 Parent has no children</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Unknown, or invalid parameter</td>
</tr>
<tr>
<td>valueOf(java.lang.String name)</td>
<td>static HasChildrenResult</td>
<td>Returns an enumeration constant of this type with the specified name. The String must match exactly an identifier used to declare an enumeration constant of this type. Do not use whitespace characters in the enumeration name.</td>
</tr>
<tr>
<td>values()</td>
<td>static HasChildrenResult[]</td>
<td>Returns an array containing the constants of this enumeration type, in the order they are declared. This method can iterate over constants as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);</code></td>
</tr>
</tbody>
</table>

- The `HasChildrenResult` Enumeration inherits the following methods from class `java.lang.Enum`:
  - clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf
- The `HasChildrenResult` Enumeration inherits the following methods from class `java.lang.Object`:
  - getClass, notify, notifyAll, wait, wait, wait

**ScriptingAttribute Annotation Type**

The `ScriptingAttribute` annotation type annotates an attribute from an object in the plugged in technology for use as a property in scripting.

```java
@Retention(value=RUNTIME)
@Target(value={METHOD,FIELD})
public @interface ScriptingAttribute

The `ScriptingAttribute` annotation type has the following value:

```java
public abstract java.lang.String value
```

**ScriptingFunction Annotation Type**

The `ScriptingFunction` annotation type annotates a method for use as a property in scripting.

```java
@Retention(value=RUNTIME)
@Target(value={METHOD,CONSTRUCTOR})
public @interface ScriptingFunction

The `ScriptingFunction` annotation type has the following value:

```java
public abstract java.lang.String value
```
ScriptingParameter Annotation Type

The `ScriptingParameter` annotation type annotates a parameter for use as a property in scripting.

```java
@Retention(value=RUNTIME)
@Target(value=PARAMETER)
public @interface ScriptingParameter
```

The `ScriptingParameter` annotation type has the following value:

```java
public abstract java.lang.String value
```
VMware vCenter Orchestrator provides a Web services API so that you can develop applications to access workflows through Web services. The main purpose of the Orchestrator Web service is to start workflows and to retrieve their output parameters through a network or the Web.

The Web service API provides a set of objects, or Web service definition language (WSDL) type definitions, and a set of methods, or Web service operations, that obtain workflows, run workflows, refresh workflow states, and obtain their output parameter values. The Web service API also allows you to implement tree viewers, based on the relations between objects obtained from plug-ins. The API has few complex object types and relatively few operations.

**NOTE** To help understand how Orchestrator implements Web services, familiarize yourself with the Web services API for your development framework, for example Java or .Net.

This chapter includes the following topics:

- “Writing a Web Service Client Application,” on page 167
- “Web Service API Objects,” on page 182
- “Web Service API Operations,” on page 187

**Writing a Web Service Client Application**

Most applications that use the Orchestrator Web Service API have a common structure. To create Orchestrator Web service client applications, you perform a standard sequence of tasks.

**Process for Creating an Orchestrator Web Service Client Application**

Developing a Web services client application follows a broad sequence of stages.  

*Figure 7-1* shows how to create a typical Orchestrator Web service client application.
Figure 7-1. Process for Creating Orchestrator Web Service Applications

Create a VSOWebControl object to connect to the Web service

HTTP

HTTPS

(Optional) check the connection to the server using echoWorkflow

(Optional) check for plug-ins using getAllPlugins

If necessary, find objects to execute workflows upon

Use find to locate an object of a particular type, that matches a particular query criterion

Use findForId to locate an object with a particular ID number

Use hasChildrenInRelation and findRelation to find children of a particular relation type

Find a workflow

Use getAllWorkflows to list all workflows

Use getWorkflowsWithId to find workflows with a particular name

Use getWorkflowsForId to find a workflow based on its unique ID

(Optional) check whether the current user has rights to read, execute, or edit the workflow using hasRights

Define the workflow’s inParameters

Execute the workflow using executeWorkflow, which creates a WorkflowToken

Perform different actions while the WorkflowToken executes

Check the status of the workflow with getWorkflowTokenStatus

Find other WorkflowToken objects using getWorkflowTokenForId

Provide runtime input with answerWorkflowInput

Cancel the workflow using cancelWorkflow

Send a custom event using sendCustomEvent

When the WorkflowToken completes, check the results with getWorkflowTokenResult

Display, process, or otherwise act upon the results of the workflow

Follow the broad stages of development illustrated to create Orchestrator Web services client applications that satisfy most of your requirements.
Web Service Endpoint

The Web service endpoint is the port upon which you connect a Web service client to the Orchestrator server.

You connect to the Orchestrator Web service's endpoint at the following URL, in which `<Orchestrator_server>` is the IP address of the host on which the Orchestrator server is running.

http://<Orchestrator_server>:8280/vmware-vmo-webcontrol/webservice

The Web service runs over HTTP or HTTPS on port 8280 or 8281 of the Orchestrator server. Access to the Web service API requires a valid username and password on the Orchestrator server. Because every access to the service is authenticated separately, a secure HTTPS connection is not strictly necessary. However, the Web service sends passwords over the network without encryption, so use a secure HTTPS connection if security is an issue for your applications.

NOTE Networks secured by HTTPS access the Web service endpoint on port 8281. In your network, the port number might be different from the defaults of 8280 or 8281.

Generating the Orchestrator Web Service Stubs

You generate the Web service objects from the Orchestrator WSDL description file to create the client and server stubs for the Web service application.

Orchestrator publishes the WSDL description file at the following location.


You generate the Web service client and server stubs by using a Java or .Net code generator. The Orchestrator Web service supports all WSDL 1.1 parsers. Generating the Web service provides the following objects.

**VSOWebControl**

The Web service defines a WSDL port type named `VSOWebControl`, through which you access all the Orchestrator Web service operations.

**WebServiceStub**

The Web service defines client and server side stubs that the application uses to start the Web service.

**VSOWebControlProxy**

The Web service provides access to the Orchestrator Web service operations through a proxy.

**VSOWebControlService**

The `VSOWebControlService` service is a remote procedure call (RPC) Service implementation.

**VSOWebControlServiceLocator**

The `VSOWebControlServiceLocator` service extends `VSOWebControlService` to provide the following operations.

- `getwebserviceAddress` obtains the Web service's endpoint URL.
- `getwebservice` obtains the client-side stub for the Web service application, and instantiates the `VSOWebControl` port type object with the appropriate endpoint URL.
Create a Web Service Client

You can use the Orchestrator Web service API to create a Web service client to connect to the Orchestrator Server. The Web service connection allows you to access workflows in the Orchestrator server and perform operations on them.

Prerequisites

You must have generated the Web service client and server stubs from the Orchestrator WSDL definition by using a code generator.

Procedure

1. **Connect to the Orchestrator Web Service** on page 171
   Web service applications establish connections to the Orchestrator server through simple object access protocol (SOAP) binding, using either the HTTP or HTTPS protocols.

2. **Find Objects in the Orchestrator Server** on page 171
   To perform any useful task with a workflow, you must find the objects on which the workflow will run. The Orchestrator Web service API provides functions for finding objects of all types in the VMware Infrastructure inventory.

3. **Find Objects by Using the find Operation** on page 172
   You can use the `find` operation to find objects of any type that match a particular search criterion, that you set in the `query` parameter.

4. **Find Objects by Using the findForId Operation** on page 173
   You can use the `findForId` operation to find an object if you know a specific object's unique ID.

5. **Find Objects by Using the findRelation Operation** on page 174
   You can use the `findRelation` operation to locate the children of a particular object.

6. **Find Workflows in the Orchestrator Server** on page 175
   When you have found the objects with which to interact, you must find the workflows that perform these interactions.

7. **Find Workflows by Using the getAllWorkflows Operation** on page 176
   The `getAllWorkflows` operation lists all workflows that a user can access as an array of `Workflow` objects.

8. **Find Workflows by Using the getWorkflowsWithName Operation** on page 176
   If you know the name of a particular workflow, as it is defined in the Orchestrator client, the Web service application can obtain this workflow using its name or part of its name.

9. **Find Workflows by Using the getWorkflowForID Operation** on page 176
   If you know a particular workflow ID, a Web service application can obtain this workflow by using the `getWorkflowForID` operation.

10. **Run Workflows from a Web Service Client** on page 177
    The main purpose of a Web services client is to run workflows across a network.

11. **Interact with a Workflow While it Runs** on page 178
    After the workflow starts, the Web services client can perform various actions in response to events while the workflow is running.

12. **Obtain Workflow Results** on page 180
    After the workflow completes its run, you can retrieve the results by calling the `getWorkflowTokenResult()` operation.
**Connect to the Orchestrator Web Service**

Web service applications establish connections to the Orchestrator server through simple object access protocol (SOAP) binding, using either the HTTP or HTTPS protocols.

**Prerequisites**

You must have generated the Orchestrator Web service client and server stubs from the Orchestrator WSDL definition. You must create a Web service client application class that implements the VS0WebControl interface.

**Procedure**

1. In your Web service client application class, create a VS0WebControl instance that connects to the Web service endpoint.

   You can either create an unsecured connection using HTTP, or a secure connection using HTTPS. The default HTTP port is 8280 and the default HTTPS port is 8281. The URL is also a default.

   - The following example shows how to create an HTTP connection to the Web service.

     ```java
     String urlprefix = "http://10.0.0.1:8280" ;
     URL url = new URL(urlprefix + "/vmware-vmo-webcontrol/webservice");
     vsoWebControl = new VS0WebControlServiceLocator().getwebservice(url);
     ```

   - The following example shows how to create an HTTPS connection to the Web service.

     ```java
     String urlprefix = "https://10.0.0.1:8281" ;
     URL url = new URL(urlprefix + "/vmware-vmo-webcontrol/webservice");
     vsoWebControl = new VS0WebControlServiceLocator().getwebservice(url);
     ```

2. Check the server connections by calling the `echo` operation.

   The following example shows how you can call the `echo` operation.

   ```java
   vsoWebControl.echo(string);
   ```

   The preceding call to the `echo` operation returns the String object that you provided as an argument.

3. (Optional) Check what plug-ins are running the Orchestrator server by calling the `getAllPlugins` operation.

   The following example shows how you can call the `getAllPlugins` operation.

   ```java
   ModuleInfo[] modules = vsoWebControl.getAllPlugins(username, password);
   ```

   The preceding call to the `getAllPlugins` operation returns an array of ModuleInfo objects, each of which contains the name and version information about a plug-in running in the Orchestrator server.

You created a connection to the Orchestrator Web service, verified the connection, and established what technologies plug-in to the Orchestrator server.

**What to do next**

Find objects in the Orchestrator server through the Web service connection.

**Find Objects in the Orchestrator Server**

To perform any useful task with a workflow, you must find the objects on which the workflow will run. The Orchestrator Web service API provides functions for finding objects of all types in the VMware Infrastructure inventory.

Workflows typically run on objects in the vCenter Server. Workflows can also run on objects from outside the vCenter Server by accessing them through plug-ins.
The operations that the Web service API defines for finding objects are as follows.

- `find`
- `findForId`
- `findRelation`
- `hasChildrenInRelation`

All of the operations that find objects return `FinderResult` objects, either individually, as an array, or embedded in a `QueryResult` object.

### Find Objects by Using the `find` Operation

You can use the `find` operation to find objects of any type that match a particular search criterion, that you set in the `query` parameter.

The `vso.xml` file of the plug-in through which you access the object defines the syntax of the `query` parameter.

### Prerequisites

You must have created a connection to the Orchestrator Web services endpoint in your Web service client application class.

### Procedure

1. Create a `QueryResult` object by calling the `find` operation on an object.

   The following code example shows how an application can call the `find` operation to find out how many virtual machines are accessible by a particular user through the vCenter Server 4.0 plug-in.

   ```java
   QueryResult queryResult = vsoWebControl.find("VC:VirtualMachine", null, <username>, <password>);
   if (queryResult != null) {
       System.out.println("Found " + queryResult.getTotalCount() + " obs.");
       FinderResult[] elts = queryResult.getElements();
       finderResult = elts[0];
       displayFinderResult(finderResult);
   } else {
       System.out.println("Found nothing");
   }
   ```

   According to the query syntax defined by the vCenter Server 4.0 plug-in, setting the `query` parameter to `null` returns the list of all of the objects of the type specified by the first parameter. The preceding code example performs the following tasks.

   - Gets the list of any `VC:VirtualMachine` objects in the library.
   - Calls the `QueryResult` object's `getTotalCount` operation to obtain the total number of `VC:VirtualMachine` objects found and print the value.
Calls the QueryResult object's getElements operation to obtain the details of the objects found as an array of FinderResult objects.

Passes the array of FinderResult objects to the internal method displayFinderResult, which extracts the information.

2 Extract the results from a FinderResult object.

To show, interpret, or process the results in the FinderResult objects that the find operation returns, you must convey these results to the Web service application.

The following example shows how to extract the results returned in a FinderResult object.

```java
public static void displayFinderResult(FinderResult finderResult) {
    if (finderResult != null) {
        System.out.println("Finder result is of type '" + finderResult.getType() + ", id '" + finderResult.getId() + ", and uri '" + finderResult.getDunesUri() + ");
        System.out.println("And has properties :");
        Property[] props = finderResult.getProperties();
        if (props != null) {
            for (int ii = 0; ii < props.length; ii++) {
                System.out.println("\t" + props[ii].getName() + "=" + props[ii].getValue());
            }
        }
    }
}
```

The example defines an internal method, displayFinderResult, which takes a FinderResult object and obtains and shows its type, ID, the URI at which it is located, and its properties. You can use the URI to call workflows, for example. The getType, getId, getProperties and getDunesUri methods are defined by the FinderResult object.

You found objects in the Orchestrator server that the Web service client can access and and run workflows upon.

What to do next

Implement Web service operations in the client application to find workflows in the Orchestrator server.

Find Objects by Using the findForId Operation

You can use the findForId operation to find an object if you know a specific object's unique ID.

To use findForId, you match a specific type of object to its identifier.

Prerequisites

You must have created a connection to the Orchestrator Web services endpoint in your Web service client application class.

Procedure

1 Create a FinderResult object by calling the findForId operation on an object.

```java
finderResult = vsoWebControl.findForId("VC:VirtualMachine", "vcenter/vm-xx", username, password);
```
In the preceding example, `vcenter/vm-xx` is the ID of a virtual machine object that the `findForID` operation finds.

The `findForID` operation returns a `FinderResult` instance directly, rather than creating an array of `FinderResult` objects like `find`. Finding objects by their unique ID always returns only one object.

2. Extract the results from a `FinderResult` object.

To show, interpret, or process the results in the `FinderResult` objects that the `find` operation returns, you must convey these results to the Web service application.

The following example shows how to extract the results returned in a `FinderResult` object.

```java
public static void displayFinderResult(FinderResult finderResult) {
    if (finderResult != null) {
        System.out.println("Finder result is of type '" +
                      finderResult.getType()
                      + ", id " + finderResult.getId()
                      + ", and uri " +
                      finderResult.getDunesUri() + ");
        System.out.println("And has properties : ");
        Property[] props = finderResult.getProperties();
        if (props != null) {
            for (int ii = 0; ii < props.length; ii++) {
                System.out.println("t" + props[ii].getName() + "="
                                + props[ii].getValue());
            }
        }
    }
}
```

The example defines an internal method, `displayFinderResult`, which takes a `FinderResult` object and obtains and shows its type, ID, the URI at which it is located, and its properties. You can use the URI to call workflows, for example. The `getType`, `getId`, `getProperties` and `getDunesUri` methods are defined by the `FinderResult` object.

You found objects in the Orchestrator server that the Web service client can access and and run workflows upon.

**What to do next**

Implement Web service operations in the client application to find workflows in the Orchestrator server.

**Find Objects by Using the `findRelation` Operation**

You can use the `findRelation` operation to locate the children of a particular object.

The `findRelation` operation returns an array of `FinderResult` objects that correspond to the children of a particular object.

**Prerequisites**

You must have created a connection to the Orchestrator Web services endpoint in your Web service client application class.

**Procedure**

1. Create an array of `FinderResult` objects by calling the `findRelation` operation on an object.

   ```java
   FinderResult[] results = vsoWebControl.findRelation("VC:ComputeResource",
   "vcenter/domain-s114", "getResourcePool()", "username", "password");
   ```
The preceding example returns an array of FinderResult objects that match the following criteria.

- The parent element is of the type VC:ComputeResource.
- The parent element's ID is vchost/domain-s114.
- The returned children are related to the parent by the getResourcePool relation, defined by the Orchestrator vCenter Server 4 plug-in.

2 Extract the results from a FinderResult object.

To show, interpret, or process the results in the FinderResult objects that the find operation returns, you must convey these results to the Web service application.

The following example shows how to extract the results returned in a FinderResult object.

```java
public static void displayFinderResult(FinderResult finderResult) {
    if (finderResult != null) {
        System.out.println("Finder result is of type ", finderResult.getType() + ", id " + finderResult.getId() + " and uri ");
        System.out.println("And has properties : ");
        Property[] props = finderResult.getProperties();
        if (props != null) {
            for (int ii = 0; ii < props.length; ii++) {
                System.out.println("\t" + props[ii].getName() + "=
                                + props[ii].getValue());
            }
        }
    }
}
```

The example defines an internal method, displayFinderResult, which takes a FinderResult object and obtains and shows its type, ID, the URI at which it is located, and its properties. You can use the URI to call workflows, for example. The getType, getId, getProperties and getDunesUri methods are defined by the FinderResult object.

You found objects in the Orchestrator server that the Web service client can access and and run workflows upon.

**What to do next**

Implement Web service operations in the client application to find workflows in the Orchestrator server.

**Find Workflows in the Orchestrator Server**

When you have found the objects with which to interact, you must find the workflows that perform these interactions.

The Orchestrator Web service API includes the following operations to find all the workflows running in a given environment, to find a workflow with a particular name, or to find workflows with a particular ID.

- getAllWorkflows
- getWorkflowsWithName
- getWorkflowForID
Find Workflows by Using the `getAllWorkflows` Operation

The `getAllWorkflows` operation lists all workflows that a user can access as an array of `Workflow` objects.

Because the `getAllWorkflows` operation returns `Workflow` objects that contain all the information about a workflow, it is useful for applications that require full information about workflows, such as the workflow's name, ID, description, parameters, and attributes.

**Prerequisites**

You must have implemented Web service operations in your client application to find objects in the Orchestrator server.

**Procedure**

- Create an array of `Workflow` objects by calling the `getAllWorkflows` operation.

  ```java
  Workflow[] workflows = vsoWebControl.getAllWorkflows(username, password);
  ```

  The preceding code example calls `getAllWorkflows` to get an array of `Workflow` objects that the Web service client can run.

You found workflows in the Orchestrator server that the Web service client can run on objects.

**What to do next**

Implement operations in the Web services client to run the workflows it finds.

Find Workflows by Using the `getWorkflowsWithName` Operation

If you know the name of a particular workflow, as it is defined in the Orchestrator client, the Web service application can obtain this workflow using its name or part of its name.

The `getWorkflowsWithName` operation returns an array of workflows, so you can use it to match several workflows by using wildcards.

**Prerequisites**

You must have implemented Web service operations in your client application to find objects in the Orchestrator server.

**Procedure**

- Create an array of `Workflow` objects by calling the `getWorkflowsWithName` operation.

  ```java
  Workflow[] workflows = vsoWebControl.getWorkflowsWithName("Simple user interaction", username, password);
  ```

  The preceding code example calls the `getWorkflowsWithName` operation to obtain all workflows for which the name, or part of the name, is `Simple user interaction`.

You found workflows in the Orchestrator server that the Web service client can run on objects.

**What to do next**

Implement operations in the Web services client to run the workflows it finds.

Find Workflows by Using the `getWorkflowForID` Operation

If you know a particular workflow ID, a Web service application can obtain this workflow by using the `getWorkflowForID` operation.

The `getWorkflowForID` operation returns a single `Workflow` instance, because all workflow IDs are unique.
Prerequisites
You must have implemented Web service operations in your client application to find objects in the Orchestrator server.

Procedure

1. Create a Workflow object by calling the getWorkflowForID operation.

   String workflowId = "1880808080808080808080808080808087808080011713796199469943be4c882";
   Workflow workflow = vsoWebControl.getWorkflowForID(workflowId, username, password);

You found a workflow in the Orchestrator server that the Web service client can run on objects.

What to do next
Implement operations in the Web services client to run the workflows it finds.

Run Workflows from a Web Service Client
The main purpose of a Web services client is to run workflows across a network.

Prerequisites
You must have implemented Web service operations in the client to find workflows in the Orchestrator server.

Procedure

1. (Optional) Check the workflow user permissions by calling the hasRights operation.

   You can verify if a user has rights to read, run, or edit a particular workflow using the hasRights operation. This operation is not mandatory, but checking user rights before you run a workflow can help prevent exceptions.

   String workflowId = "1880808080808080808080808080808087808080011713796199469943be4c882";
   Boolean rights = vsoWebControl.hasRights(workflowId, username, password, 'x');

   The preceding code example calls the hasRights operation to discover whether the user has the right to run the workflow identified by workflowId.

   If the user has the right to run the workflow, hasRights returns true. Otherwise, hasRights returns false.

2. Set the workflow attributes in a WorkflowTokenAttribute object.

   The Web services client passes WorkflowTokenAttributes arrays to a WorkflowToken object, which runs the workflow.

   WorkflowTokenAttribute[] attributes = new WorkflowTokenAttribute[1];
   WorkflowTokenAttribute attribute = new WorkflowTokenAttribute();
   attribute.setName("vm");
   attribute.setType(finderResult.getType());
   attribute.setValue(finderResult.getDunesUri());
   attributes[0] = attribute;
The preceding example creates a `WorkflowTokenAttribute` object, then populates it with the following information:

- The name of the attribute, in this case, `vm`.
- The type of attribute, as discovered in a `FinderResult` object defined elsewhere in the code.
- The attribute value, which in this case is a `dunesUri` string, signifying that the value specifies an object accessed through a plug-in.

3 Run the workflow by calling the `executeWorkflow` operation.

To run a workflow, you pass the workflow attributes to the `executeWorkflow` operation in the form of a `WorkflowTokenAttribute` array.

Running a workflow creates a `WorkflowToken` object, which represents the instance of the workflow that runs with the specific input parameters that it receives when it starts.

```
WorkflowToken token = vsoWebControl.executeWorkflow(workflowId, username, password, attributes);
```

In the preceding example, the `attributes` property is the array of `WorkflowTokenAttribute` objects created in Step 2.

Sometimes, workflows require input parameters during their run. In these cases, you can provide attributes through a user interaction while the workflow is running. You can pass attributes to the workflow during its run using the `answerWorkflowInput` operation.

You implemented operations in the Web service client that check user permissions, pass attributes to a workflow, and run the workflow.

**What to do next**

Implement operations in the Web services client to interact with workflows while they run.

**Interact with a Workflow While it Runs**

After the workflow starts, the Web services client can perform various actions in response to events while the workflow is running.

**Prerequisites**

You must have implemented operations in the Web service client to run workflows in the Orchestrator server.
Procedure

1. Find running workflows by calling the `getWorkflowTokenForId` operation.

   Calling `getWorkflowTokenForId` obtains a `WorkflowToken` object, which contains all of the information about that specific workflow token.

   ```java
   WorkflowToken onemoretoken = vsoWebControl.getWorkflowTokenForId(workflowTokenId, username, password);
   AllActiveWorkflowTokens[n] = onemoretoken;
   ``

   The preceding code example obtains a `WorkflowToken` object from its ID and sets it into an array of running `WorkflowToken` objects.

2. Check the status of a workflow token by calling the `getWorkflowTokenStatus` operation.

   When a workflow runs, an application's main event loop usually concentrates on checking the status of the workflow at regular intervals. The `getWorkflowTokenStatus` operation requires an array of the IDs of the workflow tokens for which it is obtaining the status.

   ```java
   String workflowId = workflows[0].getId();
   WorkflowToken token = vsoWebControl.executeWorkflow(workflowId, username, password, null);
   String[] tokenIds = { token.getId() };
   String tokenStatus = "";
   while ("completed".equals(tokenStatus) == false && "failed".equals(tokenStatus) == false && "canceled".equals(tokenStatus) == false && "waiting".equals(tokenStatus) == false) { Thread.sleep(1 * 1000);
   // Wait 1s
   String[] status = vsoWebControl.getWorkflowTokenStatus(tokenIds, username, password);
   tokenStatus = status[0];
   System.out.println("Workflow is still running...(" + tokenStatus + ")");
   }
   
   The preceding example obtains the IDs of an array of workflow tokens. It checks the status of multiple `WorkflowToken` objects at one time by running `getWorkflowTokenStatus` in a loop.

   The preceding example keeps the application updated on the status of the `WorkflowToken` objects by checking their state at one second intervals. For example, If the workflow is in the `waiting` state, it is waiting for runtime input from the `answerWorkflowInput` operation.

3. Provide inputs from user interactions by calling the `answerWorkflowInput` operation.

   If a workflow is waiting for user input in the `waiting` state, an application's event loop can specify that input at any time. You can create `WorkflowTokenAttribute` arrays as normal, and then supply them to a workflow during its run by using the `answerWorkflowInput` operation. The following example continues the code from Step 2.

   ```java
   if ("waiting".equals(tokenStatus) == true) {
       System.out.println("Answering user interaction");
       WorkflowTokenAttribute[] attributes = new WorkflowTokenAttribute[2];
       WorkflowTokenAttribute attribute = null;
       attribute = new WorkflowTokenAttribute();
       attribute.setName("param1");
       attribute.setType("string");
       attribute.setValue("answer1");
       attributes[0] = attribute;
       attribute = new WorkflowTokenAttribute();
       attribute.setName("param2");
       attribute.setType("number");
   }
   ```
attribute.setValue("123");
attributes[1] = attribute;
vsowebControl.answerWorkflowInput(token.getId(), attributes, username, password);
}

In the preceding example, if the workflow is in the waiting state, the application creates two WorkFlowTokenAttribute objects. The objects call the various WorkFlowTokenAttribute operations to obtain the attribute values. The process then adds these WorkFlowTokenAttribute objects into a WorkFlowTokenAttribute array.

4 Cancel a workflow by calling the cancelWorkflow operation.

You can cancel a workflow at any time using the cancelWorkflow operation.

vsowebControl.cancelWorkflow(workflowTokenId, username, password);

5 Check that the workflow canceled successfully.

Because the cancelWorkflow operation does not return anything, you must obtain the WorkflowToken status to make sure the workflow canceled successfully, as the following code example shows.

String[] status = vsowebControl.getWorkflowTokenStatus(tokenIds, username, password);
if ("canceled".equals(status) == true) {
    System.out.println("Workflow canceled");
}

The Web service client interacts with workflows by finding their status, supplying input parameters from user interactions, and by canceling the workflows.

What to do next

Implement operations in the Web services client to extract the workflow results.

Obtain Workflow Results

After the workflow completes its run, you can retrieve the results by calling the getWorkflowTokenResult( ) operation.

Prerequisites

You must have implemented how workflows start in the Orchestrator server in the Web services client.

Procedure

1 Obtain the results of a running workflow by calling the getWorkflowTokenResult( ) operation.

The getWorkflowTokenResult( ) operation stores the results as an array of attributes.

WorkflowTokenAttribute[] retAttributes =
    vsowebControl.getWorkflowTokenResult(token.getId(), username, password);
The preceding example code obtains the result of a workflow token with a specific identifier.

2 (Optional) Check the workflow results by calling the WorkflowTokenAttribute.getValue() operation.

```java
WorkflowTokenAttribute resultCode = result[0];
WorkflowTokenAttribute resultMessage = result[1];
System.out.println("Workflow output code... (" + resultCode.getValue() + ")");
System.out.println("Workflow output message... (" + resultMessage.getValue() + ")");
```

3 Emit the workflow token’s result attributes for display or for use by other applications.

```java
for (int ii = 0; ii < retAttributes.length; ii++) {
    System.out.println("Name: '\" + retAttributes[ii].getName()
        + '\" - Type: '\" + retAttributes[ii].getType()
        + '\" - Value: '\" + retAttributes[ii].getValue()
}
```

The preceding example code prints out the name, type, and value of the workflow token’s result attributes.

You defined a Web services client that finds objects in Orchestrator, runs workflows on them, interacts with the running workflows, and extracts the results of running those workflows.

**Time Zones and Running Workflows through Web Services**

Running workflows through Web services can lead to erroneous timestamping, if the run request comes from an application running in a different time zone to the Orchestrator server.

If a workflow takes the time and date as an input parameter, and generates the time and date as output when it runs, and if this workflow runs through a Web services application, the time and date sent as an input parameter reflects the time and date of the system on which the Web services application is running. The time and date that the workflow sends as its output reflects the time and date of the system on which the Orchestrator server is running. If the Web services application is running in a different time zone than the Orchestrator server, the time returned by the workflow does not match the time that the Web services application provided as input when it called executeWorkflow or getWorkflowTokenResult.

To avoid this problem, you can create a function to compare dates in your Web services application. You must serialize the date and time, taking the time zone information into account. The following Java code example shows how to transform a String that Orchestrator returns into a Date object.

```java
public Date dateFromString(String value){
    java.text.DateFormat s_dateFormat = new java.text.SimpleDateFormat("yyyyMMddHHmmssZ");
    Date date = null;
    if (value != null && value.length() > 0) {
        try {
            date = s_dateFormat.parse(value);
        } catch (ParseException e) {
            System.err.println("Converting String to Date : ERROR");
            date = null;
        }
    }
    return date;
}
```
Web Service Application Examples

Orchestrator provides working examples of Web services client applications that provide Web access to Orchestrator.

You can download the Orchestrator examples ZIP file from the VMware vCenter Orchestrator 4.0 Documentation download page. For information about where to find the documentation download page, see “Example Applications.”

Web Service API Objects

The Orchestrator Web service API provides a collection of objects that serve as WSDL complex types and a collection of methods that server as WSDL operations.

FinderResult Object

A FinderResult represents an object from the Orchestrator inventory that Orchestrator locates either directly or through a plug-in. For example, a FinderResult object can represent a virtual machine from vCenter Server.

FinderResult objects represent any object that a plug-in registers with Orchestrator in its vso.xml file. FinderResult objects represent the items, from all installed plug-ins, that you find when you call one of the find* operations. The items returned can be virtual machine objects, ESX objects, infrastructure criteria, or Orchestrator workflow tokens. Most workflows require FinderResult instances as input parameters, as most workflows act upon Orchestrator objects.

You cannot set a FinderResult as a workflow attribute directly. You must set WorkflowTokenAttribute in workflows instead, which take the type and the dunesUri from FinderResult objects.

The find operation finds objects according to query criteria that the vso.xml file defines. It does not return FinderResult objects directly, but returns QueryResult objects instead. QueryResult objects contain arrays of FinderResult objects.

The objects searched for can also be identified by ID or by relation using the findForId and findRelation operations, as the following example shows.

```java
public FinderResult findForId(String type, String id, String username, String password);
public FinderResult[] findRelation(String parentType, String parentId, String relation, String username, String password);
```

**NOTE** FinderResult is not an Orchestrator scriptable object.

The following table shows the properties of the FinderResult object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>type</td>
<td>Type of object found.</td>
</tr>
<tr>
<td>String</td>
<td>id</td>
<td>ID of the discovered object.</td>
</tr>
<tr>
<td>Array of</td>
<td>properties</td>
<td>A list of the discovered object’s properties.</td>
</tr>
<tr>
<td>properties</td>
<td></td>
<td>The format of the properties values is defined by each plug-in in its vso.xml file, under the FinderResult description.</td>
</tr>
<tr>
<td>String</td>
<td>dunesUri</td>
<td>A string representation of the object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If a FinderResult object is accessed through a plug-in, it is identified by a dunesUri string, rather than by another type of string or ID. The format of the dunesUri is as follows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dunes://service.dunes.ch/CustomSDKObject?id='&lt;object_ID&gt;' &amp;dunesName='&lt;plug-in_name&gt;:&lt;object_type&gt;'</td>
</tr>
</tbody>
</table>
ModuleInfo Object

ModuleInfo stores the name, version, description, and name attributes for each plug-in. A Web service application can use these attributes to modify its behavior based on the presence or absence of certain modules or module versions.

The getAllPlugins operation returns arrays of ModuleInfo objects to list all the plug-ins a user can access, as the following example shows.

public ModuleInfo[] getAllPlugins(String username, String password);

The following table shows the properties of the ModuleInfo object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>moduleName</td>
<td>The name of the plug-in.</td>
</tr>
<tr>
<td>String</td>
<td>moduleVersion</td>
<td>The plug-in version.</td>
</tr>
<tr>
<td>String</td>
<td>moduleDescription</td>
<td>Description of the plug-in.</td>
</tr>
<tr>
<td>String</td>
<td>moduleDisplayName</td>
<td>The plug-in name shown in Orchestrator.</td>
</tr>
</tbody>
</table>

Property Object

A Property object represents a key-value pair that describes the properties of an item in the Orchestrator inventory.

You can obtain a Property object by calling the getProperties operation on a FinderResult object, as the following example shows.

Property[] props = finderResult.getProperties();

This example method call returns the contents of the FinderResult object’s properties attribute.

The following table shows the properties of the Property object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>name</td>
<td>Property name.</td>
</tr>
<tr>
<td>String</td>
<td>value</td>
<td>Property value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The format of a property’s values is defined by each plug-in in its vso.xml file, under the FinderResult description.</td>
</tr>
</tbody>
</table>

QueryResult Object

The QueryResult object represents the results of a find query.

A QueryResult object contains an array of FinderResult objects and a counter. A QueryResult object is returned by the find operation, as the following example shows.

public QueryResult find(String type, String query, String username, String password);

The following table shows the properties of the QueryResult object.
### Workflow Object

A **Workflow** object represents an Orchestrator workflow that defines a certain sequence of tasks, decisions, and operations.

Users with the correct permissions can obtain specific Workflow objects by name or by ID, or they can obtain all the workflows in the server they have the permission to see. Orchestrator provides the following operations to obtain Workflow objects.

```java
public Workflow[] getWorkflowsWithName(String name, String username, String password);
public Workflow getWorkflowForId(String workflowId, String username, String password);
public Workflow[] getAllWorkflows(String username, String password);
```

The following table shows the properties of the Workflow object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>id</td>
<td>The workflow ID. The id String is a globally unique ID string. Workflows that Orchestrator creates have identifiers that are very large strings, with a very low probability of namespace collision.</td>
</tr>
<tr>
<td>String</td>
<td>name</td>
<td>The name of the workflow, as it appears in the workflow’s Name text box in Orchestrator.</td>
</tr>
<tr>
<td>String</td>
<td>description</td>
<td>A detailed description of what the workflow does.</td>
</tr>
<tr>
<td>WorkflowParameter[]</td>
<td>inParameters</td>
<td>The inParameters array is the set of WorkflowParameter objects that are the workflow’s input parameters. The workflow can manipulate these input parameters or use them directly as the input parameters for tasks and other workflows. You can set up arbitrary input parameters to provide any necessary input parameters. Omitting a required parameter at runtime causes the workflow to fail.</td>
</tr>
<tr>
<td>WorkflowParameter[]</td>
<td>outParameters</td>
<td>The outParameters array is the set of WorkflowParameter objects that result from running a workflow. This array allows the workflow to send errors, the names of any created objects, and other information as output. You can set up arbitrary output parameters to generate any information that you need.</td>
</tr>
<tr>
<td>WorkflowParameter[]</td>
<td>attributes</td>
<td>The attributes array is a set of WorkflowParameter objects that represent constants and pre-set variables for a given workflow. Attributes differ from inParameters because they are intended to represent environmental constants or variables, rather than runtime information. <strong>Note</strong>: You cannot retrieve workflow attribute values by using the Web service. You can only retrieve output parameter values.</td>
</tr>
</tbody>
</table>
WorkflowParameter Object

The WorkflowParameter object defines a parameter in a workflow, for example, an input, an output, or an attribute.

Workflow developers can set up arbitrary parameters to provide any input parameters or output parameters that the workflows need. The format of the parameters is defined entirely by the workflow.

The following table shows the properties of the WorkflowParameter object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>name</td>
<td>The parameter name.</td>
</tr>
<tr>
<td>String</td>
<td>type</td>
<td>The parameter type.</td>
</tr>
</tbody>
</table>

WorkflowToken Object

A WorkflowToken object represents a specific instance of a workflow in the running, waiting, waiting-signal, canceled, completed or failed state.

You obtain a WorkflowToken object by starting a workflow or by obtaining an existing workflow token by its ID, as the following method signatures show.

```java
public WorkflowToken executeWorkflow(String workflowId, String username, String password, WorkflowTokenAttribute[] attributes);
public WorkflowToken getWorkflowTokenForId(String workflowTokenId, String username, String password);
```

The following table shows the properties of the WorkflowToken object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>id</td>
<td>The identifier of this particular instance of a completed workflow.</td>
</tr>
<tr>
<td>String</td>
<td>title</td>
<td>The title of this particular instance of a completed workflow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>By default, the WorkflowToken title is the same as the Workflow title, although some operations do allow you to set a different WorkflowToken title when you start the workflow.</td>
</tr>
<tr>
<td>String</td>
<td>workflowId</td>
<td>The identifier of the workflow of which this WorkflowToken object is a running instance.</td>
</tr>
<tr>
<td>String</td>
<td>currentItemName</td>
<td>The name of the step in the workflow that is running at the moment when getWorkflowTokenForId is called.</td>
</tr>
<tr>
<td>String</td>
<td>currentItemState</td>
<td>The state of the current step in the workflow, with the following possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>running: the step is running</td>
</tr>
<tr>
<td></td>
<td></td>
<td>waiting: the step is waiting for runtime parameters, which can be provided by answerWorkflowInput</td>
</tr>
<tr>
<td></td>
<td></td>
<td>waiting-signal: the step is waiting for an external event from a plug-in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>canceled: the step was canceled by a user or API-integrated program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>completed: the step has finished</td>
</tr>
<tr>
<td></td>
<td></td>
<td>failed: the step encountered an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You must run getWorkflowTokenForId every time you update this value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note</strong>: VMware recommends that you do not use currentItemState. The globalState property makes currentItemState redundant.</td>
</tr>
</tbody>
</table>

VMware, Inc.

185
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>globalState</td>
<td>The state of the workflow as a whole, with the following possible values:</td>
</tr>
<tr>
<td></td>
<td>running</td>
<td>the workflow is running</td>
</tr>
<tr>
<td></td>
<td>waiting</td>
<td>the workflow is waiting for runtime parameters, which can be provided by answerWorkflowInput</td>
</tr>
<tr>
<td></td>
<td>waiting-signal</td>
<td>the workflow is waiting for an external event</td>
</tr>
<tr>
<td></td>
<td>canceled</td>
<td>the workflow was canceled by a user or by an application</td>
</tr>
<tr>
<td></td>
<td>completed</td>
<td>the workflow has finished</td>
</tr>
<tr>
<td></td>
<td>failed</td>
<td>the workflow encountered an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The globalState is the state of the workflow as a whole. You must run getWorkflowTokenForId every time you update this value.</td>
</tr>
<tr>
<td></td>
<td>startDate</td>
<td>The date and time that this workflow token started</td>
</tr>
<tr>
<td></td>
<td>endDate</td>
<td>Date and time that this workflow token ended, if the workflow token has finished. The endDate value is filled in at the moment the workflow</td>
</tr>
<tr>
<td></td>
<td>xmlContent</td>
<td>Defines input parameters, output parameters, attributes, and the content of error messages. The values of the attributes and parameters are set in CDATA elements and error messages are set in &lt;exception&gt; tags, as the following example shows.</td>
</tr>
</tbody>
</table>

```xml
<token>
  <atts>
    <stack>
      <att n='attstr' t='string' e='n'>
        <![CDATA[attribute]]>
        Attribute value</att>
      <att n='instr' t='string' e='n'>
        <![CDATA[]]>
        Input parameter value</att>
      <att n='outstr' t='string' e='n'>
        <![CDATA[]]>
        Output parameter value</att>
    </stack>
  </atts>
  <exception encoded='n'>Error message</exception>
</token>
```

**WorkflowTokenAttribute Object**

A WorkflowTokenAttribute object represents an input or output parameter of a running instance of a workflow.

A WorkflowTokenAttribute is a value that you pass to a predefined WorkflowParameter when a WorkflowToken begins, or in some cases, at runtime. When you run a workflow, you supply the input parameters for that particular workflow as WorkflowTokenAttribute objects. The executeWorkflow operation takes an array of WorkflowTokenAttribute objects as an argument when you call it, as the following example shows.

```java
public WorkflowToken executeWorkflow(String workflowId, String username, String password, WorkflowTokenAttribute[] attributes);
```

Workflows also use WorkflowTokenAttribute as the output parameter of a run workflow. WorkflowTokenAttribute contains the results of a completed WorkflowToken created by running executeWorkflow. You can collect the result of a WorkflowToken, in the form of a WorkflowTokenAttribute, by calling getWorkflowTokenResult, as the following example shows.

```java
public WorkflowTokenAttribute[] getWorkflowTokenResult(String workflowTokenId, String username, String password);
```
You can also pass an array of WorkflowTokenAttribute objects to the answerWorkflowInput operation to provide input that a workflow token needs while it runs.

```java
public void answerWorkflowInput(String workflowTokenId, WorkflowTokenAttribute[] answerInputs, String username, String password);
```

The following table shows the properties of the WorkflowTokenAttribute object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>name</td>
<td>Name of the input or output parameter</td>
</tr>
<tr>
<td>String</td>
<td>type</td>
<td>Type of input or output parameter</td>
</tr>
</tbody>
</table>
| String | value      | The value property represents either the input or output parameter value for this particular workflow token, in the form of a string. If the type is an array of objects, the value is a string of the following format: "#{<type1>#<value1>#{<type2>#{<value2>#{...}}#}" If the value property specifies an object obtained from a plug-in, then the input or output parameter value is a dunesUri string that points to the object in question. The following example shows the format of the dunesUri. dunes://service.dunes.ch/CustomSDKObject?id='object_ID'&dunesName='<plug-in_name>:<object_type>

Web Service API Operations

The Orchestrator Web service API provides a collection of methods that server as WSDL operations.

**Note** Every Web service operation, except `echo`, `echoWorkflow`, and `sendCustomEvent` uses the Orchestrator user name and password to authenticate the session. The operations throw exceptions if you use the incorrect username or password.

**answerWorkflowInput Operation**

The `answerWorkflowInput` operation passes information from a user or an external application to a workflow while the workflow is running.

If a running workflow reaches a stage that requires an input from a user action or external application, the WorkflowToken enters the waiting state until it receives the input from `answerWorkflowInput`. The `answerWorkflowInput` operation provides input in the form of an array of WorkflowTokenAttribute objects.

The `answerWorkflowInput` operation is declared as the following example shows.

```java
public void answerWorkflowInput(String workflowTokenId, WorkflowTokenAttribute[] answerInputs, String username, String password);
```

The Web service performs only a simple validation of the input attributes you provide for running a workflow. The Web service verifies only that the attributes that you set in the WorkflowTokenAttribute objects are of the expected type. The Web service does not perform complex validation to verify that you set all of the WorkflowTokenAttribute objects' properties correctly. The Web service does not access the parameter properties that the workflow developer set in the workflow Presentation. If one of the WorkflowTokenAttribute objects' properties is not set, or if an attribute value is not one that the workflow expects, the Web service sends the `answerWorkflowInput` request, with the invalid WorkflowTokenAttribute object. If a WorkflowTokenAttribute object is invalid, the workflow fails, entering the failed state without informing the Web service application. Your Web service application can check whether a workflow runs correctly or fails by calling the `getWorkflowTokenStatus` operation during and after the workflow runs.
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowTokenId</td>
<td>The ID of a running workflow that is waiting for input from a user interaction or external application</td>
</tr>
<tr>
<td>Array of WorkflowTokenAttribute objects</td>
<td>answerInputs</td>
<td>The result of the user interaction or external application, passed as input to the waiting workflow</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password</td>
</tr>
</tbody>
</table>

**Return Value**

No return value. Throws an exception if you pass it an invalid parameter.

**cancelWorkflow Operation**

The `cancelWorkflow` operation cancels a workflow.

The behavior of the `cancelWorkflow` operation depends on the workflow that it cancels. A canceled workflow stops running in the Orchestrator server and enters the `canceled` state, but the actions that it has already run or started running do not stop or reverse themselves. For example, if a workflow is performing a Power On Virtual Machine operation when you cancel it, the virtual machine does not stop powering on, nor does it power itself off if it has already started.

The `cancelWorkflow` operation is declared as follows.

```java
class OrchestratorService{
    public void cancelWorkflow(String workflowTokenId, String username, String password);
}
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowTokenId</td>
<td>The identifier of the running workflow to cancel</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password</td>
</tr>
</tbody>
</table>

**Return Value**

No return value. The `cancelWorkflow` operation returns an exception if you pass it an invalid parameter.

**echoWorkflow Operation**

The `echoWorkflow` operation tests the connection to the Web service by checking serialization.

The `echoWorkflow` operation provides a useful debugging tool if you are connecting to an older Web service implementation. Calling this operation verifies the connection to the server by checking that the serialize and deserialize operations work correctly.

The `echoWorkflow` operation is declared as follows.

```java
class OrchestratorService{
    public Workflow echoWorkflow(Workflow workflow);
}
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflow</td>
<td>workflow</td>
<td>The <code>echoWorkflow</code> operation takes a <code>Workflow</code> object as a parameter. If the connection and serialization are working correctly, it returns the same workflow.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns the same `Workflow` object as the object provided as an input parameter.
**executeWorkflow Operation**

The `executeWorkflow` operation runs a specified workflow.

The `executeWorkflow` takes an array of `WorkflowTokenAttribute` objects as input parameters, which provide the specific attributes with which this particular workflow instance runs.

The `executeWorkflow` operation is declared as follows.

```java
public WorkflowToken executeWorkflow(String workflowId, String username, String password, WorkflowTokenAttribute[] attributes);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowId</td>
<td>The identifier of the workflow to run</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password</td>
</tr>
<tr>
<td>Array of WorkflowTokenAttribute instances</td>
<td>workflowInputs</td>
<td>Array of input parameters required to run the workflow</td>
</tr>
</tbody>
</table>

**Return Value**

Returns a `WorkflowToken` object. Returns an exception if you pass it an invalid parameter.

**find Operation**

The `find` operation finds elements that correspond to a particular query.

The `find` operation obtains objects of any type by searching for a particular name. The query results are provided in the form of a `QueryResult` object, which contains an array of `FinderResult` objects with a total counter. The query itself is passed to `find` as the second parameter, as the following operation declaration shows.

```java
public QueryResult find(String type, String query, String username, String password);
```

Query parsing is performed by the plug-in that contains the objects you are looking for. The query language used by the `find` operation is defined by the plug-in. Consequently, the syntax of the `query` parameter differs according to the implementation of the plug-in. Most of the officially supported Orchestrator plug-ins do not store any objects in the inventory, so they do not expose anything that can be searched for. Table 7-1 provides the syntax and behavior for the query parameter for the officially supported Orchestrator plug-ins.

This table describes the `find` operation `query` parameter syntax for each of the supported Orchestrator plug-ins.

**Table 7-1. Query Syntax of the Orchestrator Plug-Ins**

<table>
<thead>
<tr>
<th>Orchestrator Plug-in</th>
<th>Query Parameter Syntax</th>
<th>Query Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database (for example Lifecycle Manager)</td>
<td>String</td>
<td>Searches for object names in SQL database tables, that Orchestrator sets the search String in an SQL WHERE keyword search. It searches the primary keys, then the object IDs in the database.</td>
</tr>
<tr>
<td>Enumeration</td>
<td>Not applicable</td>
<td>The enumeration plug-in stores nothing in the inventory. You can find enumerations on each data type that contains enumeration types.</td>
</tr>
<tr>
<td>Jakarta common set</td>
<td>Not applicable</td>
<td>The Jakarta plug-in stores nothing in the inventory.</td>
</tr>
</tbody>
</table>
### Table 7-1. Query Syntax of the Orchestrator Plug-Ins (Continued)

<table>
<thead>
<tr>
<th>Orchestrator Plug-in</th>
<th>Query Parameter Syntax</th>
<th>Query Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>Not applicable</td>
<td>The JDBC plug-in stores nothing in the inventory.</td>
</tr>
<tr>
<td>Library</td>
<td>Not applicable</td>
<td>The library plug-in stores nothing in the inventory.</td>
</tr>
<tr>
<td>Mail</td>
<td>Not applicable</td>
<td>The mail plug-in stores nothing in the inventory.</td>
</tr>
<tr>
<td>SSH</td>
<td>If you have configured Orchestrator to use SSH connections, you can make queries SSH commands.</td>
<td>The SSH plug-in stores nothing in the inventory.</td>
</tr>
<tr>
<td>VMware Infrastructure 3.5</td>
<td>String or null</td>
<td>Ignores the query string and returns all objects of the specified type.</td>
</tr>
<tr>
<td>vCenter Server 4.0</td>
<td>String or null</td>
<td>Ignores the query string and returns all objects of the specified type.</td>
</tr>
<tr>
<td>XML</td>
<td>Not applicable</td>
<td>The XML plug-in stores nothing in the inventory.</td>
</tr>
</tbody>
</table>

When you develop plug-ins, you can define a query language to use `find` to search for named objects through the custom plug-in. This definition is not mandatory. The syntax of the query parameter is entirely dependent on the query language that the plug-in implements. To avoid defining a query language, make `find` return all objects, as in the case of the VMware Infrastructure plug-ins.

The size of the array of objects that the `QueryResult` returns depends on the definition of the plug-in through which you make the query. For the queries you make through the standard Orchestrator plug-ins, the array contains an unlimited number of `FinderResult` objects. Developers of third-party plug-ins, however, can set a limit on the number of results the query returns. In these cases, if the value of `totalCount` exceeds the number of objects in the array of `FinderResult` objects, the array does not include all the objects found in the queried inventory. The `totalCount` property does report the total number of `FinderResult` objects found. The `totalCount` property can be negative, which signifies that the plug-in cannot determine how many corresponding objects are in the plug-in.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>type</td>
<td>Type of object looked for.</td>
</tr>
<tr>
<td>String</td>
<td>query</td>
<td>The query. The query is a string enclosed in quotation marks. Any object of the type specified by the <code>type</code> parameter with a name that matches the query string is returned in the <code>QueryResult</code>.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns the result of the query as a `QueryResult` object.

If `find` fails to match an object, `QueryResult.getTotalCount` returns 0 and `QueryResult.getElement` returns null.

If the server does not recognize the object type or plug-in searched for, `find` throws an exception. `find` also returns an exception if you pass it an invalid parameter.
**findForId Operation**

The findForId operation searches for a specific FinderResult object according to that FinderResult object's type and id properties.

You can use the findForId operation to acquire information about FinderResult objects you have already found by using the other find* operations. For example, you can use the findForId method to obtain the state of a FinderResult object you found by using the find operation.

The findForId operation is declared as the following example shows.

```java
public FinderResult findForId(String type, String id, String username, String password);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>type</td>
<td>Type of object looked for.</td>
</tr>
<tr>
<td>String</td>
<td>id</td>
<td>ID of the object looked for.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns a FinderResult object containing details of the object found. Returns null if you pass it an invalid parameter.

**findRelation Operation**

The findRelation operation finds all the children elements in an inventory that belong to a particular parent or type of parent.

Knowing how a child is related to its parent is useful if you develop tree viewers to view the objects in a library. The findRelation operation is declared as follows.

```java
public FinderResult[] findRelation(String parentType, String parentId, String relation, String username, String password);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>parentType</td>
<td>The type of parent object. The parentType property can be the name of a plug-in, or it can specify a more narrowly defined parent. For example, you can specify the parentType as &quot;VC:&quot; to obtain the objects at the root of VMware vCenter Server 4.0 plug-in, or you can a specific folder, such as &quot;VC:VmFolder&quot;.</td>
</tr>
<tr>
<td>String</td>
<td>parentId</td>
<td>The ID of a particular parent object. The parentId parameter allows you to find the children of a specific parent object, if you know its ID.</td>
</tr>
<tr>
<td>String</td>
<td>relation</td>
<td>The name of the relation. Calling findRelation returns all children elements under a parent identified by its parentId. If you omit the parentId the parentType is not the root type of the inventory, the findRelation operation returns null. See &quot;Relation Types&quot; for more information.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Relation Types**

The relation property types are defined by the plug-ins. The validity of relations depends on the parent type. This table lists the relation types defined by each of the standard plug-ins provided by Orchestrator.
<table>
<thead>
<tr>
<th>Plug-In</th>
<th>Relation Names</th>
<th>Relation Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumerations</td>
<td>No relations</td>
<td>No relations</td>
</tr>
<tr>
<td>Jakarta Commons Net</td>
<td>No relations</td>
<td>No relations</td>
</tr>
<tr>
<td>JDBC</td>
<td>No relations</td>
<td>No relations</td>
</tr>
<tr>
<td>Library</td>
<td>No relations</td>
<td>No relations</td>
</tr>
<tr>
<td>Mail</td>
<td>No relations</td>
<td>No relations</td>
</tr>
<tr>
<td>Networking</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IpAddress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPV4Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MacAddressPool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NetworkDomain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proxy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subnet</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range</td>
</tr>
<tr>
<td>SSH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>File</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Folder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RootFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SshConnection</td>
</tr>
<tr>
<td>vCenter Server</td>
<td>getComputeResource_ClusterComputeResource()</td>
<td>ComputeResource</td>
</tr>
<tr>
<td></td>
<td>getComputeResource_ComputeResource()</td>
<td>ComputeResource</td>
</tr>
<tr>
<td></td>
<td>getDatacenter()</td>
<td>Datacenter</td>
</tr>
<tr>
<td></td>
<td>getDatadstore()</td>
<td>Datadstore</td>
</tr>
<tr>
<td></td>
<td>getDatadstoreFolder()</td>
<td>DatadstoreFolder</td>
</tr>
<tr>
<td></td>
<td>getFolder()</td>
<td>DatacenterFolder</td>
</tr>
<tr>
<td></td>
<td>getFolder()</td>
<td>DatadstoreFolder</td>
</tr>
<tr>
<td></td>
<td>getFolder()</td>
<td>HostFolder</td>
</tr>
<tr>
<td></td>
<td>getFolder()</td>
<td>NetworkFolder</td>
</tr>
<tr>
<td></td>
<td>getFolder()</td>
<td>VmFolder</td>
</tr>
<tr>
<td></td>
<td>getHost()</td>
<td>HostSystem</td>
</tr>
<tr>
<td></td>
<td>getHostFolder()</td>
<td>HostFolder</td>
</tr>
<tr>
<td></td>
<td>getNetwork()</td>
<td>Network</td>
</tr>
<tr>
<td></td>
<td>getNetworkFolder()</td>
<td>NetworkFolder</td>
</tr>
<tr>
<td></td>
<td>getNetwork_DistributedVirtualPortgroup()</td>
<td>DistributedVirtualPortgroup</td>
</tr>
<tr>
<td></td>
<td>getNetwork_Network()</td>
<td>Network</td>
</tr>
<tr>
<td></td>
<td>getOwner()</td>
<td>ComputeResource</td>
</tr>
<tr>
<td></td>
<td>getParentFolder()</td>
<td>VmFolder</td>
</tr>
<tr>
<td></td>
<td>getPortgroup()</td>
<td>DistributedVirtualPortgroup</td>
</tr>
<tr>
<td></td>
<td>getRecentTask()</td>
<td>Task</td>
</tr>
<tr>
<td></td>
<td>getResourcePool()</td>
<td>ResourcePool</td>
</tr>
<tr>
<td></td>
<td>getResourcePool_ResourcePool()</td>
<td>ResourcePool</td>
</tr>
<tr>
<td></td>
<td>getResourcePool_VirtualApp()</td>
<td>VirtualApp</td>
</tr>
<tr>
<td></td>
<td>getRootFolder()</td>
<td>DatacenterFolder</td>
</tr>
<tr>
<td></td>
<td>getSdkConnections()</td>
<td>SdkConnection</td>
</tr>
<tr>
<td></td>
<td>getVm()</td>
<td>VirtualMachine</td>
</tr>
<tr>
<td></td>
<td>getVmFolder()</td>
<td>VmFolder</td>
</tr>
<tr>
<td></td>
<td>getVmSnapshot()</td>
<td>VirtualMachineSnapshot</td>
</tr>
<tr>
<td>XML</td>
<td>No relations</td>
<td>No relations</td>
</tr>
</tbody>
</table>
The relation property can also reference relation types specified in each plug-in's `vso.xml` file. The following example is an excerpt from the networking plug-in `vso.xml` file.

```xml
<relations>
  <relation name="Subnet" type="Class:Subnet"/>
  <relation name="Range" type="Class:Range"/>
  <relation name="NetworkDomain" type="Class:NetworkDomain"/>
  <relation name="MacAddressPool" type="Class:MacAddressPool"/>
</relations>
```

In addition to the relation types listed in Table 7-2, Orchestrator also defines the CHILDREN relation, to represent all relation types.

**Return Value**

Returns a list of FinderResult objects.

Returns an exception if no children are found or if you pass it an invalid parameter.

### `getAllPlugin` Operation

The `getAllPlugin` operation returns the description of all the plug-ins installed in Orchestrator.

**IMPORTANT** The `getAllPlugin` operation is deprecated. Use `getAllPlugins` instead.

### `getAllPlugins` Operation

The `getAllPlugins` operation returns the description of all the plug-ins installed in Orchestrator.

Many of the actions that you perform using Orchestrator depend on functions that you enable through plug-ins. Workflows might depend on the existence of certain custom plug-ins, or on standard plug-ins that the administrator has disabled. Consequently, you can check that the necessary plug-ins are present before you run a workflow. Without the necessary plug-ins, some object types used by workflows might be absent.

The `getAllPlugins` operation lists all the available plug-ins as an array of `ModuleInfo` objects. The `ModuleInfo` objects store the name, version, description, and name for each plug-in. A Web service application can use these attributes to modify its behavior based on the presence or absence of certain plugged-in modules or versions.

The `getAllPlugins` operation is declared as follows.

```java
public ModuleInfo[] getAllPlugins(username, password);
```

**NOTE** The `getAllPlugins` operation replaces the deprecated operation `getAllPlugin`.

The following table describes the `getAllPlugins` operation properties.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns a list of plug-in descriptions as `ModuleInfo` objects.
getAllWorkflows Operation

The `getAllWorkflows` operation finds all available workflows.

The `getAllWorkflows` operation lists all the workflows available in an Orchestrator server as an array of `Workflow` objects. The `getAllWorkflows` operation is also useful for programs that must list information about workflows, such as the workflows' names, IDs, and so on. The `Workflow` objects present all the relevant information about the workflows.

The `getAllWorkflows` operation is declared as follows.

```java
public Workflow[] getAllWorkflows(String username, String password);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns an array of `Workflow` objects.

getWorkflowForId Operation

The `getWorkflowForId` operation retrieves a workflow identified by its unique ID.

If you know the ID of a specific workflow, you can use the `getWorkflowForId` operation to obtain the workflow object. Multiple workflows running through different plug-ins might have the same name. The safest way to obtain workflows is to use the `getWorkflowsWithName` operation to obtain their ID, rather than by obtaining them by name.

You can find out a workflow ID by checking the workflow's `workflowID` property, as the following example shows.

```java
String workflowId = workflows[0].getId();
```

The `getWorkflowForId` operation is declared as follows.

```java
public Workflow getWorkflowForId(String workflowId, String username, String password);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowId</td>
<td>ID of the workflow to retrieve.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns the `Workflow` object that corresponds to the provided ID. Returns null if you pass it an invalid parameter.

getWorkflowsWithName Operation

The `getWorkflowsWithName` operation searches for workflows by their name.

The `getWorkflowsWithName` operation is declared as follows.

```java
public Workflow[] getWorkflowsWithName(String workflowName, String username, String password);
```
If you know the name (or a part of the name) of a particular workflow, you can obtain this workflow by calling `getWorkflowsWithName`. The `getWorkflowsWithName` operation returns an array of workflows, so it can be used to find several workflows at one time.

Orchestrator 4.0 provides two VMware vCenter Server plug-ins, one for vCenter Server 3.x and one for vCenter Server 4.0. If you obtain workflows by name only, it is not always easy to determine whether those workflows are running through the vCenter Server 3.x or vCenter Server 4.0 plug-in. Workflows with the same name might exist for each plug-in, unless the workflow developer appended a plug-in identifier to the workflow names, such as VI3, VI3.5 or VC. Even if workflow developers add the recommended name suffixes, the safest way to obtain workflows is by their ID, using the `getWorkflowForId` operation. However, if you do use the `getWorkflowsWithName` operation, you can check the workflow's input and output parameters to establish which in plug-in the workflow runs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowName</td>
<td>Name of the workflow to find. The value of the workflowName property can be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a full name or a wildcard (*), which returns all the workflows available to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user. You can also search for partial names. For example, if you enter <code>*Clone</code> or</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Clone*</code> as the workflowName, this returns all workflows with names that contain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the word Clone.</td>
</tr>
</tbody>
</table>

| String     | username     | Orchestrator user name.                                                      |
| String     | password     | Orchestrator password.                                                       |

**Return Value**

Returns an array of `Workflow` objects that correspond to the provided name or name fragment. Workflows are returned in an array even if only one workflow is found. Returns null if you pass it an invalid parameter.

**getWorkflowTokenForId Operation**

The `getWorkflowTokenForId` operation finds the `WorkflowToken` object for a specific workflow token.

The `getWorkflowTokenForId` operation is declared as follows.

```java
public WorkflowToken getWorkflowTokenForId(String workflowTokenId, String username,
                                          String password);
```

Individual threads or functions can run multiple workflows. The `getWorkflowTokenForId` operation allows a central process or thread to track the progress of each workflow. Using `getWorkflowTokenForId` provides access to all the information about a specific `WorkflowToken` because, although checking the token status only requires the ID, it is often useful to obtain all the information about a given token.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowTokenId</td>
<td>ID of this run of the workflow</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns a `WorkflowToken` object for a specific workflow token that corresponds to the provided workflow token ID.
getWorkflowTokenResult Operation

The getWorkflowTokenResult operation obtains the result of running a given workflow.

You can view the results produced by a WorkflowToken object by calling getWorkflowTokenResult. The results of running a workflow are delivered as an array of WorkflowTokenAttribute objects that contain the output parameters that the workflow set during its run. The structure of the output WorkflowTokenAttribute objects is the same as the structure of the input parameters passed to the workflow when it starts. The parameters have a name, type, and value.

You can obtain the results before the workflow finishes. If the workflow has set its output parameters, you can obtain their values by calling getWorkflowTokenResult while the workflow runs. This method allows the workflow to communicate its results to external systems while it is still in the running state. You can also use getWorkflowTokenResult to obtain results from workflows in the failed, waiting, and canceled states, to show the results of the workflow up to the point it entered a nonrunning or incomplete state.

Objects of the Any type do not deserialize correctly. Consequently, you cannot call getWorkflowTokenResult on a workflow token if one of the token's attributes is of the Any type. If you specify the correct object type, for example, VC:VirtualMachine, then getWorkflowTokenResult returns the correct dunesURI value.

If the object that getWorkflowTokenResult obtains is a plain Java object, you can perform deserialization on it by using the standard Java API, but to do so you must include the relevant Java class in your classpath. For example, if the object you obtain is of the type VirtualMachineRuntimeInfo, you must include VirtualMachineRuntimeInfo.class or vmware-vmosdk-vc40.jar in the classpath. You find the vmware-vmosdk-vc40.jar file in install-directory\VMware\Orchestrator\app-server\server\vmo\tmp\dars\vmware-vmosdk-vc40.dar\lib.

The getWorkflowTokenResult operation is declared as follows.

```java
public WorkflowTokenAttribute[] getWorkflowTokenResult(String workflowTokenId, String username, String password);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowTokenId</td>
<td>ID of this specific run of the workflow</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

Return Value

Returns an array of WorkflowTokenAttribute objects that correspond to the provided workflow token ID or IDs. Returns null if you pass it an invalid parameter.

getWorkflowTokenStatus Operation

The getWorkflowTokenStatus operation obtains the globalStatus of specific workflow tokens.

The getWorkflowTokenStatus operation checks the status of a workflow or an array of workflows while they run. The getWorkflowTokenStatus operation obtains the globalStatus value from running WorkflowToken objects, identified by their workflowTokenId. The globalStatus value can be one of the following.

- running: the workflow is running
- waiting: the workflow is waiting for runtime parameters, which can be provided by answerWorkflowInput
- waiting-signal: the workflow is waiting for an external event
- canceled: the workflow was canceled by a user or by an application
- completed: the workflow has finished
- failed: the workflow encountered an error

The `getWorkflowTokenStatus` operation is declared as follows.

```java
public String[] getWorkflowTokenStatus(String[] workflowTokenID, String username, String password);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array of strings</td>
<td>workflowTokenId</td>
<td>List of workflow token IDs.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns a list of workflow token status values. The returned value is a string array of the `globalStatus` of each workflow token, ordered by their `workflowTokenID` values. Returns null if you pass it an invalid parameter.

**Related Information**

For related information, see “WorkflowToken Object,” on page 185.

**hasChildrenInRelation Operation**

The `hasChildrenInRelation` operation checks whether a given relation type has any children.

In some cases, objects are most easily located through their relationships with other objects. You can obtain all the objects that relate to another object by a given relation by calling the `findRelation` operation on that object. The `findRelation` operation finds only the relatives of a known object. The `hasChildrenInRelation` operation checks for the presence of objects that present a given relation property. `hasChildrenInRelation` checks for the presence of objects that are children of other objects and are related to their parents by a given relation type. For example, a snapshot of a virtual machine is a child of the original virtual machine. Checking for all virtual machines that are children of other virtual machines enables you to identify all snapshots.

Knowing how a child is related to its parent is useful if you develop tree viewers to view the objects in the library. The `hasChildrenInRelation` operation is declared as follows.

```java
public int hasChildrenInRelation(String parentType, String parentId, String relation, String username, String password);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>parentType</td>
<td>Type of parent object. You can narrow the search by specifying the parent type, which limits the result to children related by the given relation to parents of a given parent type. This value can be null, in which case <code>hasChildrenInRelation</code> checks for child objects related by the specified relation type to all types of parent.</td>
</tr>
<tr>
<td>String</td>
<td>parentId</td>
<td>ID of a particular parent object. Specifying the <code>parentId</code> allows you to check for children related by a given relation to a particular parent. This check is useful if a particular parent has large numbers of children that are related to it by different relation types. The <code>findRelation</code> operation returns all of that parent's children, regardless of the relation type. <code>hasChildrenInRelation</code> checks for the presence of only the children related by the desired relation type. This value can be null if you call <code>hasChildrenInRelation</code> on the root object of the hierarchy of objects.</td>
</tr>
<tr>
<td>Type</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>String</td>
<td>relation</td>
<td>The type of relation by which children are related to their parents. Relation types are specified in the vso.xml file for each plug-in.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns one of the following values:

- 1  Yes, children of the specified relation type are present
- -1 No, children of the specified relation type are not present
- 0  Unknown, or an input parameter is invalid

**Related Information**

For more information, see “findRelation Operation,” on page 191.

**hasRights Operation**

The hasRights operation checks whether a user has permissions to view, edit, and run workflows.

To check what rights you have on an item in a workflow, you must have permission to view this item. If you have only edit or run permission on an item, you cannot view what rights you have on this item, and hasRights returns False.

You set a user’s rights for a workflow in the Authorizations pane in the Orchestrator client. Your Web service application can check those rights by calling the hasRights operation. In the following example, hasRights checks whether the user has the right to read the workflow.

```plaintext
hasRights(taskId, username, password, 'r')
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>taskId</td>
<td>The ID of the workflow for which you are checking a user's rights.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
</tbody>
</table>
| Int   | rights| a: The administrator can change the object’s rights
|       |       | c: The user can edit the workflow.
|       |       | I: The user can inspect the workflow schema and scripting.
|       |       | r: The user can view the workflow (but not the schema or scripting).
|       |       | x: The user can run the workflow.  |

**Note**  User rights are not cumulative. To perform all possible tasks on a workflow, a user must have all of the rights.

**Return Value**

Returns the following values:

- True, if the user has the specified rights on the workflow.
- False, if the user does not have the specified rights on the workflow.

The hasRights operation returns an “Unable to find workflow” exception if the workflow does not exist or if the user calling hasRights does not have permission to view the workflow.
sendCustomEvent Operation

The sendCustomEvent operation synchronizes workflows with external events.

public void sendCustomEvent(String eventName, String serializedProperties);

The sendCustomEvent operation sends messages from Web service clients to workflows that are waiting for a particular event to occur before they run. The waiting workflows resume their run when they receive the message from sendCustomEvent.

A custom event that calls sendCustomEvent to send a message when it occurs can be any script, workflow, or action that Orchestrator can run. For example, a workflow might use sendCustomEvent to trigger another workflow that reloads all Orchestrator plug-ins when the sending workflow performs a specific action while it is running.

The messages that sendCustomEvent sends are simple triggers, the format of which is not exposed to users. The message triggers the waiting workflow to run at the moment that the server receives it.

IMPORTANT Access to the sendCustomEvent operation is not protected by a username and password combination. VMware therefore recommends that you only use this function in secure, internal deployments. For example, do not use this operation in deployments that operate openly across the Internet.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>eventName</td>
<td>The eventName property is the name of the event that a workflow is waiting for before running. The eventName string you pass to sendCustomEvent must match the name of an Event object declared in the script, action or workflow that defines the custom event.</td>
</tr>
<tr>
<td>String</td>
<td>serializedProperties</td>
<td>The serializedProperties property defines the parameters to pass to the waiting workflow as a series of name-value pairs. The syntax of serializedProperties is as follows: &quot;name1=value1\nname2=value2\nname3=value3&quot; If the workflow requires no input parameters, the serializedProperties property can be null or omitted.</td>
</tr>
</tbody>
</table>

Return Value

No return value informs applications that the sendCustomEvent operation ran successfully.

The sendCustomEvent operation returns an exception if you pass it an invalid parameter.

Receiving Messages from sendCustomEvent

Workflows waiting for a message from sendCustomEvent before they run must declare the event they are waiting for by calling the System.waitCustomEventUntil operation from the Orchestrator API. The following example shows two calls to waitCustomEventUntil.

System.waitCustomEventUntil("internal", customEventKey, myDate);
System.waitCustomEventUntil("external", customEventKey, myDate);

The waitCustomEventUntil operation's parameters are as follows.

<table>
<thead>
<tr>
<th>internal / external</th>
<th>The awaited event comes from another workflow (internal) or from a Web service application (external).</th>
</tr>
</thead>
<tbody>
<tr>
<td>customEventKey</td>
<td>The name of the awaited event.</td>
</tr>
<tr>
<td>myDate</td>
<td>The date until which waitCustomEventUntil waits for a message from sendCustomEvent.</td>
</tr>
</tbody>
</table>
**simpleExecuteWorkflow Operation**

The simpleExecuteWorkflow operation uses string attributes to start a workflow.

**IMPORTANT** This operation is deprecated. Do not use simpleExecuteWorkflow.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowId</td>
<td>ID of the Workflow to be run.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
<tr>
<td>String</td>
<td>attributes</td>
<td>The format for the attributes parameter is a list of attributes separated by commas. Because commas are used as separators, attribute name strings containing commas are not processed correctly. Each attribute is represented by its name, type, and value, as shown in the following examples. Name1,Type1,Value1,Name2,Type2,Value2</td>
</tr>
</tbody>
</table>

**Return Value**

Runs a workflow. Returns a WorkflowToken object.
Developing Web Views

Orchestrator provides a Web 2.0 front end that includes a set of Web components to create browser-based user front ends.

This chapter includes the following topics:
- “Web View Overview,” on page 201
- “File Structure of a Web View,” on page 202
- “Web View Components,” on page 203
- “Web View Attributes,” on page 205
- “Create a Web View From a Template,” on page 205

Web View Overview

Orchestrator provides a library of customizable components to allow you to access orchestrated objects through a Web 2.0 front end, or Web view.

A Web view is a package of Web pages, style sheets, icons, and banners that represent a complete Web site. Web views can contain special components that give access to all Orchestrator capabilities from a browser-based client. You typically create Web pages externally by using Web design tools. You then import the Web pages into the Web view package. Importing completed Web pages into the Web view package allows you to separate the Web design process from the process of developing the Orchestrator Web view components.

Orchestrator Web views use Ajax technology to allow you to update content dynamically without reloading complete pages. Orchestrator provides a library of Tapestry components on the server side and Dojo components on the client side to help you build customized Web views.

Start a Web View

You start a Web view in the Orchestrator client. Orchestrator provides a Web view called weboperator that you can use to run workflows.

Procedure

1. Click Web Views in the Orchestrator client tab.
   - The weboperator Web view and any other Web views that you have imported into Orchestrator appear.
2. Right-click weboperator and select Publish.
3. Open a browser and go to http://orchestrator_server:8280.
   - In the URL above, orchestrator_server is the name or IP address of the Orchestrator server, and 8280 is the port number upon which Orchestrator publishes Web views.
4 In the Orchestrator home page, click **Web View List**.
5 Click **weboperator**.
6 Log in using your Orchestrator user name and password.

You can now access the workflows in the Orchestrator library from the weboperator Web view. You can use weboperator to interact with workflows across a network.

### Edit a Web View

You can edit existing Web views in the **Web Views** view of the Orchestrator client.

#### Procedure

1. Click the **Web Views** view in the Orchestrator client.
2. If the Web view is running, right-click the Web view and select **Unpublish**.
3. Right-click the Web view and select **Edit**.

### File Structure of a Web View

The file structure of an Orchestrator Web view must adhere to some basic rules.

A Web view application must contain the following file and folder at the root of the Web view folder.

```html
<WebView_Folder>/
default.html
```

The front page of the Web view.

```html
<WebView_Folder>/
components/
```

Contains the files of the Tapestry Web view component JWC files and the associated HTML pages and Java classes that form the Web view. You can create subfolders inside the `components` folder.

The default.html file and the `components` folder are the only mandatory elements that a Web view must contain. You must store all Tapestry component files in the `components` folder.

You can add other files and folders within the Web view folder and organize the files and folders in any way you want. You can include HTML files that are not associated to Tapestry components anywhere in the Web view folder.

For example, you can add the following optional folders to a Web view:

```html
<WebView_Folder>/css/
To contain CSS style sheets that define the appearance of the Web view.
```

```html
<WebView_Folder>/images/
To contain image files for the Web view, such as banners, logos, and icons.
```

```html
<WebView_Folder>/javascript/
To contain JavaScript files that define functions that the Web view can perform.
```
Web View Components

You build Orchestrator Web views by adding Tapestry components to HTML pages. Orchestrator provides a library of Orchestrator Tapestry components that add predefined orchestration functions to Web views. You can also create custom components to add other functions to Web views.

Working with Tapestry Components

A Tapestry Web view component consists of a specification file, template file, and a Java class file. You can use every component from the Tapestry Framework 4.0 standard in a Web view.

You add Tapestry components to a Web view by adding a \texttt{jwcid} attribute to an HTML tag in a Web page. A \texttt{jwcid} attribute is a Hibernate attribute that allows an HTML element to access a Tapestry component. Adding a \texttt{jwcid} to an HTML tag adds the function of the Tapestry component to the feature of Web page that the HTML tag defines. For example, you can add a Tapestry component that starts a workflow to an HTML link tag, so that a certain workflow starts when a user clicks the link in the Web view.

Web View Component Files

Orchestrator allows you to create custom Web view components to perform functions from Web pages. A Web view component conforms to the Tapestry Framework standard.

A Web view component must contain the following standard files that comprise a Tapestry component.

- **Tapestry component template**: An HTML file that contains the layout of a Tapestry component. A Tapestry component is a reusable object that you add to Web pages.
- **Tapestry component specification**: A file with a .\texttt{jwc} extension that refers to the Tapestry component Java class and to the Tapestry DTD definition.
- **Tapestry component Java class**: A Java class that specifies the behavior of the component.

\textbf{IMPORTANT} The HTML component template file and the Tapestry component specification must have the same name. For example, if you name a component template \texttt{MyComponent.html}, you must name the associated component specification \texttt{MyComponent.jwc}.

Initializing Tapestry Components

You initialize a Tapestry component by setting the \texttt{jwcid} attribute in an HTML tag in a Web view page.

To point the \texttt{jwcid} attribute to a Java class that defines a certain action, you prefix the name of the Tapestry component Java class that defines the behavior of the component with the @ character, as the following example shows.

\begin{verbatim}
<div jwcid="@Border">
\end{verbatim}
You can precede the @ character by a unique identifier. The unique identifier allows you to reuse the class throughout the HTML page, by referencing the unique identifier. In the following example, the class is Border and the unique identifier is myBorderComponent.

\[ jwcid="myBorderComponent@Border" \]

**Note** If you create subfolders in the components folder you must specify the full path to a component. For example, if you included a <WebView_Folder>/components/layout/ subfolder, you must set the jwcid attribute as the following example shows:

\[ <vmo jwcid="@layout/Border" \]

**Create a Web View Component**

To create a component, you must create a component specification file and a component template file. These files define the behavior of the component and how that component appears in HTML pages. You then add a jwcid attribute to HTML pages to add Orchestrator and Tapestry components to Web pages in Web views.

**Prerequisites**

You must have access to the Java classes of the Orchestrator Web view API.

**Procedure**

1. Create a folder named components at the root of your Web view folder.
2. In the components folder, create a component specification file with a .jwc extension.
   
   This file refers to an existing Java class that provides the function of this Web view component.
   
   For example, the following example file, StartVirtualMachine.jwc, references the Tapestry DTD and specifies the Java class of the WorkflowLink component from the Orchestrator Web services API.
   
   \[
   \begin{verbatim}
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE component-specification
   PUBLIC "-//Apache Software Foundation//Tapestry Specification 4.0//EN"
   "http://jakarta.apache.org/tapestry.dtd/Tapestry_4_0.dtd">
   <component-specification class="ch.dunes.web.webview.components.WorkflowLink"
   allow-body="yes" allow-informal-parameters="yes"
   </component-specification>
   \end{verbatim}
   \]
3. In the components folder, create the associated template file with an .html extension.
   
   For example, create an HTML page named StartVirtualMachine.html.
4. Add a jwcid attribute to an HTML tag in the HTML template page to start an Orchestrator or Tapestry component.
   
   For example, adding the following example jwcid attribute to the StartVirtualMachine.html page instantiates an Orchestrator WorkflowLink component that starts a given workflow when you click the appropriate link in the Web view.
   
   \[
   <a jwcid="@WorkflowLink" workflow="startVmWorkflow">Start a Virtual Machine</a>
   \]

   The jwcid attribute runs a workflow that you identify by the Web view attribute startVmWorkflow. You define the startVmWorkflow attribute in the Orchestrator client.
Common Web View Component Parameters

Web view components obtain data by different means. You define how a Web view component gets data through its parameters.

The following component parameters obtain data for a Web view component. You declare the parameters in the `jwcid` attribute in the component HTML template.

- **url**: A URL pointing to the data that the component requires.
- **action**: A Web view attribute of the `Action` type that contains the action that retrieves data.
- **actionParameters**: An array of parameters to pass to the action that retrieves data.
- **actionPageUrl**: A URL from which an action retrieves data.
- **attribute**: An attribute that contains the data that the component requires.

Web View Attributes

Web view attributes allow you to pass objects to Web view components. The functions that the Web view components define act upon these objects to perform the orchestration actions that you run from the Web view.

A Web view attribute can be an object of any type that the Orchestrator API supports. For example, a Web view attribute can be a `VC:VirtualMachine` object. A Web view component can define a function that requires this object as an attribute. For example, when a user clicks a button in a Web view, a Web view component associated to that button runs a workflow that starts a virtual machine. A Web view attribute provides the virtual machine object to the workflow that the Web view component starts.

You define Web view attributes in the Orchestrator client.

Create a Web View From a Template

Orchestrator provides a Web view template to help you create Web views. The easiest way to create a Web view is to use this template.

The process for creating a Web view from the template involves importing the template into Orchestrator and modifying the files it contains to suit your purposes. This information describes how to modify this template to make a Web view called Virtual Machine Manager.

**Procedure**

1. **Import the Web View Template** on page 206
   - Orchestrator provides a Web view template that you can adapt to create Web views.

2. **Export the Web View to a Working Folder** on page 207
   - To ease development of Web views, export the workflow template to a local folder in which you can work on it.

3. **Configure the Server For Web View Development** on page 207
   - You can configure the Orchestrator server to publish the Web view from your working folder. This process allows you to preview the Web view as you develop it, without having to import it into the Orchestrator server.

4. **Edit the Web View Front Page** on page 209
   - You create the front page of the Web view in the `default.html` file.
5 Create an Action to Obtain Virtual Machines on page 209
You can customize the front page of the Virtual Machine Manager Web view to display a list of all the virtual machines in a virtual folder. You must create an action to obtain virtual machines.

6 Set the Web View Authorizations on page 210
You define which users have permission to access a Web view by setting authorizations.

7 Set the Web View Attributes on page 211
You set the Web view attributes to direct it to the operations that it performs. You set the Web view attributes by editing the Web view in the Orchestrator client.

8 Add a Component to a Web Page on page 211
You can add components to pages in Web views that perform orchestration operations.

9 Customize the Web View Interface on page 212
The final stage of development of a Web view is to customize the interface.

Import the Web View Template
Orchestrator provides a Web view template that you can adapt to create Web views.
When you create Web views, it is easier to customize an existing template than to start from the beginning.

Prerequisites
You must have connected Orchestrator to a running vCenter Server 4.0 instance that contains some virtual machines.

Procedure
1 Click the Web Views view in the Orchestrator client.
2 Click the menu button (⋮) in the Web Views header and select New from > File template to import a Web view template into the Orchestrator server.
3 Browse to the following folder:
   install_directory\VMware\Orchestrator\apps\webviewTemplates
4 Select the default_webview.zip file and click Open.
5 Name the Web view Virtual Machine Manager.
The Virtual Machine Manager Web view now appears in the list in the Web Views view in the client.
6 (Optional) Right-click Virtual Machine Manager and select Publish to preview the empty template.
7 (Optional) In a browser, go to http://orchestrator_server:8280/vmo/ to see the list of Web views running on the Orchestrator server.
8 (Optional) Click Virtual Machine Manager and log in using your Orchestrator username and password.
   You see a basic Web view, with no operations or functions.
You created an empty Web view from a template, and optionally inspected it in a browser.

What to do next
Export the empty Web view to a working folder in which you can work on it.
Export the Web View to a Working Folder

To ease development of Web views, export the workflow template to a local folder in which you can work on it.

**Prerequisites**

You must have imported the Web view template into Orchestrator, and used it to create an empty Web view.

**Procedure**

1. Create a folder in which to develop the Virtual Machine Manager Web view.
   
   For example, `work_dir/`.

2. In the **Web Views** view of the Orchestrator client, right-click Virtual Machine Manager and select **Export to directory**.

3. Navigate to your working folder and click **Export**.

   Orchestrator creates a folder in `work_dir/` that contains the Web view. For example, if you export the Virtual Machine Manager Web view, Orchestrator creates a folder named `virtual_machine_manager/` in the working folder.

You exported the empty Web view to a working directory.

**What to do next**

Configure the Orchestrator server for Web view development.

Configure the Server For Web View Development

You can configure the Orchestrator server to publish the Web view from your working folder. This process allows you to preview the Web view as you develop it, without having to import it into the Orchestrator server.

You set the Orchestrator server to Web view development mode in the Orchestrator configuration interface. When the server runs in development mode, all web views are loaded from the development folder rather than from the Orchestrator server’s library of Web views.

**Note** Because Orchestrator publishes Web views from the development folder when the server is in development mode, you must export all Web views to the same development folder.

**Prerequisites**

You must import the Web view template, create the Virtual Machine Manager Web view, and export it to a working directory.

**Procedure**

1. Open the Orchestrator configuration interface in a browser, at the following URL:

   - `http://orchestrator_server:8282` if you connect to the server using HTTP.
   - `https://orchestrator_server:8283` if you connect to the server securely using HTTPS.

2. Log in to the configuration interface using the configuration username and password.

   The Orchestrator administrator set the username and password when they first installed Orchestrator.

3. Click the **Advanced Configuration** tab in the **General** view.

4. Check the **Webview development enable** check box.
5 Provide the path to the root of your working folder in the text box.

**Note** Be sure to provide the path to the root of the working folder. Do not include the folder that contains the workflow itself in the path.

6 Click the **Apply Settings** button.

7 Click the **Startup Options** view.

8 Click **Restart Service** to restart the Orchestrator server in Web view development mode.

You set the Orchestrator server to Web view development mode, which allows you to preview a Web view while you develop it.

**What to do next**
Create the Web view front page.

**Contents of the Exported Web View**
The Web view template contains the files that you must modify to create a custom Web view.

After you export the Virtual Machine Manager Web view, the `work_dir/vm_manager` folder contains the following folders and files.

- **/components**
  - Contains the following Tapestry component files.
  - **Border.jwc** Defines the default template component that the Web view pages use
  - **Border.html** Defines the default structure of the Web view pages

- **/css**
  - Contains the following style sheets.
  - **border.css** The style sheet that renders the border template
  - **custom.css** Allows you to customize other stylesheets, such as `webform.css` or `common.css` that Orchestrator provides to render complex components, such as forms. Every page of the Web view other than `Border.html` imports the `custom.css` file. The `custom.css` file overrides the other style sheets with the layout you define.

  **Note** The `custom.css` style sheet is the only style sheet you can edit. If you edit the other system style sheets directly rather than editing `custom.css`, your edits will be overwritten every time you upgrade to a new version of Orchestrator.

  - **login.css** Allows you customize the login page.

- **/images**
  - Contains all the images for the default look and feel of the Web view.

- **/default.html**
  - The main page of the Web view.

- **/login.html**
  - Defines the appearance of the Web view login page.

- **/login.page**
  - Defines the behavior of the Web view login page.
Edit the Web View Front Page

You create the front page of the Web view in the `default.html` file.

**Prerequisites**

You must have imported the Web view template into Orchestrator, exported its contents to a working directory, and configured Orchestrator in Web view development mode.

**Procedure**

1. Go to the root of your working directory.
2. Open the `default.html` file in an HTML editor.

   The `default.html` page uses the `Border` component to render itself. It contains very little code, as the following code sample shows.

   ```html
   <!-- Load the border located in ~/components/layout/Border.jwc -->
   <vmo jwcid="@layout/MyBorder" section="literal: home" title="Home">
     <!-- Content of the homepage -->
     <h2 style="margin-left: 16px; margin-top: 0px; padding-top:18px;">
       Welcome to Default Webview Template
     </h2>
     <p style="margin-left: 16px;">
       This webview is a base for your own webview development.
     </p>
   </vmo>
   ```

   The `vmo` tag initializes a Tapestry component by setting the `jwcid` attribute to point to `Border`. The `Border` component renders the layout of the page. Content you place in the `vmo` tag renders in the body of the component.

**What to do next**

You can add functions to the `default.html` page.

Create an Action to Obtain Virtual Machines

You can customize the front page of the Virtual Machine Manager Web view to display a list of all the virtual machines in a virtual folder. You must create an action to obtain virtual machines.

**Prerequisites**

You must have imported the Web view template into Orchestrator, exported its contents to a working directory, and configured Orchestrator in Web view development mode.

**Procedure**

1. Click the Actions view in the Orchestrator client.
2. Right-click the root of the actions hierarchical list and select New module to create a new action module.
3. Name the new module `org.my_company.vmManager`.
4. Right-click `org.my_company.vmManager` in the hierarchical list of modules and select Add Action.
5. Name the new action `getVmByFolder`.
6. Open the action workbench by right-clicking `getVmByFolder` and selecting Edit.
7. Click the Scripting tab.
8 Right-click in the parameters pane of the Scripting tab and select Add parameter.
9 Name the parameter **vmFolder**.
10 Click the **vmFolder** parameter's Type link and select **Array of > VC:VirtualMachine**
11 Add the following JavaScript code in the scripting pad in the Scripting tab.
   ```javascript
   return vmFolder.vm
   ```
12 Click the Return type:Void link to set the return type for the action.
13 Select **Array of: > VC:VirtualMachine** from the selection dialog box and click Accept.
14 Click **Save and Close** to close the actions workbench.

You created an action that returns an array of virtual machines from a folder.

**What to do next**
Set the Web view authorizations.

## Set the Web View Authorizations

You define which users have permission to access a Web view by setting authorizations.

You set authorizations for Web views in the **Authorizations** view in the Orchestrator client.

**Prerequisites**
You must have created the **getVmByFolder** action to obtain an array of virtual machines for the Virtual Machine Manager Web view.

**Procedure**
1 Click the **Authorizations** view in the Orchestrator client.
2 Click the menu button (四方) in the **Authorizations** view header and select Create authorization.
3 Provide a name for the authorization in the Name text box.
4 Click Not set next to LDAP Group to select an LDAP user group to authorize to use the Web view.
5 Provide a description of the authorization in the Description text box.
6 Click Create.
7 Right-click the authorization and select Edit.
8 Click the **References** tab.
9 Click Create reference.
10 Select a folder from the vCenter Server inventory hierarchical list and click Select to authorize access to this folder.
11 Click Create reference again and select the getVm action.
12 Click Save and Close to exit the authorizations workbench.

You set the permission to access the folder and action that the Virtual Machine Manager Web view accesses.

**What to do next**
Set the Web view attributes.
Set the Web View Attributes

You set the Web view attributes to direct it to the operations that it performs. You set the Web view attributes by editing the Web view in the Orchestrator client.

Prerequisites

You must have created the `getVmByFolder` action to obtain an array of virtual machines for the Virtual Machine Manager Web view, and set the authorizations to access it.

Procedure

1. Click the Web Views view in the Orchestrator client.
2. Right-click Virtual Machine Manager and select Unpublish.
   
   You cannot edit a published Web view.
3. Right-click Virtual Machine Manager and select Edit.
4. Click the Attributes tab in the Web view workbench.
5. Right-click in the Attributes tab and select Add attribute.
6. Click the attribute name and name it `getVirtualMachineList`.
   
   Click the `getVirtualMachineList` Type link and select Action.
7. Click the `getVirtualMachineListValue` link and select the `getVmByFolder` action you created earlier.
8. Repeat the preceding steps to create another attribute with the following properties.
   - Name: `vmFolder`
   - Type: VC:VmFolder
   - Value: A virtual folder from your inventory that contains virtual machines.
9. Click Save and Close to exit the Web view workbench.

You defined attributes that direct the Web view to the action that you created and to the folder of virtual machines it accesses.

What to do next

Add a component to a Web view page to access the action you defined.

Add a Component to a Web Page

You can add components to pages in Web views that perform orchestration operations.

Prerequisites

You must have the following tasks completed:

- Create the `getVmByFolder` action to obtain an array of virtual machines for the Virtual Machine Manager Web view.
- Set the authorizations to access the view.
- Set the Virtual Machine Manager Web view’s attributes to access this action and a folder of virtual machines.
Procedure

1. Open the `default.html` Web view file in an HTML editor.

2. Add a `vso:ListPane` Component to the `default.html` file to list all of the virtual machines in the page.

   You add `vso:ListPane` Component by adding the following code between the `vmo` tags in the `default.html` file.

   ```xml
   <vmo jwcid="@layout/MyBorder" section="literal: home" title="Home">
       <hl>Virtual Machine Manager</hl>
       <div jwcid="vmList@vso:ListPane"
           action="getVirtualMachineList"
           actionParameters="attribute:vmFolder"
           detailUrl="<myWebViewComponent>.html">
           Select a virtual machine on the left to display it.
       </div>
   </vmo>
   
   The `vmo` tag instantiates the `vso:ListPane` component, as follows.

   
   - `jwcid="vmList@ListPane"` 
     \( \text{vmList} \) is the unique identifier of the component and `@ListPane` is the component class name.
   
   - `action="getVirtualMachineList"` 
     Links to the Web view attribute that contains the name of the action to perform.
   
   - `actionParameters="attribute:vmFolder"` 
     Uses an OGNL statement to pass attributes to the action. The preceding code passes the `vmFolder` Webview attribute to the action.
   
   - `detailUrl="<myWebViewComponent>.html"` 
     The path to an HTML Web view component template that you have created.

   You added a component to the Virtual Machine Manager Web view that performs the action that you defined to obtain virtual machines from a given folder in the vCenter Server.

What to do next

Customize the interface.

Customize the Web View Interface

The final stage of development of a Web view is to customize the interface.

Prerequisites

You must have added a component to the Web view front page to perform an action to obtain virtual machines.

Procedure

1. In a browser, go to the Virtual Machine Manager Web view and refresh the page.

   The component you added is present in the Web view, but you must modify the layout of the page.

2. Open the `work_dir/vm_manager/css/custom.css` style sheet file.

3. Add the following code to `work_dir/vm_manager/css/custom.css`.

   ```css
   .vso_listPane {
      border: 1px solid #CCC;
      height: 300px;
      margin-bottom: 1em;
   }
   ```
4 Refresh the page in the browser.

The component has a thin border and a fixed height.

You customized the appearance of the Web view by adjusting the custom.css style sheet.

**What to do next**

Customize any part of the page by modifying the custom.css file. Override the Border.html file and border.css to modify the overall layout. You can override any existing component presentation by using partial templates and style sheets. You can create Tapestry components.
Refactoring Orchestrator Applications After Upgrading vCenter Server

If you upgrade the virtual infrastructure from VMware Infrastructure 3.5 to vCenter Server 4.0, you must refactor any Orchestrator applications that you wrote for use with the older version. Orchestrator 4.0 provides workflows to help you refactor the applications to the new version. Alternatively, you can run VMware Infrastructure 3.5 on Orchestrator 4.0 by installing the VMware Infrastructure 3.5 plug-in.

This chapter includes the following topics:

- “When to Refactor Applications,” on page 215
- “Install the VMware Infrastructure 3.5 Plug-In,” on page 216
- “Refactoring Packages with the Basic Refactoring Workflow,” on page 216
- “Refactoring Packages with the Advanced Refactoring Workflows,” on page 220

When to Refactor Applications

If you upgrade the virtual infrastructure from VMware Infrastructure 3.5 to vCenter Server 4.0, you must take action to continue to run your existing applications.

You develop Orchestrator applications as plug-ins. Plug-ins consist of one or more packages that can contain workflows, actions, Java classes, XML files, Web views, configuration elements, or policy templates. The package names and certain object types used by the vCenter Server 4.0 API are different from those used by previous versions of VMware Infrastructure. The name and type changes are listed in Table 9-1.

If you upgrade the virtual infrastructure from VMware Infrastructure 3.5 to vCenter Server 4.0, one course of action is to install the VMware Infrastructure 3.5 plug-in on an Orchestrator server 4.0 platform and then import your applications. The VMware Infrastructure 3.5 plug-in communicates with the vCenter Server 4.0 plug-in and allows you to run your applications without change.

Alternatively, to benefit from all the features of vCenter Server 4.0, you can refactor Orchestrator applications that you wrote with the old version. Orchestrator 4.0 provides workflows to help you refactor the applications to the new version.

Table 9-1. Object Name Changes from Previous Versions of VMware Infrastructure to vCenter Server 4.0

<table>
<thead>
<tr>
<th>Object</th>
<th>Value in Previous Versions of VMware Infrastructure</th>
<th>Value in vCenter Server 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package name</td>
<td>vim3</td>
<td>vcenter</td>
</tr>
<tr>
<td>FinderResult type</td>
<td>VIM3</td>
<td>VC</td>
</tr>
<tr>
<td>Scripting type</td>
<td>Vim</td>
<td>Vc &lt;ScriptName&gt;</td>
</tr>
<tr>
<td>Host</td>
<td>VMware3:VimHost</td>
<td>VC:SdkConnection</td>
</tr>
</tbody>
</table>
Because of these name and type changes, you must update all the VMware Infrastructure function calls made by the applications so that they can find the functions in the vCenter Server 4.0 API packages, through the vCenter Server 4.0 plug-in.

**IMPORTANT** To avoid accidentally overwriting packages during the refactoring process, you must back up your packages and applications before you run the refactoring workflows. If the refactoring fails, restore your server to its previous state.

In addition, when you refactor an application, all the refactored workflows have new workflow IDs. If you have applications, for example, Web service clients, that access workflows by using their IDs, you must update those workflow IDs accordingly.

### Install the VMware Infrastructure 3.5 Plug-In

To continue running applications you developed for VMware Infrastructure 3.5 with Orchestrator 4.0, one solution is to install the optional VMware Infrastructure 3.5 plug-in. The VMware Infrastructure 3.5 plug-in is delivered with Orchestrator 4.0, but is not installed by default.

If you install the VMware Infrastructure 3.5 plug-in, you can run applications you developed for VMware Infrastructure 3.5. However, you will not benefit from the features of vCenter Server 4.0 with these applications. To benefit from the features of vCenter Server 4.0, refactor your VMware Infrastructure 3.5 applications.

**Procedure**

1. Open the Orchestrator configuration interface in a browser, at the following URL and log in.
   
   http://<orchestrator_server_ip_address>:8282

2. In the **General** tab, click **Install Application**.

3. Browse to one of the following locations.
   - c:\Program Files\VMware\Orchestrator\extras\plugins if you installed the standalone version of Orchestrator.
   - c:\Program Files\VMware\Infrastructure\Orchestrator\extras\plugins if the vCenter Server installed Orchestrator.

4. Select vmo_vi35_4_0_0_<build_number>.vmoapp and click **Open**.

5. Click **Install**.

6. Click **Startup Options**.

7. Click **Restart Service** to restart the Orchestrator server.

You installed the VMware Infrastructure 3.5 plug-in.

**What to do next**

You can run applications you wrote for VMware Infrastructure 3.5 using Orchestrator 4.0.

### Refactoring Packages with the Basic Refactoring Workflow

Orchestrator provides a basic workflow to help you refactor most packages so that they access vCenter Server 4.0.

The refactoring workflow makes a copy of an existing VMware Infrastructure 3.5 Orchestrator package and modifies all the elements in the copy to use vCenter Server 4.0. When you run the workflows, the original VMware Infrastructure 3.5 application remains untouched, but the new duplicate is updated to run using vCenter Server 4.0.
Set Up the Refactoring Tutorial Example Application (Optional)

You can use the Refactor Tutorial example to experiment with the refactoring workflows. You can import the example package into Orchestrator and set up the example.

The Refactor Tutorial example consists of a package containing two basic workflows and a Web view.

- The Suspend virtual machine workflow requests the suspension of a virtual machine running in a VMware Infrastructure 3.5 inventory and waits for the VMware Infrastructure to complete the suspension request.
- The Resume virtual machine workflow requests the resumption of the suspended VMware Infrastructure 3.5 virtual machine and waits for the VMware Infrastructure to complete the resumption request.
- The Refactor Tutorial Web view allows you to run the Suspend virtual machine and Resume virtual machine workflows from a Web interface in a browser.

Initially, the Refactor Tutorial Web view accesses and interacts with virtual machines running in the VMware Infrastructure 3.5 inventory. After you run the refactoring workflows, this Web view application accesses and interacts with the virtual machines in the vCenter Server 4.0 inventory instead.

Prerequisites

- You must have the VMware Infrastructure 3.5 and the vCenter Server 4.0 plug-ins installed on the Orchestrator server.
- Orchestrator must be connected to a VMware Infrastructure 3.5 server and to a vCenter Server 4.0 server, with virtual machines running in both.
- You must have downloaded the ZIP file of examples from the Orchestrator documentation index page. See “Example Applications.”

Procedure

1. Unzip the ZIP file of examples to an appropriate location.
2. Open <installation_directory>/Refactoring.
   - The Refactoring folder contains com.vmware.refactor.tutorial.package.
   - Click the Packages view in the left pane of the Orchestrator client.
   - Right-click anywhere on the whitespace under the list of packages and select Import Package.
   - Locate com.vmware.library.refactor.tutorial.package and click Open.
   - Information about the package certificates appears.
   - Click Import.
   - The package contents appear, with a checklist of elements to import.
   - Click Import checked elements.
4. Select Documentation > Refactor tutorial to check that the Refactoring Tutorial workflows are present in the Workflows view in the left pane.
5. Start the Refactor Tutorial Web view.
   - Click the Web Views view in the left pane.
   - Right-click the Refactor Tutorial Web view and select Publish to start it.
6. In a Web browser, go to http://Orchestrator_server:8280, in which Orchestrator_server is the name or IP address of the machine on which the Orchestrator server is running.
Select Web View List > Refactor Tutorial.

Log in using the same user name and password that you use to access the Orchestrator client.

You installed the Refactor Tutorial example. The Refactor Tutorial Web view allows you to suspend and resume virtual machines running in the VMware Infrastructure 3.5 server. You can experiment with refactoring an application from VMware Infrastructure 3.5 to vCenter Server 4.0.

What to do next
You can refactor this example application.

Run the Basic Refactoring Workflow

The basic refactoring workflow successfully refactors most VMware Infrastructure 3.5 applications so that they use the vCenter Server 4.0 API.

Prerequisites
- You must have the VMware Infrastructure 3.5 and the vCenter Server 4.0 plug-ins installed on the Orchestrator server.
- Orchestrator must be connected to a VMware Infrastructure 3.5 server and to a vCenter Server 4.0 server, with virtual machines running in both.
- You must have imported the com.vmware.library.refactoring package in the Orchestrator client.
- You must have a VMware Infrastructure 3.5 Orchestrator application to refactor. For example, the com.vmware.refactor.tutorial example.

Procedure
1. Click the Workflows view in the left pane of the Orchestrator client interface.
2. Select Library > Refactoring in the workflows hierarchical list to view the refactoring workflows.
3. Right-click the Migrate Package to vCenter Server 4 workflow and select Execute Workflow.
4. Under Package, click the Source Package Not Set link and select the package to refactor from VMware Infrastructure 3.5 to vCenter Server 4.0.
   For example, com.vmware.refactor.tutorial.
5. Enter a destination package in the Destination Package text box.
   You use the destination package to create the refactored application.
   For example, com.vmware.refactor.tutorial_vcenter40.
6. Under Rules, point the refactoring workflow to the different sets of objects that the application contains, and provide destinations in which to make copies of these objects.
   If you do not provide destinations, the refactoring tool does not perform the refactoring. If the application does not feature a certain type of object, leave that text box set to Not set. For example, the refactor tutorial example has one Web view object and one workflow object, so you set the following source and destination locations.

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Source Location</th>
<th>Destination Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Rules</td>
<td>Not set</td>
<td>N/A</td>
</tr>
<tr>
<td>Webview Rules</td>
<td>Refactor Tutorial</td>
<td>refactorutorial_copy</td>
</tr>
<tr>
<td>Workflow Rules</td>
<td>Refactor tutorial</td>
<td>Documentation/Refactor tutorial Copy</td>
</tr>
<tr>
<td>Actions Rules</td>
<td>Not set</td>
<td>N/A</td>
</tr>
</tbody>
</table>
The refactoring tutorial example has no configuration elements or actions, so you do not set these values. However, for applications that do contain configuration elements or actions, you must provide source and destination locations.

**NOTE** When you set the source location of action rules, you cannot select the location from a list. You must type the source location manually.

7 Under Save XML, click the **Not set** link to select a Resources category in which to save the XML log file of the refactoring workflow.

8 Click **Submit**.

The Migrate Package to vCenter Server 4 workflow runs.

The Migrate Package to vCenter Server 4.0 workflow copied and refactored the application so that the new version implements the vCenter Server 4.0 plug-in instead of the VMware Infrastructure 3.5 plug-in. The VMware Infrastructure 3.5 version of the application remains in place, unmodified.

**What to do next**

Verify that the refactoring workflow refactored the application correctly.

### Verify the Refactoring

When you have run the refactoring workflow, verify that the workflow refactored the application correctly.

**Procedure**

1 Click the **Packages** view in the Orchestrator client interface to check that the new package is present.

   In the Refactoring Tutorial example, a package called `com.vmware.refactor.tutorial_vcenter40` is listed.

2 Click the **Workflows** view to check that the refactored workflows are present.

   In the Refactoring Tutorial example, select **Documentation > Refactor tutorial copy** to view the Submit VM and Resume VM workflows that are in the workflows hierarchical list.

3 Check that the new workflows implement the vCenter Server 4.0 plug-in.
   a Click on one of the new workflows.
   b In the right pane, click the **Schema** tab.
   c In the schema diagram, double-click one of the elements to show the element in the element hierarchical list on the left.

   The vCenter Server 4.0 plug-in appears in the list.

   For example, in the Refactor Tutorial example, select **Refactor Tutorial Copy > Suspend VM** and double-click the **Suspend VM and wait** workflow element. The Suspend VM and wait workflow element is located in the **Library > vCenter Server > Virtual machine management > Power Management** node of the workflows hierarchical list. In the original version of the example, this element is in the VIM3 node rather than the vCenter Server node.

4 Click the **Web views** view.

5 Right-click the old **Refactor Tutorial** Web view and select **Unpublish**.

6 Right-click the new, refactored **Refactor Tutorial** Web view and select **Publish**.

7 In a browser, go to `http://<orchestrator_server>:8080/vmo/refactortutorial/default.html` to access the Refactor Tutorial Web view.

8 Log in to the Refactor Tutorial Web view with your Orchestrator user name and password.
9  Click Suspend VM.
10  Search for a virtual machine to suspend.

    If the virtual machines that you can suspend are in the vCenter Server 4.0 environment, the refactoring was successful.

You verified that you successfully refactored an application. Your application now implements the vCenter Server 4.0 plug-in.

**Refactoring Packages with the Advanced Refactoring Workflows**

If the basic refactoring workflow does not successfully refactor the VMware Infrastructure 3.5 application, you can try to refactor the application by using the advanced refactoring workflows.

The advanced refactoring workflows use the same workflows that the basic refactoring workflow calls upon. If the basic workflow does not work, or to define the output parameters of the refactoring workflows yourself, you can use the advanced refactoring workflows directly.

The advanced refactoring workflows make a copy of an existing VMware Infrastructure 3.5 Orchestrator application and modify all the elements in the copy to use vCenter Server 4.0. When you run the workflows, the original VMware Infrastructure 3.5 application remains untouched, but the new duplicate is updated to run using vCenter Server 4.0.

**Advanced Refactoring Workflows**

The advanced refactoring workflows use XML description files to define which applications to refactor, where to create the duplicate application, which elements in the application to refactor, where to find the refactoring rules, and so on. The advanced refactoring workflows include a workflow that creates these XML description files for you.

Table 9-2 shows the advanced refactoring workflows that Orchestrator provides.

### Table 9-2. Advanced Refactoring Workflows

<table>
<thead>
<tr>
<th>Workflow Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Library > Refactoring > Advanced > RefactorDescription XML | Requests user input to define the following information:  
  - The name and location of the application to refactor  
  - The rules for copying the package objects  
  - The types of elements to refactor in the duplicate application  
  - Names and locations for the resource files that the refactoring workflows use  
  This information is recorded in XML description files that the other refactoring workflows use when they run. |
| Library > Refactoring > Advanced > Copy VMware Infrastructure 3.5 application and migrate to vCenter Server 4.0 plug-in | Performs the following actions:  
  - Creates a duplicate of the application with the name and in the location specified in the XML description files.  
  - Updates the elements in the duplicate application so that they implement the vCenter Server 4.0 plug-in.  
  - Optionally provides a log of all the elements that the workflow has updated. This log is an XML mapping file that you can use when refactoring other applications. |
Table 9-2. Advanced Refactoring Workflows (Continued)

<table>
<thead>
<tr>
<th>Workflow Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library &gt; Refactoring &gt; Advanced &gt; Refactor - Copy</td>
<td>Performs the following actions:</td>
</tr>
</tbody>
</table>
| VMware Infrastructure 3.5 application and migrate to vCenter Server 4.0 plug-in with String input | n Creates a duplicate of the application with the name and in the location specified in an XML description you provide to the workflow as a String.  
 n Updates the elements in the duplicate application so that they implement the vCenter Server 4.0 plug-in.  
 n Optionally provides a log of all the elements that the workflow has updated. This log is an XML mapping file that you can use when refactoring other applications. |
| Library > Refactoring > Advanced > Refactor - Execute refactor                 | Performs refactoring without input parameters. This workflow is run by the Refactor - Copy VMware Infrastructure 3.5 application and migrate to vCenter Server 4.0 plug-in workflow. If necessary, you can modify this workflow to define refactoring rules specific to the application. This is an advanced use case. |
| Library > Refactoring > Advanced > Refactor - Execute refactor using resources | Performs refactoring according to rules defined in a resource element (an XML file) instead of in an XML string. This workflow is run by the Refactor - Copy VMware Infrastructure 3.5 application and migrate to vCenter Server 4.0 plug-in workflow. If necessary, you can modify this workflow to define refactoring rules specific to the application. This is an advanced use case. |

Run the Advanced Refactoring Workflows

If the basic Migrate Package to vCenter 4.0 workflow does not successfully refactor your application, you can run the advanced refactoring workflows.

**Prerequisites**

- You must have the VMware Infrastructure 3.5 and the vCenter Server 4.0 plug-ins installed on the Orchestrator server.
- You must have imported the `com.vmware.library.refactoring` package in the Orchestrator client.
- You must have a VMware Infrastructure 3.5 Orchestrator application to refactor. For example, the `com.vmware.refactor.tutorial` example.

**Procedure**

1. Run the Refactor - Create RefactorDescription XML workflow to create a `copy.xml` XML description file.
2. Run the Refactor - Create RefactorDescription XML workflow again to create an `update-references.xml` XML description file.
3. Run the Refactor - Copy VMware3 based application and migrate to use vCenter 4.0 plugin workflow to refactor the application and migrate to the vCenter Server 4.0 plug-in workflow.

   To run the refactoring workflow, you must pass to it the `copy.xml` file and the `update-references.xml` file.

You refactored your application by running the advanced refactoring workflows directly.
Index

A
Action element 21
action elements, binding 65
Action view 101
actions
    adding 102
    attributes 104
    basic guidelines 103
    binding 66
    coding guidelines 103
    creating 82, 102
    finding elements that implement 103
    naming 104
    parameters 104
    reusing 101
Actions 101
Actions view 102
advanced refactoring workflow 220, 221
answerWorkflowInput 178, 187
API Explorer, accessing 107
attributes
    definition 18, 60
    read-write properties 76, 97
authorizations 210

B
basic refactoring workflow 216, 218
basic refactoring workflow, set up example 217
binding
    action elements 65
    decision elements 64
    scriptable tasks 68
bindings
    action 66
    defining 30, 86, 87
    exception 34, 76
    scriptable tasks 69
Boolean choices 33

C
cancelWorkflow 178, 188
check-pointing 9
configuration elements, creating 52
creating workflows 15
Custom Decision element 21

D
decision element, bindings 64
Decision element 21
decision elements, linking 32
Decision elements 33
dunesUri 182, 186

E
echo operation 171
echoWorkflow 188
End Workflow element 21
exception bindings, creating 34
exception handing 34
exceptions binding 76
executeWorkflow 189
executeWorkflow operation 177

F
find 183, 189
find operation 171, 172
FinderResult 171, 173, 174, 182
findForId 191
findForId operation 171, 173
findRelation 171, 174, 191

G
getAllPlugin 193
getAllPlugins 193
gGetAllPlugins operation 171
gGetAllWorkflows 175, 176, 194
gGetWorkflowForId 194
gGetWorkflowForId 175, 176
gGetWorkflowsWithName 175, 176, 194
gGetWorkflowTokenForId 178
gGetWorkflowTokenForId 195
gGetWorkflowTokenResult 180, 196
gGetWorkflowTokenStatus 178, 196
globalStatus 196

H
hasChildrenInRelation 197
HasChildrenResult Enumeration 163
hasRights 198
hasRights operation 177
I
IDynamicFinder interface 157
IN bindings 30
input parameters
definition 82
obtaining from user 36
properties 37
providing during run 40
setting properties 37
input parameters dialog box, creating 78, 98
input parameters, obtaining from user 35
IPluginAdaptor 125
IPluginAdaptor interface 126, 144, 157
IPluginEventPublisher interface 158
IPluginFactory 125, 146
IPluginFactory interface 126, 158
IPluginNotificationHandler 159
IPluginPublisher interface 159

J
JavaScript 105, 110

L
linking
decision elements 32
schema elements 28, 62, 84
long-running workflows
date object 48
Date object 47
timer-based 49
Trigger object 47
Triggeer-based 51

M
ModuleInfo 183
Mozilla Rhino JavaScript engine 105

N
nested workflows 46

O
Orchestrator API 101, 106
Orchestrator architecture 11
OUT bindings 30

P
packages
create 119
digital rights management 119
permissions 120
signature 119
parameter properties
dynamic 37
static 37
parameters
definition 18, 60, 82
properties 77
read-write properties 76
persistence 9
plug-in adapter, creating 126
plug-in API
HasChildrenResult Enumeration 163
IDynamicFinder interface 157
IPluginAdaptor interface 157
IPluginEventPublisher interface 158
IPluginFactory interface 158
IPluginNotificationHandler 159
IPluginPublisher interface 159
PluginExecutionException 160
PluginOperationException 160
PluginTrigger 160
PluginWatcher 161
QueryResult 162
ScriptingAttribute annotation 164
ScriptingFunction annotation 164
ScriptingParameter annotation 165
SDKFinderProperty class 162
plug-in factory, creating 126
plug-ins
*.dar file 141, 155
adapter 125, 126
adaptor 144
components 124
creating 142
define finders 151
developing 123
enumerations 153
example application 142
example JAR files 144
factory 125, 126, 146
mapping events 153
mapping objects 154
naming objects 140
vso.xml file 149
PluginExecutionException 160
PluginOperationException 160
PluginTrigger 160
PluginWatcher 161
policy engine 9
presentation
creating 78, 98
creating display groups 98
display groups 35
input steps 35
Presentation tab 35, 37, 98
Presentation Tab 37
presentations 16
properties
  parameter 77
  read-write 76
Property 183

Q
QueryResult 162, 172, 183

R
refactoring
  advanced workflow 220, 221
  basic workflow 216, 218
  verify 219

S
schema
  bindings 29, 30
  custom decisions 32
  data flow 29, 30
  decisions 28, 32
  exception path 27, 28
  links 27, 28
  logical flow 27, 28
  standard path 27, 28
schema element, properties 24
schema elements
  binding 86, 87
  bindings 30
  decisions 33
  linking 28, 62, 84
  properties 25
  user interaction 40
schemas 16
Scriptable Task element 21
scriptable task elements, binding 68, 69
scripting
  access scripting engine from actions 107
  access scripting engine from policies 107
  access scripting engine from workflow 106
  access to Java classes 110
  adding objects 108
  adding parameters 109
  API Explorer 107
  auto-completion 108
  basic examples 113
  exception handling 111
JavaScript object types 108

scripted elements 105
shutter system property 110
scripting engine 9, 106
ScriptingAttribute annotation 164
ScriptingFunction annotation 164
ScriptingParameter annotation 165
SDKFinderProperty class 162
search, modifying results 21
search results 20
security 9
sendCustomEvent 199
simpleExecuteWorkflow 200
Start Workflow element 21
system properties 110

U
User Interaction element 21
user interactions
  attributes 40
  elements 40
user roles 10
using 101

V
versioning 9
viewing 101
VMware Infrastructure 3.5, install plug-in 216
vso.xml
  action element 129
  architecture 127
  attribute element 137
  attributes element 137
  constructor element 136
  constructor parameter element 137
  constructors element 136
  description element 129
  entries element 139
  entry element 140
  enumeration element 139
  enumerations element 139
  events element 134
  finder element 131
  finder-datasources element 130
  finders element 131
  gauge element 135
gauge-properties element 135
gauge-property element 135
  id element 133
installation element 129
inventory element 131
inventory-children element 133
method element 138
method parameter element 138
method parameters element 138
methods element 137
object element 136
parameters element 137
properties element 132
property element 132
relation element 133
relation-link element 134
relations element 133
scripting-objects element 136
singleton element 138
trigger element 134
trigger-properties element 134
trigger-property element 134
url element 139
webview-component-library element 130
vso.xml file
definition 127
elements 128
module element 128
vsoWebControl 171

W
waitForCustomEvent 199
Waiting Event element 21
Waiting Timer element 21
Web service
client creation process 167
download examples 182
endpoint 169
generating stubs 169
HTTP connection 171
HTTPS connection 171
obtain results 180
operation arguments 169
running workflows 177
WSDL description 169
Web service API
objects 182
operations 187
Web services
create client 170
finding objects 171–174
finding workflows 175, 176
time zones 181
workflow interaction 178
writing client application 167
Web view
overview 201
starting 201
weboperator 201
Web view component, create 204
Web views
attributes 205
authorizations 210
component files 203
component parameters 205
components 203
create action 209
create from template 205
editing 202
exporting 207
file structure 202
jwcid attribute 203, 204
specification file 203
Tapestry 203
Tapestry components 203
template 205, 206
template file 203
template files 208
workbench 202
Web views (Tapestry) 203
weboperator 201
Webviews 201
workflows, validation 55
workflow
attributes 17, 18, 60
create simple 57
creating 81
creating new 59
end 54
notes 63
parameters 17, 18, 60
presentation 16, 36
running 79, 98
schema 16
validation 79, 98
zones 63, 85
workflow categories 15
workflow development 13
workflow engine 9
workflow presentation, creating 36
workflow schema
bindings 26
create 20, 60, 83
edit 20
elements 20
links 26
schema element properties tabs 25
viewing 20
Workflow schema, schema element properties 24
workflow schema, elements 21
workflow token attributes 54
check points 54
workflow validation tool 55
workflow workbench
   General tab 16
   opening 15
tabs 16
WorkflowParameter 185
workflows
   asynchronous 42, 45
   branching 33
   creation 15
   editing 15
   nested 42
OGNL expression values 39
permissions 53, 54
phases of development 14
propagating changes 43
running 54, 56
scheduled 42, 45
starting 42
synchronous 42, 44
testing 14
validation 55
WorkflowToken 185
WorkflowTokenAttribute 177
WorkFlowTokenAttribute 186
workflowTokenId 196