Developing a Web Services Client for VMware vCenter Orchestrator

vCenter Orchestrator 5.1

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see http://www.vmware.com/support/pubs.

EN-000769-01

vmware®
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
Contents

Developing Web Services Client for VMware vCenter Orchestrator  7

Updated Information  9

1 Developing a Web Services Client  11

2 Using the vCenter Orchestrator REST API  13
   Authenticating Against Orchestrator and Third-Party Systems  14
      Using vCenter Single Sign On Authentication with the Orchestrator REST API  14
      Using LDAP Authentication with the Orchestrator REST API  15
   Accessing the Reference Documentation for the Orchestrator REST API  16
   Using the Java REST SDK  16
   Operations with Workflows  17
      Find a Workflow and Retrieve Its Definition  17
      Run a Workflow  19
      Run a Workflow After Validating Its Input Parameters Against the Workflow Presentation  21
      Interacting with a Workflow While It Runs  24
   Working with Tasks  30
      Create a Task  30
      Modify a Task  31
      Check the State of a Task  31
   Finding Objects in the Orchestrator Inventory  32
      Find Objects by Type and ID  32
      Find Objects by Relations  33
      Apply Filters  34
   Importing and Exporting Orchestrator Objects  34
      Import a Workflow  34
      Export a Workflow  35
      Import an Action  35
      Export an Action  35
      Import a Package  36
      Export a Package  36
      Import a Resource  36
      Export a Resource  37
      Import a Configuration Element  37
      Export a Configuration Element  37
   Deleting Orchestrator Objects  38
      Delete a Workflow  38
      Delete an Action  38
      Delete a Package  38
      Delete a Resource  39
      Delete a Configuration Element  39
WorkflowTokenAttribute Object 70

5 Web Service API Operation Reference 73
answerWorkflowInput Operation 74
cancelWorkflow Operation 74
echo Operation 75
echoWorkflow Operation 75
executeWorkflow Operation 75
find Operation 76
findForId Operation 77
findRelation Operation 78
getAllPlugins Operation 80
getAllWorkflows Operation 80
getWorkflowForId Operation 81
getWorkflowInputForId Operation 81
getWorkflowInputForWorkflowTokenId Operation 82
getWorkflowsWithName Operation 82
getWorkflowTokenBusinessState Operation 83
getWorkflowTokenForId Operation 83
getWorkflowTokenResult Operation 84
getWorkflowTokenStatus Operation 84
hasChildrenInRelation Operation 85
hasRights Operation 86
sendCustomEvent Operation 87
simpleExecuteWorkflow Operation 88

Index 91
Developing Web Services Client for VMware vCenter Orchestrator

Developing Web Services Client for VMware vCenter Orchestrator provides information about developing a Web services client for VMware® vCenter Orchestrator.

Orchestrator provides Web services APIs so that you can develop applications to access and use workflows through the Web. Orchestrator provides a representational state transfer (REST) API as well as a simple object access protocol (SOAP) service that you can use to perform various operations over workflows.

Intended Audience

This information is intended for Web application developers who want to access the Orchestrator processes across a network, through technologies such as SOAP and RESTful Web services.
Updated Information

*Developing a Web Services Client for VMware vCenter Orchestrator* is updated with each release of the product or when necessary.

This table provides the update history of *Developing a Web Services Client for VMware vCenter Orchestrator*.

<table>
<thead>
<tr>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN-000769-01</td>
<td>Added information about new SOAP API operations:</td>
</tr>
<tr>
<td></td>
<td>- “getWorkflowInputForId Operation,” on page 81</td>
</tr>
<tr>
<td></td>
<td>- “getWorkflowInputForWorkflowTokenId Operation,” on page 82</td>
</tr>
<tr>
<td></td>
<td>- “getWorkflowTokenBusinessState Operation,” on page 83</td>
</tr>
<tr>
<td>EN-000769-00</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
VMware vCenter Orchestrator provides Web services APIs so that you can develop applications to access workflows through the Web. The main purpose of the Orchestrator Web services APIs is to allow you to integrate Orchestrator workflows in custom Web-based applications.

Orchestrator provides Web services APIs that are based on two types of technologies:

- A representational state transfer (REST) API. The Orchestrator REST API exposes the objects in the Orchestrator inventory and the inventories of the installed plug-ins as resources that you can access at predefined URLs. HTTP requests at these URLs result in triggering operations over workflows. The Orchestrator REST API exposes inventory objects as resources through a set of RESTful Web services that you can use to retrieve the definitions of workflows, run workflows, check the status of the running workflows, cancel workflow runs, process waiting user interactions, retrieve the presentation of workflows, and so on.

- A simple object access protocol (SOAP) service. The Orchestrator SOAP service API provides a set of Web service definition language (WSDL) object type definitions and a set of Web service operations, that obtain workflows, run workflows, refresh workflow states, and obtain their output parameter values. You can also use the SOAP service 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.
Using the vCenter Orchestrator REST API

The Orchestrator REST API provides functionality that allows you to communicate with the Orchestrator server directly through HTTP and perform various workflow-related operations over workflows.

The Orchestrator REST API exposes the objects from the inventories of the Orchestrator server and the installed plug-ins as resources at predefined URLs. You make HTTP calls at these URLs to trigger operations in Orchestrator. In this way, you can perform various tasks over workflows:

- Run a workflow, schedule a workflow, retrieve the runs of a workflow, answer to a user interaction, and cancel a workflow run.
- Retrieve details about a workflow such as its input and output parameters and its presentation.
- Retrieve details about a workflow run, such as its state, generated logs, start date, and end date.
- Browse the inventories of Orchestrator and the installed plug-ins.
- Import and export workflows, actions, and packages.

By using the Orchestrator REST API you can easily integrate Orchestrator workflows in custom applications that you can build in any programming language.

The Orchestrator REST API also provides eTag support as well as a mechanism for caching of response data.

This chapter includes the following topics:

- “Authenticating Against Orchestrator and Third-Party Systems,” on page 14
- “Accessing the Reference Documentation for the Orchestrator REST API,” on page 16
- “Using the Java REST SDK,” on page 16
- “Operations with Workflows,” on page 17
- “Working with Tasks,” on page 30
- “Finding Objects in the Orchestrator Inventory,” on page 32
- “Importing and Exporting Orchestrator Objects,” on page 34
- “Deleting Orchestrator Objects,” on page 38
- “Setting Permissions on Orchestrator Objects,” on page 39
Authenticating Against Orchestrator and Third-Party Systems

You must authenticate against Orchestrator in the HTTP requests that you make through the Orchestrator REST API. If you use the Orchestrator REST API to access resources on a third-party system, such as vCenter Server, you must authenticate against that system as well.

For example, to access all workflows in the Orchestrator inventory, you must authenticate against Orchestrator. However, to run a workflow in vCenter Server, you must authenticate against Orchestrator and vCenter Server.

Depending on whether you configure Orchestrator with LDAP or with vCenter Single Sign On, the authentication scheme for the Orchestrator REST API is different. If Orchestrator uses LDAP, you must authenticate by using valid credentials. If Orchestrator uses vCenter Single Sign On, you must authenticate by using a holder-of-key token that the vCenter Single Sign On Server issued.

If you make HTTP requests at the top-level URL of the Orchestrator REST API, you do not need to authenticate against Orchestrator. The top level URL of the Orchestrator REST API is https://vcoHost:port/api/.

A GET request at the top level URL of the REST API returns URLs to all resources that are accessible through the API. To make HTTP requests at these URLs, you must authenticate against Orchestrator or the third-party system where the resources are located.

Using vCenter Single Sign On Authentication with the Orchestrator REST API

If Orchestrator is configured with the vCenter Single Sign On Server, you need a principal holder-of-key token to access system objects in Orchestrator through the vCO REST API. To access vCenter Server or third-party systems that use the vCenter Single Sign On Server through the Orchestrator server, you need a delegate holder-of-key token for Orchestrator and your principal token.

Accessing System Objects in Orchestrator

You can access system objects in Orchestrator at the URLs of the Inventory and the Catalog services of the REST API.

- https://vcoHost:port/api/inventory/System/
- https://vcoHost:port/api/catalog/System/

When you access system objects in Orchestrator, you pass your principal holder-of-key token in the Authorization header of HTTP requests that you make to the Inventory or the Catalog service.

For example, to retrieve all system objects of type workflow you make a GET request at https://vcoHost:port/api/catalog/System/Workflow/. To authenticate against Orchestrator, you need to pass your principal holder-of-key token in the Authorization header of the request.

Accessing Objects in Third-Party Systems

To perform operations in third-party systems that are registered with the vCenter Single Sign On Server through the Orchestrator REST API, you must authenticate against Orchestrator and the third-party system. You include two headers in the HTTP calls that you make through the Orchestrator REST API.

- Authorization. You must pass your principal holder-of-key token in this header.
- vCOAuthorization. You must pass a delegate holder-of-key token for Orchestrator in this header. You must acquire the delegate token for Orchestrator from the vCenter Single Sign On Server. Orchestrator uses the delegate token to authenticate against the third-party system on your behalf.
For example, to run a workflow that uses a virtual machine through the Orchestrator REST API, you access resources both in Orchestrator and in vCenter Server. To authenticate against Orchestrator and vCenter Server, you must pass your principal holder-of-key token in the Authorization header of the request that you make, and the delegate token in the VC0Authorization header. In this way, you authenticate against Orchestrator with your principal token and Orchestrator authenticates on your behalf against vCenter Server with the delegate token.

The vCenter Single Sign On Server treats Orchestrator as a solution, and every solution is registered with a unique user name with the vCenter Single Sign On Server. You request a delegate token for Orchestrator by passing the solution user name of Orchestrator and a principal holder-of-key token to the vCenter Single Sign On Server. The token that the vCenter Single Sign On Server issues is a delegate holder-of-key token for Orchestrator to authenticate on your behalf against third-party systems.

Get the Solution User Name of Orchestrator

The vCenter Single Sign On Server treats Orchestrator as a solution, and every solution is registered with a unique user name with the vCenter Single Sign On Server. To be able to request a delegate holder-of-key token for Orchestrator from the vCenter Single Sign On Server, you need the solution user name of Orchestrator.

Prerequisites

Verify that you have a valid principal holder-of-key token that the vCenter Single Sign On Server issued.

Procedure

1. Make a GET request at the URL of the solution user name of Orchestrator:

   ```
   GET https://{vcoHost}:{port}/api/users/
   ```

2. Provide your principal holder-of-key token in the Authorization header of the request.

   The `<user solution-user="vCOSolutionUserName"/>` element of the response contains the solution user name of Orchestrator. The following is an example of a solution user name of Orchestrator.

   ```
   <user xmlns="http://www.vmware.com/vco" solution-user="vCO-133acc26ff78e5695b102146326" admin-rights="true"/>
   ```

What to do next

Use the solution user name of Orchestrator and your principal holder-of-key token to request a delegate holder-of-key token from the vCenter Single Sign On Server.

Using LDAP Authentication with the Orchestrator REST API

You must apply the Basic HTTP Authentication scheme when Orchestrator is configured with LDAP, or if you use the Orchestrator server to access a third-party system that is configured with LDAP.

The Basic HTTP Authentication scheme allows you to authenticate against Orchestrator or a third-party system by including an Authorization header in the API calls that you make. You must provide a base64-encoded credentials in the Authorization header. Orchestrator uses the same credentials to authenticate on your behalf against third-party systems that are configured with LDAP.

For details about the Basic HTTP Authentication, see RFC 2617.
Accessing the Reference Documentation for the Orchestrator REST API

The reference documentation for the Orchestrator REST API contains information about the RESTful Web services of the API, the data model that is applicable for the API, the response codes that are valid for the API, code examples, and so on.

The reference documentation of the Orchestrator REST API is installed together with Orchestrator. The reference documentation is available at https://vcoHost:port/api/docs/.

Using the Java REST SDK

You can use a Java SDK library to call operations on the Orchestrator REST API in Java applications and work directly with objects.

Every RESTful Web service of the Orchestrator REST SDK has a wrapping Java class with methods that correspond to the operations that can be run by using the service.

The Java REST SDK is installed together with Orchestrator. The Java client library, reference documentation, and examples are available at https://vcoHost:port/api/docs/downloads.html.

Example: Run a Workflow and Wait for its Completion

The following example code runs a workflow and waits for it to complete.

```java
//start a new session to Orchestrator by using specified credentials
VcoSession session = DefaultVcoSessionFactory.newLdapSession(new URI("https://vco-server:8281/api/"), "username", "password");

//create the services
WorkflowService workflowService = new WorkflowService(session);
ExecutionService executionService = new ExecutionService(session);

//find a workflow by ID
Workflow workflow = workflowService.getWorkflow("1231235");

//create an ExecutionContext from the user's input

//run the workflow
WorkflowExecution execution = executionService.execute(workflow, context);

//wait for the workflow to reach the user interaction state, checking every 500 milliseconds
execution = executionService.awaitState(execution, 500, 10, WorkflowExecutionState.CANCELED, WorkflowExecutionState.FAILED, WorkflowExecutionState.COMPLETED);

String nameParamValue = new ParameterExtractor().fromTheOutputOf(execution).extractString("name");
System.out.println("workflow was executed with 'name' input set to" + nameParamValue);
```
Operations with Workflows

The Orchestrator REST API provides Web services that you can use to perform various operations with workflows.

Find a Workflow and Retrieve Its Definition

To be able to perform any kind of operation with a workflow, you must find that workflow in the Orchestrator inventory and retrieve its definition. The definition lists the workflow input and output parameters, and contains links to the available workflow runs, the workflow presentation, and other objects.

Prerequisites

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

Procedure

1. Find the inventory item of the workflow.
   - If you have the full name of the workflow or a key word from the name, make a GET request at the URL of the Workflow service by applying a filter:
     
     GET https://{vcoHost}:{port}/api/workflows?conditions=name={workflowFullName}
     
     GET https://{vcoHost}:{port}/api/workflows?conditions=name~{keyWord}
   - Search for the workflow through the Catalog or the Inventory service by making a GET request at the URL that is an entry point for the workflow inventory items:
     
     GET https://{vcoHost}:{port}/api/catalog/System/Workflow/
     
     GET https://{vcoHost}:{port}/api/inventory/System/Workflows/

2. Retrieve the inventory item of the workflow by making a GET request at its URL:

   GET https://{vcoHost}:{port}/api/catalog/System/Workflow/{workflowID}/

3. Retrieve the definition of the workflow by making a GET request at the URL of the definition:

   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/

Example: Search for the Send Hello Workflow

You can find the Send Hello workflow and retrieve its definition:

1. To find the Send Hello workflow, make a GET request at the URL of the Workflow service by applying a filter:

   GET https://localhost:8281/api/workflows?conditions=name=Hello

   You receive a list of the workflows that contain Hello in their names:

   <xml version="1.0" encoding="UTF-8" standalone="yes">
   <inventory-items xmlns="http://www.vmware.com/vco" total="2">
     <link rel="down"
       href="https://localhost:8281/api/catalog/System/Workflow/CF808080808080808080808080808080E6808080013086668236014d16e1/">
       <attributes>
         <attribute name="id" value="CF808080808080808080808080808080E6808080013086668236014d16e1"/>
   </xml>
2 Make a GET request at the URL of the inventory item of the Send Hello workflow:

GET https://localhost:8281/api/catalog/System/Workflow/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/

You receive the inventory item of the Send Hello workflow in the response body:

<xml version="1.0" encoding="UTF-8" standalone="yes">
    <relations>
      <link rel="down" href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/"/>
    </relations>
    <attributes>
      <attribute name="id" value="CF808080808080808080808080808080DA808080013086668236014a0614d16e1"/>
      <attribute name="canExecute" value="true"/>
      <attribute name="description" value=""/>
      <attribute name="name" value="Send Hello"/>
      <attribute name="type" value="Workflow"/>
      <attribute name="canEdit" value="true"/>
    </attributes>
  </inventory-item>
</xml>

3 To retrieve the workflow’s definition make a GET request at its URL:

GET https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/
You receive the definition of the Send Hello workflow in the response body:

```xml
<xml version="1.0" encoding="UTF-8" standalone="yes">
<workflow xmlns="http://www.vmware.com/vco" customized-icon="false"
href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/">
<relations>
  <link rel="up"
  href="https://localhost:8281/api/inventory/System/Workflows/Samples/HelloWorld/" />
  <link rel="add"
  href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/executions/" />
  <link rel="down"
  href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/executions/" />
  <link rel="down"
  href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/presentation/" />
  <link rel="down"
  href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/tasks/" />
  <link rel="down"
  href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/icon/" />
</relations>
<input-parameters>
  <parameter name="name" type="string" />
</input-parameters>
<output-parameters>
  <parameter name="message" type="string" />
</output-parameters>
<name>Send Hello</name>
<description></description>
</workflow>
```

Run a Workflow

You run a workflow through the Orchestrator REST API by creating a new execution object for a particular workflow.

**Prerequisites**

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

**Procedure**

1. Retrieve the definition of the workflow that you want to run by making a GET request at the URL of the definition:

   ```
   GET http://{vcoHost}:{port}/api/workflows/{workflowID}/
   ```
You receive the definition of the workflow in the response body of the request. In the workflow definition, you can view the input parameters of the workflow, the workflow description, and other information.

2 Make a POST request at the URL that holds the execution objects of the workflow:

```
POST https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/
```

3 Provide values for the input parameters of the workflow in an execution-context element in the request body.

If you provide an empty execution-context in the request body, the workflow runs with default values for its input parameters, if any.

If the POST request is successful, you receive the status code 202 with an empty response body and a link to the newly created execution object in the Location header.

**Example: Run the Send Hello Workflow**

You can retrieve the definition of the Send Hello workflow and run it.

1 Make a GET request at the URL that holds the definition of the Send Hello workflow:

```
GET https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/
```

You receive the workflow definition in the response body of the request:

```
<xml version="1.0" encoding="UTF-8" standalone="yes">
<workflow xmlns="http://www.vmware.com/vco" customized-icon="false" href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/">
<relations>
<link rel="up" href="https://localhost:8281/api/inventory/System/Workflows/Samples/HelloWorld/" />
<link rel="add" href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/executions/" />
<link rel="down" href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/presentation/" />
<link rel="down" href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/tasks/" />
<link rel="down" href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/icon/" />
</relations>
<input-parameters>

```
2 Make a POST request at the URL that holds the execution objects for the workflow:

```
POST https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/executions/
```

Pass values for the input parameters in an execution-context element in the request body:

```
<execution-context xmlns="http://www.vmware.com/vco">
  <parameters>
    <parameter name="name" type="string">
      <string>John Smith</string>
    </parameter>
  </parameters>
</execution-context>
```

Run a Workflow After Validating Its Input Parameters Against the Workflow Presentation

The presentation of a workflow can define constraints for the values that you can pass to the input parameters of the workflow, such as a predefined list of values or a certain range of values. To ensure that the workflow runs successfully, you must validate the values that you pass to the input parameters of the workflow against the definition of the workflow presentation.

When you integrate workflows in custom applications, you might need to create a wizard where you enter values for the input parameters of the workflow when you run it. By using the Workflow Presentation service, you can instantiate the presentation of a workflow and pass values for its input parameters in parts that correspond to the different screens of the wizard. You can validate the values that you pass to the input parameters against the constraints that are defined in the workflow presentation.

Prerequisites

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

Procedure

1 Retrieve the definition of the workflow that you want to run by making a GET request at the URL that contains the workflow definition:

```
GET https://{vcoHost}:{port}/api/workflows/{workflowID}/
```

You receive the definition of the workflow in the response body of the request. In the workflow definition, you can view the input parameters of the workflow, the workflow description and other information.

2 Retrieve the definition of the workflow presentation by making a GET request at its URL:

```
GET https://{vcoHost}:{port}/api/workflows/{workflowID}/presentation/
```

3 In the response body of the request, examine the definition of the workflow presentation for any constraints of the values that you can pass to the input parameters.

   For example, an input parameter can have a predefined list of values to choose from.

4 Instantiate the workflow presentation by making a POST request at the URL of the presentation instances:

```
POST https://{vcoHost}:{port}/api/workflows/{workflowID}/presentation/instances/
```
Provide an execution-context element in the request body to instantiate the presentation.

You can pass an empty execution-context or pass an execution-context with values only for some of the input parameters.

To pass values to the input parameters in parts, make as many POST or PUT requests as needed at the URL that holds the presentation instance:

PUT https://{vcoHost}:{port}/api/workflows/{workflowID}/presentation/instances/{executionID}/

Review the response body of the POST or PUT request that you made.

If the values that you passed to the input parameters are valid, you find a valid="true" attribute in the execution tag. If the presentation is valid, you can take the values that are listed in the out-parameters element of the response, and pass them as values to the input parameters when you run the workflow.

If the values for the input parameters are valid, run the workflow by making a POST request at the URL that holds the workflow executions:

POST https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/

Provide the valid values to the input parameters of the workflow in an execution-context element.

Example: Run the Send Hello Workflow by Validating Its Input Parameters

You can run the Send Hello workflow by validating its input parameters against the definitions of its presentation.

Make a GET request at the URL that holds the definition of the Send Hello workflow:

GET https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/

You receive the workflow definition in the response body of the request:

<xml version="1.0" encoding="UTF-8" standalone="yes">
<workflow xmlns="http://www.vmware.com/vco" customized-icon="false"
href="https://localhost:
8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/">
<relations>
<link rel="up" href="https://localhost:8281/api/inventory/System/Workflows/Samples/HelloWorld/" />
<link rel="add" href="https://localhost:
8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/executions/" />
<link rel="down" href="https://localhost:
8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/presentation/" />
<link rel="down" href="https://localhost:
8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/tasks/" />
</link rel="down" href="https://localhost:
8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/icon/"/>
2 Make a GET request at the URL that holds the definition of the workflow presentation:

GET https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/presentation/

3 Make a POST request at the URL that holds the execution instances of the workflow presentation:

POST https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/presentation/instances/

Provide an empty execution-context so that just to instantiate the presentation without providing any values for the input parameters:

<execution-context xmlns="http://www.vmware.com/vco"/>

The response body contains error messages attached to every field, indicating that the values for the input parameters are invalid.

......

<fields>
  <field type="string" hidden="false" id="name">
    <display-name>Name</display-name>
    <description>Name</description>
    <messages>
      <message severity="ERROR" code="VCO-CNS0002">
        <Summary>
          The minimum number of characters allowed for this field is 3.0
        </Summary>
      </message>
    </messages>
  </field>
  ......
In the response body of the request, you can check whether the values of the input parameters are valid:

```
<execution started-by="vcoadmin" valid="true"/>
```

5 If the presentation is valid, run the workflow by making a POST request at the URL that holds the workflow executions:

```
POST https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/executions/
```

In the request body, pass values to the input parameters of the workflow. Use the same values that are returned as output parameters of the workflow presentation, or directly use the request body of the last POST request that you made to the workflow presentation.

**Interacting with a Workflow While It Runs**

The Orchestrator REST API allows you to perform various operations with a workflow during its run. You can get the status of a running workflow, answer to a waiting user interaction, and cancel a workflow run.

**Get Workflow Run Objects and Check the Workflow Status**

You can get information about the runs of a workflow, such as the start and end dates, the state of the run, and the values for the input parameters. You can also get logs that are generated for a workflow run.

**Prerequisites**

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

**Procedure**

1. Retrieve the definition of the workflow whose status you want to check by making a GET request at the URL of the workflow:

   ```
   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/
   ```

   You receive the definition of the workflow in the response body of the request. The workflow definition contains a link to the execution instances of the workflow.

2. Retrieve the available execution instances of the workflow by making a GET request at their URL:

   ```
   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/
   ```

   The response body of the request lists the available execution instances of the workflow where you can view the start and end dates of every workflow run as well their status and initiator.

3. (Optional) To get more details about a particular run of the workflow, make a GET request at the URL of that run:

   ```
   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/{executionID}/
   ```

   In the response body of the request, you receive the XML representation of the particular workflow run. You can check the values of the input parameters that are passed for this run, the user who initiated the run, the start and end dates, as well as the state of the run.
4 (Optional) To retrieve the logs that are generated for the workflow run, make a GET request at the URL that holds the logs:

GET https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/{executionID}/logs/

5 (Optional) To retrieve additional information about the state of the run, make a GET request at the URL that holds the state of the workflow:

GET https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/{executionID}/state/

Example: Get the Runs of the Send Hello Workflow and Check the State of a Particular Run

If you have run the Send Hello workflow, you can get the available execution objects and check details about them.

1 Get the definition of the Send Hello workflow by making a GET request at the URL that holds the definition:

GET https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/

2 Get the available runs of the workflow by making a GET request at the URL that holds the execution objects for the workflow:

GET https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/executions/

3 From the response body of the request, select a workflow run and make a GET request to retrieve it:

GET http://localhost:8280/api/workflows/CF808080808080808080808080808080DA8080800132145338690643f66a027ec/

The response body contains the XML representation of the workflow run with the specified ID, where you can check details about that run:

....... 
<input-parameters>
    <parameter name="name" type="string">
        <string>John Smith</string>
    </parameter>
</input-parameters>

<output-parameters>
    <parameter name="message" type="string">
        <string>Hello, John Smith!</string>
    </parameter>
</output-parameters>

<start-date>2012-01-31T14:28:40.223+03:00</start-date>

<end-date>2012-01-31T14:28:40.410+03:00</end-date>

<started-by>vcoadmin</started-by>

......

Answer to a Waiting User Interaction

You can answer to a waiting user interaction of a workflow run through the Orchestator REST API.

Prerequisites

Verify that you have imported the sample workflows package in Orchestator. The package is included in the Orchestator sample applications ZIP file that you can download from the Orchestator documentation page.
Procedure

1. Retrieve the list of all user interaction objects by making a GET request at the URL that holds the available user interaction objects, or by filtering only the waiting user interactions:

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://vcoHost:port/api/catalog/System/UserInteraction">https://vcoHost:port/api/catalog/System/UserInteraction</a></td>
<td>Holds the available user interaction objects in Orchestrator</td>
</tr>
<tr>
<td><a href="https://vcoHost:port/api/catalog/System/UserInteraction?status=0">https://vcoHost:port/api/catalog/System/UserInteraction?status=0</a></td>
<td>Filters only the waiting user interaction objects.</td>
</tr>
</tbody>
</table>

You receive a list of the available user interaction objects. User interactions that are waiting have an attribute with name `state` and value `waiting`.

2. Make a GET request at the URL that holds the inventory item of the waiting user interaction to which you want to answer:

GET https://{vcoHost}:{port}/api/catalog/System/UserInteraction/{userInteractionID}/

The inventory item contains a link to the user interaction instance. The user interaction instance is associated with a particular workflow run.

3. Make a POST request at the URL of the user interaction instance for the particular workflow execution:

POST https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/{executionID}/interaction/

4. Provide values for the input parameters of the user interaction in an execution-context element in the request body.

The REST API returns a 204 status when you answer to a user interaction successfully.

Example: Answer to the User Interaction of the Interactive Hello World Workflow

You can run the Interactive Hello World sample workflow and answer to its user interaction.

1. Search for the waiting user interaction of the workflow by making GET request at the endpoint for the user interaction objects of the Catalog service:


2. Locate the user interaction inventory object for the Interactive Hello World workflow and make a GET request at its URL:

GET https://localhost:8281/api/catalog/System/UserInteraction/888080808080808080808080808080805A8080800132145338690643f66a027ec/

3. Make a POST request at the URL of the user interaction objects for the currently running workflow execution:

POST https://localhost:8281/api/workflows/CF808080808080808080808080808080E6808080013086668236014a0614d16e1/executions/88808080808080808080808080808080578080800132145338690643f66a027ec/interaction/
Provide a value for the input parameter in the request body:

```xml
<execution-context xmlns="http://www.vmware.com/vco">
    <parameters>
        <parameter name="name" type="string">
            <string>John Smith</string>
        </parameter>
    </parameters>
</execution-context>
```

**Answer to a User Interaction After Validating Input Parameters**

The presentation of a user interaction might define constraints for the values that you can pass to the input parameters of the workflow. When you answer to a user interaction, you can validate the values that you pass to the input parameters against the constraints that are defined in the presentation of the user interaction.

**Prerequisites**

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

**Procedure**

1. Retrieve the list of all user interaction objects by making a GET request at the URL that holds the available user interaction objects, or by filtering only the waiting user interactions:

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://vco">https://vco</a> host:port/api/catalog/System/UserInteraction</td>
<td>Holds the available user interaction objects in Orchestrator.</td>
</tr>
<tr>
<td><a href="https://vco">https://vco</a> host:port/api/catalog/System/UserInteraction?status=0</td>
<td>Filters only the waiting user interaction objects.</td>
</tr>
</tbody>
</table>

   You receive a list of the available user interaction objects. User interactions that are waiting have an attribute with name `state` and value `waiting`.

2. Make a GET request at the URL that holds the inventory item of the waiting user interaction that you want to answer:

   ```
   GET https://{vcoHost}:{port}/api/catalog/System/UserInteraction/{userInteractionID}/
   ```

   The response body contains a link to the user interaction instance. The user interaction instance is associated with a particular workflow run.

3. Make a GET request at the URL of the user interaction instance:

   ```
   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/{executionID}/interaction/
   ```

   In the response body, you find a down link to the presentation of the user interaction.

4. Make a GET request at the URL of the presentation of the user interaction:

   ```
   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/{executionID}/interaction/presentation/
   ```

   You receive the definition of the user interaction presentation in the response body.

5. In the presentation definition, check for constraints of the values that you can pass to the input parameters.
6 Run the user interaction presentation by making a POST request at the URL where the instances of the presentation reside:

   POST https://{vcoHost}:
   {port}/api/workflows/{workflowID}/executions/{executionID}/interaction/presentation/instances/

7 In the request body, provide values for the input parameters in an execution-context element.

   In the response body, you receive the instance of the user interaction presentation. If the values that you passed to the input parameters are valid, you find a valid="true" attribute in the execution element. In the out-parameters element, you find the valid values for the input parameters that you can use to answer to the user interaction.

8 Answer to the user interaction by making a POST request at the URL where the user interaction instance resides:

   POST https://{vcoHost}:
   {port}/api/workflows/{workflowID}/executions/{executionID}/interaction/

9 In the request body, pass an execution-context context with the values for the input parameters.

   You can use the same request body as the one for the POST request that you made at the URL for the user interaction presentation.

   If the last request is successful, you receive a status code 204 and an empty response body.

**Example: Answer to the User Interaction of the Interactive Hello World Workflow by Validating Input Parameters**

You can answer to the user interaction of the Interactive Hello World workflow by validating the values of the input parameters against the constraints that are defined in the presentation of the user interaction.

1 Search for the waiting user interactions of the workflow by making a GET request at the endpoint for the user interaction objects of the Catalog service:


2 Locate the user interaction inventory object for the Interactive Hello World workflow and make a GET request at its URL:

   GET https://localhost:8281/api/catalog/System/UserInteraction/888080808080808080808080808080805A8080800132145338690643f66a027ec/

3 Make a GET request at the URL of the user interaction instance:

   GET https://localhost:8281/api/catalog/System/UserInteraction/888080808080808080808080808080805A8080800132145338690643f66a027ec/interaction/

4 Make a GET request at the URL of the user interaction presentation:


   The presentation defines the input parameter as mandatory, and contains a constraint for the length of the string that you can pass.

5 Make a POST request at the URL that holds the instances of the user interaction presentation:

Provide a value for the input parameter in the request body:

```xml
<execution-context xmlns="http://www.vmware.com/vco">
  <parameters>
    <parameter name="name" type="string">
      <string>John Smith</string>
    </parameter>
  </parameters>
</execution-context>
```

The execution element of the response body contains a `valid="true"` attribute, indicating that the input parameter value is valid against the constraints in the user interaction presentation. The valid value is listed in the `output-parameters` element:

```xml
.............
<output-parameters>
  <parameter name="name" type="string">
    <string>John Smith</string>
  </parameter>
</output-parameters>
.............
```

6 Make a POST request at the URL of the user interaction instance by passing the same request body as in the POST request in step 5.

```
POST https://localhost:8281/api/catalog/System/UserInteraction/88888888888888888888888888888888A8808080132145338690643f66a027ec/interaction/
```

**Cancel a Workflow Run**

You can cancel the run of a workflow by using the Orchestrator REST API.

**Procedure**

1. Retrieve the definition of the workflow by making a GET request at the URL of the workflow's definition:

   ```
   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/
   ```

   The workflow definition contains a link to the available execution objects of the workflow.

2. Get the available workflow runs by making a GET request to the URL that holds the available execution objects for the workflow:

   ```
   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/
   ```

3. From the list of the available workflow executions, select the one that you want to cancel and make a DELETE request at its URL:

   ```
   DELETE https://{vcoHost}:{port}/api/workflows/{workflowID}/executions/{executionID}/
   ```
Working with Tasks

Using the Task service of the Orchestrator REST API, you can perform any operation that is related to managing tasks in Orchestrator. You can create a task for scheduling a workflow, modify the properties of an already existing task, delete a task, and so on.

Create a Task

You can create a task for scheduling a workflow by using the Orchestrator REST API.

Prerequisites

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

Procedure

1. Retrieve the definition of the workflow for which you want to create a task by making a GET request at the URL of the workflow:

   ```
   GET https://{vcoHost}:{port}/api/workflows/{workflowID}/
   ```

   In the workflow definition you can view the name and the ID of the workflow, as well as its input parameters.

2. To create a new task for the workflow, make a POST request at the URL of the Task service:

   ```
   POST https://{vcoHost}:{port}/api/tasks/
   ```

3. In the request body, provide the parameters for the new task in a task element.

   If the request is successful, the API responds with status code 202 and an empty response body.

Example: Create a Task for the Send Hello Workflow

You can create a task that schedules the Send Hello workflow to run on the fifteenth minute of every hour starting from a specific date.

1. Make a GET request at the URL of the Send Hello workflow to retrieve its definition:

   ```
   GET https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/
   ```

2. Make a POST request at the URL of the Task service by providing the parameters of the new task in the request body:

   ```
   POST https://localhost:8281/api/tasks/
   <task xmlns="http://www.vmware.com/vco">
     <name>Send Hello Task</name>
     <recurrence-cycle>every-hours</recurrence-cycle>
     <recurrence-start-date>2012-01-31T11:00:00+00:00</recurrence-start-date>
     <recurrence-end-date>2012-02-05T11:00:00+00:00</recurrence-end-date>
     <recurrence-pattern>15:15</recurrence-pattern>
     <input-parameters>
       <parameter name="name" type="string">
         <string>John Smith</string>
       </parameter>
     </input-parameters>
     <workflow href="https://localhost:8281/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/">
   ```
Modify a Task

You can change the properties of an existing task by using the Orchestrator REST API.

You can only add new scheduling properties to a task or change the values of the already existing properties.
If you want to replace the scheduling properties of a task, you must delete the task and create a new one.

Prerequisites

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

Procedure

1. Make a GET request at the URL of the task that you want to modify:
   
   ```
   GET https://{vcoHost}:{port}/api/tasks/{task ID}/
   ```

2. Check the properties of the task in the response body of the request.

3. To modify the task, make a POST request at the URL of the task by providing the new properties of the task in a task-data element in the request body.

If the POST request is successful, the API reruns a status code 200 and the updated task in the response body.

Example: Update the Send Hello Example Task

You can update the start and the end dates of a task. You can modify the example task that is introduced in “Create a Task,” on page 30. You must make a POST request at the URL of the task by providing the new start and end dates in the request body:

```xml
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<task-data xmlns="http://www.vmware.com/vco">
  <recurrence-start-date>2012-02-01T14:00:00+02:00</recurrence-start-date>
  <recurrence-end-date>2012-02-05T14:00:00+02:00</recurrence-end-date>
</task-data>
```

Check the State of a Task

You can check the state of the currently available tasks or check the state for all execution instances of a certain task.

Prerequisites

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

Procedure

- To check the status of all currently available tasks, make a GET request at the URL of the Task service:

  ```
  GET https://{vcoHost}:{port}/api/tasks/
  ```

  The response body contains the definitions of the currently available tasks in Orchestrator. The state of every task is available in an attribute element, whose name is state. Respectively, the value for the element can be finished, pending, running and so on.
To check the status of all executions of a certain task, make a GET request at the URL where the task executions reside:

```
GET https://{vcoHost}:{port}/api/tasks/{taskID}/executions/
```

You receive a list of the available executions for the task in the response body. The state of every execution is available in the `state` element of the task execution object.

### Finding Objects in the Orchestrator Inventory

You can find any object in the Orchestrator inventory by using the Catalog or the Inventory services. You can access only a certain subset of objects by applying filter parameters at the end of the URLs where you make HTTP requests.

You can use the Catalog service to find objects in the Orchestrator inventory that are of a certain type, or retrieve a specific object by its type and ID. For example, you can retrieve all objects that are of type `workflow` or `action`, or can retrieve a specific workflow or action.

The Inventory service allows you to browse the Orchestrator inventory by parent-child relations. Using the Inventory service, you can access objects that are available at a specific location in the Orchestrator inventory. For example, you can retrieve all workflows for Datacenter management by browsing to their location in the Orchestrator inventory, that is `Library/vCenter/Datacenter`.

Every service from the Orchestrator REST API supports filter parameters that you can add at the end of URLs when making HTTP requests. Using the filter parameters, you can narrow the results that you receive in the response body of a request at a specific URL.

### Find Objects by Type and ID

You can use the Catalog service of the REST API to find objects in Orchestrator by type and ID.

#### Prerequisites

Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

#### Procedure

1. Make a GET request at the URL of the Catalog Service:

   ```
   GET https://{vcoHost}:{port}/api/catalog/
   ```

   The response body of the request contains down links to the catalog entry points of the plug-ins that expose inventories in Orchestrator as well as to the system objects in Orchestrator:

   - `https://{vcoHost}:{port}/api/catalog/{plug-in namespace}/`
   - `https://{vcoHost}:{port}/api/catalog/System/`

2. To access objects that a plug-in exposes or the system objects in Orchestrator, make a GET request at the URL of the catalog entry point for the plug-in or at the URL where the system objects in Orchestrator reside.

   The response body of the request contains links to the types of objects that are exposed.

3. Make GET request at the URL of the type of object that you want to access.

   ```
   GET https://{vcoHost}:{port}/api/catalog/{namespace}/{objectType}/
   ```

4. Make a GET request at the URL of the specific object that you want to find:

   ```
   GET https://{vcoHost}:{port}/api/catalog/{namespace}/{objectType}/{objectId}/
   ```
Example: Find the Send Hello Workflow
You can find the sample Send Hello workflow by using the Catalog Service.
1. Make a GET request at the URL of the Catalog Service:
   GET https://localhost:8281/api/catalog/
2. Make a GET request at the URL where all system objects in Orchestrator are located:
   GET https://localhost:8281/api/catalog/System/
3. Make GET request at the URL where all workflows reside:
   GET https://localhost:8281/api/catalog/Workflow/
4. Make GET request at the URL of the Send Hello workflow:
   GET https://localhost:8281/api/catalog/Workflow/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/

Find Objects by Relations
You can use the Inventory service of the Orchestrator REST to browse the Orchestrator and the plug-in inventories as a hierarchy.

Prerequisites
Verify that you have imported the sample workflows package in Orchestrator. The package is included in the Orchestrator sample applications ZIP file that you can download from the Orchestrator documentation page.

Procedure
1. Make a GET request at the URL of the Inventory service:
   GET https://{vcoHost}:{port}/api/inventory/
   The response body contains down links to the registered inventories of the installed plug-ins as well as to the system objects in Orchestrator under System.
2. Make a GET request at the down link of the inventory that you want to access.
3. Make GET requests at the up and down links for the items in the inventory until you reach the object that you want to find.

Example: Find the Send Hello Workflow
You can browse the Orchestrator Inventory to find the Send Hello workflow.
1. Make a GET request at the URL of the Inventory service:
   GET https://localhost:8281/api/inventory/
2. Make a GET request at the URL where the system objects in Orchestrator reside:
   GET https://localhost:8281/api/inventory/System/
3. Make GET request at the URL where all workflows in Orchestrator reside:
   GET https://localhost:8281/api/inventory/System/Workflows/
4. Make a GET request at the URL of the Samples workflow category:
   GET https://localhost:8281/api/inventory/System/Workflows/Samples/
5. Use the down link for the Hello World workflow category where to locate the Send Hello workflow.
Apply Filters

The services of the Orchestrator REST API support additional URL parameters that allow you to narrow the objects that HTTP requests to the API return.

Different query parameters are supported for every URL to a resource that you can access through the REST API. To learn which query parameters are applicable to a URL, see the vCenter Orchestrator REST API reference documentation.

Procedure

To narrow the results from a request at a certain URL, apply filters at the end of the URL:

URL?filter_1&filter_2&filter_3&....&filter_N. Every filter contains query parameters that are valid for the relevant URL. For information about the valid query parameters for every URL, see the Orchestrator REST API reference documentation.

Example: Filter Workflows

If you look for workflows that contain a specific word in their name, for example datastore, you can apply the following filter in a request to the Catalog Service:

GET https://localhost:8281/api/catalog/System/Workflow?conditions=name~datastore

To limit the amount of the workflows that are returned to a certain number, for example five, apply an additional filter to the request:

GET https://localhost:8281/api/catalog/System/Workflow?conditions=name~datastore&maxResult=5

Importing and Exporting Orchestrator Objects

The Orchestrator REST API provides Web services that you can use to import and export workflows, actions, packages, resources, and configuration elements.

Import a Workflow

You can import a workflow through the Orchestrator REST API. Depending on the library of your REST client application, you can use custom code that defines the properties of the workflow.

Prerequisites

The workflow binary content should be available as multi-part content. For details, see RFC-2387.

Procedure

1. In a REST client application, add request headers to define the properties of the workflow that you want to import.

2. Make a POST request at the URL of the workflow objects:

   POST http://{vcoHost}:{port}/api/workflows/

If the POST request is successful, you receive the status code 202.
Export a Workflow

You can export a workflow through the Orchestrator REST API and download the workflow as a file.

Procedure

1. In a REST client application, add a request header with the following values.
   - Name: accept
   - Value: application/zip

2. Make a GET request at the URL of the workflow that you want to export:

   GET http://{vcoHost}:{port}/api/workflows/{workflowID}/

If the GET request is successful, you receive the status code 200. The workflow binary content is available as an attachment with a default file name workflow_name.workflow. You can save the file with a REST client application.

Import an Action

You can import an action through the Orchestrator REST API.

Depending on the library of your REST client application, you can use custom code that defines the properties of the action.

Prerequisites

The action binary content should be available as multi-part content. For details, see RFC-2387.

Procedure

1. In a REST client application, add request headers to define the properties of the action that you want to import.

2. Make a POST request at the URL of the action objects:

   POST http://{vcoHost}:{port}/api/actions/

If the POST request is successful, you receive the status code 202.

Export an Action

You can export an action through the Orchestrator REST API and download the action as a file.

Procedure

1. In a REST client application, add a request header with the following values.
   - Name: accept
   - Value: application/zip

2. Make a GET request at the URL of the action that you want to export:

   GET http://{vcoHost}:{port}/api/actions/{actionID}/

If the GET request is successful, you receive the status code 200. The action binary content is available as an attachment with a default file name action_name.action. You can save the file with a REST client application.
**Import a Package**

You can import a package through the Orchestrator REST API.

Depending on the library of your REST client application, you can use custom code that defines the properties of the package.

**Prerequisites**

The package binary content should be available as multi-part content. For details, see RFC-2387.

**Procedure**

1. In a REST client application, add request headers to define the properties of the package that you want to import.

2. Make a POST request at the URL of the package objects:
   
   ```
   POST http://{vcoHost}:{port}/api/packages/
   ```

   If the POST request is successful, you receive the status code 202.

**Export a Package**

You can export a package through the Orchestrator REST API and download the package as a file.

**Procedure**

1. In a REST client application, add a request header with the following values.

   - **Name**: accept
   - **Value**: application/zip

2. Make a GET request at the URL of the package that you want to export:

   ```
   GET http://{vcoHost}:{port}/api/packages/{package_name}/
   ```

   If the GET request is successful, you receive the status code 200. The package binary content is available as an attachment with a default file name `{package_name}.package`. You can save the file with a REST client application.

**Import a Resource**

You can import a resource through the Orchestrator REST API.

Depending on the library of your REST client application, you can use custom code that defines the properties of the resource.

**Prerequisites**

The resource binary content should be available as multi-part content. For details, see RFC-2387.

**Procedure**

1. In a REST client application, add request headers to define the properties of the resource that you want to import.

2. Make a POST request at the URL of the resource objects:

   ```
   POST http://{vcoHost}:{port}/api/resources/
   ```

   If the POST request is successful, you receive the status code 202.
Export a Resource

You can export a resource through the Orchestrator REST API.

Procedure

1. In a REST client application, add a request header with the following values.
   - Name: accept
   - Value: application/octet-stream
2. Make a GET request at the URL of the resource that you want to export:
   
   ```
   GET http://{vcoHost}:{port}/api/resources/{resourceID}/
   ```

   If the GET request is successful, you receive the status code 200. The content of the resource is available in the response body.

Import a Configuration Element

You can import a configuration element through the Orchestrator REST API.

Depending on the library of your REST client application, you can use custom code that defines the properties of the configuration element.

Prerequisites

The configuration element binary content should be available as multi-part content. For details, see RFC-2387.

Procedure

1. In a REST client application, add request headers to define the properties of the configuration element that you want to import.
2. Make a POST request at the URL of the configuration element objects:
   
   ```
   POST http://{vcoHost}:{port}/api/configurations/
   ```

   If the POST request is successful, you receive the status code 202.

Export a Configuration Element

You can export a configuration element through the Orchestrator REST API.

Procedure

1. In a REST client application, add a request header with the following values.
   - Name: accept
   - Value: application/vcoobject+xml
2. Make a GET request at the URL of the configuration element that you want to export:
   
   ```
   GET http://{vcoHost}:{port}/api/configurations/{configuration_elementID}/
   ```

   If the GET request is successful, you receive the status code 200. The configuration element content is available in the response body.
Deleting Orchestrator Objects

The Orchestrator REST API provides Web services that you can use to delete workflows, actions, packages, resources, and configuration elements.

Delete a Workflow

You can delete a workflow by using the Orchestrator REST API.

Procedure

1. Make a GET request and retrieve the ID of the workflow from the list of returned workflows:
   
   GET http://{vcoHost}:{port}/api/workflows/

2. Make a DELETE request at the URL of the workflow:
   
   DELETE http://{vcoHost}:{port}/api/workflows/{workflowID}/

If the DELETE request is successful, you receive the status code 200, and the response body is empty.

Delete an Action

You can delete an action by using the Orchestrator REST API.

Procedure

1. Make a GET request and retrieve the ID of the action from the list of returned actions:
   
   GET http://{vcoHost}:{port}/api/actions/

2. Make a DELETE request at the URL of the action:
   
   DELETE http://{vcoHost}:{port}/api/actions/{actionID}/

If the DELETE request is successful, you receive the status code 200, and the response body is empty.

Delete a Package

You can delete a package by using the Orchestrator REST API.

When you delete a package, the elements from the package are not deleted. If you want to delete the content of a package, you must provide an option parameter.

Procedure

1. Make a GET request and retrieve the name of the package from the list of returned packages:
   
   GET http://{vcoHost}:{port}/api/packages/
Make a DELETE request at the URL of the package, and if you want to delete elements from the package, provide an option parameter at the end of the request:

```
DELETE http://{vcoHost}:{port}/api/packages/{package_name}/?option={parameter}
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deletePackage</td>
<td>Only the package is deleted, while its content is retained.</td>
</tr>
<tr>
<td>deletePackageWithContent</td>
<td>The package and all its content is deleted. If other packages share elements with the deleted package, the shared elements are deleted from the other packages.</td>
</tr>
<tr>
<td>deletePackageKeepingShared</td>
<td>The package and the content that is not shared is deleted. Elements that are shared with other packages are not deleted.</td>
</tr>
</tbody>
</table>

If you do not provide an option parameter, the default deletePackage parameter is used.

If the DELETE request is successful, you receive the status code 200, and the response body is empty.

**Delete a Resource**

You can delete a resource by using the Orchestrator REST API.

**Procedure**

1. Make a GET request and retrieve the ID of the resource from the list of returned resources:
   
   ```
   GET http://{vcoHost}:{port}/api/resources/
   ```

2. Make a DELETE request at the URL of the resource:
   
   ```
   DELETE http://{vcoHost}:{port}/api/resources/{resourceID}/
   ```

If the DELETE request is successful, you receive the status code 200, and the response body is empty.

**Delete a Configuration Element**

You can delete a configuration element by using the Orchestrator REST API.

**Procedure**

1. Make a GET request and retrieve the ID of the configuration element from the list of returned configuration elements:
   
   ```
   GET http://{vcoHost}:{port}/api/configurations/
   ```

2. Make a DELETE request at the URL of the configuration element:
   
   ```
   DELETE http://{vcoHost}:{port}/api/configurations/{configuration_elementID}/
   ```

If the DELETE request is successful, you receive the status code 200, and the response body is empty.

**Setting Permissions on Orchestrator Objects**

You can set custom permissions for an Orchestrator object by using the REST API. To set the permissions, you must make a POST request at the URL of the object's permissions and define the permissions in the request body.

You can also use the Orchestrator REST API to retrieve information about an object's permissions or delete the existing permissions.
REST API Permissions

When you set permissions by using the Orchestrator REST API, you must use a set of characters to define the permissions.

You can define the permissions for an element by including a sequence of characters in the <rights> tag of the request body of a POST request.

The characters that you can use to set permissions through the Orchestrator REST API have specific meanings.

Table 2-1. Orchestrator REST API Permissions Character Set

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Gives view permissions.</td>
</tr>
<tr>
<td>x</td>
<td>Gives execute permissions.</td>
</tr>
<tr>
<td>i</td>
<td>Gives inspect permissions.</td>
</tr>
<tr>
<td>c</td>
<td>Gives edit permissions.</td>
</tr>
<tr>
<td>a</td>
<td>Gives administrative permissions.</td>
</tr>
</tbody>
</table>

Example: Syntax for Setting Permissions

You can use the following example syntax in the request body of a POST request at the URL of an Orchestrator element's permissions.

```xml
<permissions xmlns="http://www.vmware.com/vco">
  <permission>
    <principal>cn=vcousers,ou=vco,dc=appliance</principal>
    <rights>ric</rights>
  </permission>
</permissions>
```

By setting ric permissions in the <rights> tag of the request body, you allow members of the vcousers user group to view, inspect, and edit the Orchestrator element.

Retrieve the Permissions of a Workflow

You can retrieve information about the permissions of a workflow by using the Orchestrator REST API.

Procedure

1. Make a GET request and retrieve the ID of the workflow from the list of returned workflows:
   
   ```
   GET http://{vcoHost}:{port}/api/workflows/
   ```

2. Make a GET request at the URL of the workflow's permissions:
   
   ```
   GET http://{vcoHost}:{port}/api/workflows/{workflowID}/permissions/
   ```

If the GET request is successful, you receive the status code 200. Information about the workflow's permissions is available in the response body.
Delete the Permissions of a Workflow

You can delete the permissions of a workflow by using the Orchestrator REST API. You can delete the existing permissions of a workflow before you set new permissions.

Procedure

1. Make a GET request and retrieve the ID of the workflow from the list of returned workflows:
   
   ```
   GET http://{vcoHost}:{port}/api/workflows/
   ```

2. Make a DELETE request at the URL of the workflow’s permissions:
   
   ```
   DELETE http://{vcoHost}:{port}/api/workflows/{workflowID}/permissions/
   ```

   If the DELETE request is successful, you receive the status code 204, and the response body is empty.

Set the Permissions for a Workflow

You can set the permissions for a workflow by using the Orchestrator REST API.

Prerequisites

Review the types of permissions that you can set and the syntax that you can use in the request body. See “REST API Permissions,” on page 40.

Procedure

1. Make a GET request and retrieve the ID of the workflow from the list of returned workflows:
   
   ```
   GET http://{vcoHost}:{port}/api/workflows/
   ```

2. In a REST client application, add request headers to define the properties of the workflow for which you want to set permissions.

3. In the request body, specify the permissions that you want to set.

4. Make a POST request at the URL of the workflow’s permissions:
   
   ```
   POST http://{vcoHost}:{port}/api/workflows/{workflowID}/permissions/
   ```

   If the POST request is successful, you receive the status code 201. Information about the workflow’s permissions is available in the response body.

Retrieve the Permissions of an Action

You can retrieve information about the permissions of an action by using the Orchestrator REST API.

Procedure

1. Make a GET request and retrieve the ID of the action from the list of returned actions:
   
   ```
   GET http://{vcoHost}:{port}/api/actions/
   ```

2. Make a GET request at the URL of the action’s permissions:
   
   ```
   GET http://{vcoHost}:{port}/api/actions/{actionID}/permissions/
   ```

   If the GET request is successful, you receive the status code 200. Information about the action’s permissions is available in the response body.
Delete the Permissions of an Action

You can delete the permissions of an action by using the Orchestrator REST API. You can delete the existing permissions of an action before you set new permissions.

Procedure

1. Make a GET request and retrieve the ID of the action from the list of returned actions:
   
   ```
   GET http://{vcoHost}:{port}/api/actions/
   ```

2. Make a DELETE request at the URL of the action’s permissions:
   
   ```
   DELETE http://{vcoHost}:{port}/api/actions/{actionID}/permissions/
   ```

   If the DELETE request is successful, you receive the status code 204, and the response body is empty.

Set the Permissions for an Action

You can set the permissions for an action by using the Orchestrator REST API.

Prerequisites

Review the types of permissions that you can set and the syntax that you can use in the request body. See “REST API Permissions,” on page 40.

Procedure

1. Make a GET request and retrieve the ID of the action from the list of returned actions:
   
   ```
   GET http://{vcoHost}:{port}/api/actions/
   ```

2. In a REST client application, add request headers to define the properties of the action for which you want to set permissions.

3. In the request body, specify the permissions that you want to set.

4. Make a POST request at the URL of the action’s permissions:
   
   ```
   POST http://{vcoHost}:{port}/api/actions/{actionID}/permissions/
   ```

   If the POST request is successful, you receive the status code 201. Information about the action’s permissions is available in the response body.

Retrieve the Permissions of a Package

You can retrieve information about the permissions of a package by using the Orchestrator REST API.

Procedure

1. Make a GET request and retrieve the name of the package from the list of returned packages:
   
   ```
   GET http://{vcoHost}:{port}/api/packages/
   ```

2. Make a GET request at the URL of the package’s permissions:
   
   ```
   GET http://{vcoHost}:{port}/api/packages/{package_name}/permissions/
   ```

   If the GET request is successful, you receive the status code 200. Information about the package’s permissions is available in the response body.
Delete the Permissions of a Package

You can delete the permissions of a package by using the Orchestrator REST API. You can delete the existing permissions of a package before you set new permissions.

Procedure

1. Make a GET request and retrieve the name of the package from the list of returned packages:
   
   ```
   GET http://{vcoHost}:{port}/api/packages/
   ```

2. Make a DELETE request at the URL of the package’s permissions:
   
   ```
   DELETE http://{vcoHost}:{port}/api/packages/{package_name}/permissions/
   ```

   If the DELETE request is successful, you receive the status code 204, and the response body is empty.

Set the Permissions for a Package

You can set the permissions for a package by using the Orchestrator REST API.

Prerequisites

Review the types of permissions that you can set and the syntax that you can use in the request body. See “REST API Permissions,” on page 40.

Procedure

1. Make a GET request and retrieve the name of the package from the list of returned packages:
   
   ```
   GET http://{vcoHost}:{port}/api/packages/
   ```

2. In a REST client application, add request headers to define the properties of the package for which you want to set permissions.

3. In the request body, specify the permissions that you want to set.

4. Make a POST request at the URL of the package’s permissions:
   
   ```
   POST http://{vcoHost}:{port}/api/packages/{package_name}/permissions/
   ```

   If the POST request is successful, you receive the status code 201. Information about the package’s permissions is available in the response body.

Retrieve the Permissions of a Resource

You can retrieve information about the permissions of a resource by using the Orchestrator REST API.

Procedure

1. Make a GET request and retrieve the ID of the resource from the list of returned resources:
   
   ```
   GET http://{vcoHost}:{port}/api/resources/
   ```

2. Make a GET request at the URL of the resource’s permissions:
   
   ```
   GET http://{vcoHost}:{port}/api/resources/{resourceID}/permissions/
   ```

   If the GET request is successful, you receive the status code 200. Information about the resource’s permissions is available in the response body.
Delete the Permissions of a Resource

You can delete the permissions of a resource by using the Orchestrator REST API. You can delete the existing permissions of a resource before you set new permissions.

Procedure

1. Make a GET request and retrieve the ID of the resource from the list of returned resources:
   
   GET http://{vcoHost}:{port}/api/resources/

2. Make a DELETE request at the URL of the resource’s permissions:
   
   DELETE http://{vcoHost}:{port}/api/resources/{resourceID}/permissions/

   If the DELETE request is successful, you receive the status code 204, and the response body is empty.

Set the Permissions for a Resource

You can set the permissions for a resource by using the Orchestrator REST API.

Prerequisites

Review the types of permissions that you can set and the syntax that you can use in the request body. See “REST API Permissions,” on page 40.

Procedure

1. Make a GET request and retrieve the ID of the resource from the list of returned resources:
   
   GET http://{vcoHost}:{port}/api/resources/

2. In a REST client application, add request headers to define the properties of the resource for which you want to set permissions.

3. In the request body, specify the permissions that you want to set.

4. Make a POST request at the URL of the resource’s permissions:
   
   POST http://{vcoHost}:{port}/api/resources/{resourceID}/permissions/

   If the POST request is successful, you receive the status code 201. Information about the resource’s permissions is available in the response body.

Retrieve the Permissions of a Configuration Element

You can retrieve information about the permissions of a configuration element by using the Orchestrator REST API.

Procedure

1. Make a GET request and retrieve the ID of the configuration element from the list of returned configuration elements:
   
   GET http://{vcoHost}:{port}/api/configurations/

2. Make a GET request at the URL of the configuration element’s permissions:
   
   GET http://{vcoHost}:{port}/api/configurations/{configuration_elementID}/permissions/

   If the GET request is successful, you receive the status code 200. Information about the configuration element’s permissions is available in the response body.
Delete the Permissions of a Configuration Element

You can delete the permissions of a configuration element by using the Orchestrator REST API. You can delete the existing permissions of a configuration element before you set new permissions.

Procedure

1. Make a GET request and retrieve the ID of the configuration element from the list of returned configuration elements:

   GET http://{vcoHost}:{port}/api/configurations/

2. Make a DELETE request at the URL of the configuration element's permissions:

   DELETE http://{vcoHost}:{port}/api/configurations/{configuration_elementID}/permissions/

If the DELETE request is successful, you receive the status code 204, and the response body is empty.

Set the Permissions for a Configuration Element

You can set the permissions for a configuration element by using the Orchestrator REST API.

Prerequisites

Review the types of permissions that you can set and the syntax that you can use in the request body. See “REST API Permissions,” on page 40.

Procedure

1. Make a GET request and retrieve the ID of the configuration element from the list of returned configuration elements:

   GET http://{vcoHost}:{port}/api/configurations/

2. In a REST client application, add request headers to define the properties of the configuration element for which you want to set permissions.

3. In the request body, specify the permissions that you want to set.

4. Make a POST request at the URL of the configuration element's permissions:

   POST http://{vcoHost}:{port}/api/configurations/{configuration_elementID}/permissions/

If the POST request is successful, you receive the status code 201. Information about the configuration element's permissions is available in the response body.
Writing a Client Application for the Orchestrator SOAP Service

Most applications that use the Orchestrator SOAP service have a common structure. To create a client application for the Orchestrator SOAP service, you must perform a standard sequence of tasks.

This chapter includes the following topics:

- “Process for Creating an Orchestrator Web Service Client Application,” on page 47
- “Web Service Endpoint,” on page 49
- “Generating the Orchestrator Web Service Stubs,” on page 49
- “Accessing the Server from Web Service Clients,” on page 50
- “Create a Web Service Client,” on page 50
- “Time Zones and Running Workflows Through Web Services,” on page 61
- “Web Service Application Examples,” on page 61

Process for Creating an Orchestrator Web Service Client Application

Developing a Web services client application follows a broad sequence of stages.

The following figure shows how to create a typical Orchestrator Web service client application.
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 or host name 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 client and server stubs from the Orchestrator WSDL.

Orchestrator publishes the WSDL 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.

NOTE The exact objects that the Orchestrator Web service generates depend on your code generator. The objects in the following list are those that the Axis 1.4 code generator generates. Other code generators might generate the objects differently. If the generator that you use generates different objects, use VSOWebControlService as the point of access to the other Web service objects.

<table>
<thead>
<tr>
<th>Table 3-1. Java classes generated with Axis 1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>VSOWebControl</td>
</tr>
<tr>
<td>WebServiceStub</td>
</tr>
<tr>
<td>VSOWebControlProxy</td>
</tr>
<tr>
<td>VSOWebControlService</td>
</tr>
<tr>
<td>VSOWebControlServiceLocator</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Accessing the Server from Web Service Clients

By default, Orchestrator permits access to workflows from Web service clients. However, the Orchestrator administrator can configure the server to deny connections from Web service clients.

If the Orchestrator administrator has disabled access to the server from Web service clients, the server only answers Web service client calls from the echo() and echoWorkflow() methods, for testing purposes.

The Orchestrator administrator enables and disables access to the server from Web service clients by setting a system property. For information about setting system properties, see Installing and Configuring VMware vCenter Orchestrator.

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 stub from the Orchestrator WSDL definition by using a code generator.

Procedure

1. **Connect to the Orchestrator Web Service** on page 51
   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 52
   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 52
   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 53
   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 54
   You can use the findRelation operation to locate the children of a particular object.

6. **Find Workflows in the Orchestrator Server** on page 55
   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 56
   The getAllWorkflows operation lists all workflows that a user can access as an array of Workflow objects.

8. **Retrieve the ID of a Workflow** on page 56
   Every workflow has a unique ID that you can retrieve by using the Orchestrator client and a text editor. You need the workflow ID to perform operations over a workflow by using the Orchestrator SOAP API.

9. **Find Workflows by Using the getWorkflowsWithName Operation** on page 56
   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.
**Find Workflows by Using the getWorkflowForID Operation** on page 57

If you know a particular workflow ID, a Web service application can obtain this workflow by using the getWorkflowForID operation.

**Run Workflows from a Web Service Client** on page 57

The main purpose of a Web services client is to run workflows across a network.

**Interact with a Workflow While it Runs** on page 58

After the workflow starts, the Web services client can perform various actions in response to events while the workflow is running.

**Obtain Workflow Results** on page 60

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 VSOWebControl interface.

**Procedure**

1. In your Web service client application class, create a VSOWebControl 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 vsoWebControl = new VSOWebControlServiceLocator().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 VSOWebControlServiceLocator().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 plug-in.

   ```java
   QueryResult queryResult = vsoWebControl.find("VC:VirtualMachine", null, <username>, <password>);
   if (queryResult != null) {
       System.out.println("Found " + queryResult.getTotalCount() + " objs.");
       FinderResult[] elts = queryResult.getElements();
       finderResult = elts[0];
       displayFinderResult(finderResult);
   }
   ```
According to the query syntax defined by the vCenter Server 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 set arguments when starting or answering workflows. 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 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 set arguments when starting or answering workflows. 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 run workflows upon.

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 `vcenter/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 ", + 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 set arguments when starting or answering workflows. 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`
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.

```
Workflow[] workflows = vsocWebControl.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.

Retrieve the ID of a Workflow

Every workflow has a unique ID that you can retrieve by using the Orchestrator client and a text editor. You need the workflow ID to perform operations over a workflow by using the Orchestrator SOAP API.

Procedure

1. In the Orchestrator client, select the Workflows view.
2. From the workflow library, select the workflow whose ID you want to retrieve and press Ctrl+C.
3. Open a text editor and press Ctrl+V.

The workflow name and ID appear in the text editor.

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**

- Create a `Workflow` object by calling the `getWorkflowForID` operation.

  ```java
  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.

   ```java
   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.

```java
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.

```java
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 a WorkflowToken by calling `getWorkflowTokenStatus()`.

   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) Print the workflow results.

WorkflowTokenAttribute resultCode = retAttributes[0];
WorkflowTokenAttribute resultMessage = retAttributes[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.
   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 Documentation landing page.
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.

This chapter includes the following topics:

- “FinderResult Object,” on page 63
- “ModuleInfo Object,” on page 64
- “Property Object,” on page 65
- “QueryResult Object,” on page 65
- “Workflow Object,” on page 66
- “WorkflowParameter Object,” on page 67
- “WorkflowToken Object,” on page 67
- “WorkflowTokenAttribute Object,” on page 70

**FinderResult Object**

A FinderResult represents an object from the Orchestrator inventory that Orchestrator locates in an external application by using 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 any type of object that an Orchestrator plug-in defines. 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 properties</td>
<td>properties</td>
<td>A list of the discovered object's properties.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The format of the properties values is defined by each plug-in in its vsoc.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/Custom SDKObject?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 display 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 plug-ins or plug-in versions.

The getAllPlugins operation returns arrays of ModuleInfo objects to list all the plug-ins a user can access, as the following example shows.

```java
public ModuleInfo[] getAllPlugins(username, 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>Name of the plug-in, used as a prefix in object names.</td>
</tr>
<tr>
<td>String</td>
<td>moduleVersion</td>
<td>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>Plug-in name shown in the Orchestrator inventory.</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.

```java
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. 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.

```java
public QueryResult find(String type, String query, String username, String password);
```

The following table shows the properties of the QueryResult object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long</td>
<td>totalCount</td>
<td>The total number of objects found. The QueryResult object contains an array of FinderResult objects. The vso.xml file for the relevant plug-in sets the number of FinderResult objects the query returns. The standard plug-ins that Orchestrator provides all return an unlimited number of FinderResult objects. The totalCount property reports the total number of FinderResult objects found. If the value of totalCount is greater than the number set by the plug-in, the array of FinderResult returned does not include all the objects found in the queried inventory.</td>
</tr>
<tr>
<td>FinderResult[]</td>
<td>elements</td>
<td>An array of FinderResult objects.</td>
</tr>
</tbody>
</table>
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 they have the permission to see.

Orchestrator provides the following operations to obtain Workflow objects.

```java
public Workflow[] getWorkflowsWithName(String workflowName, 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>
</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. 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>
</tbody>
</table>
| String    | currentItemState | The state of the current step in the workflow, with the following possible values:   
  - running: the step is running   
  - waiting: the step is waiting for runtime parameters, which can be provided by answerWorkflowInput   
  - waiting-signal: the step is waiting for an external event from a plug-in   
  - canceled: the step was canceled by a user or API-integrated program   
  - completed: the step has finished   
  - failed: the step encountered an error   
You must run getWorkflowTokenForId every time you update this value.   
Note: VMware recommends that you do not use currentItemState. The globalState property makes currentItemState redundant. |
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| String   | `globalState` | The state of the workflow as a whole, with the following possible values:  
  - 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 `globalState` is the state of the workflow as a whole. You must run `getWorkflowTokenForId` every time you update this value. |
| String   | `startDate` | The date and time that this workflow token started  
  The `startDate` value is set at the moment the workflow starts. When you obtain a token, its `startDate` has already been initialized. |
### Type | Value | Description
--- | --- | ---
String | endDate | 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 reaches the end of its run. The `endDate` is only set when the workflow finishes in one of the completed, failed or canceled states.

String | xmlContent | 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 `<exception>` tags, as the following example shows.

```xml
<token>
  <atts>
    <stack>
      <att n='attstr' t='string' e='n'>
        <! [CDATA[attribute]]>Attribute value</att>
      </stack>
    </atts>
  </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>'
```
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.

This chapter includes the following topics:

- “answerWorkflowInput Operation,” on page 74
- “cancelWorkflow Operation,” on page 74
- “echo Operation,” on page 75
- “echoWorkflow Operation,” on page 75
- “executeWorkflow Operation,” on page 75
- “find Operation,” on page 76
- “findForId Operation,” on page 77
- “findRelation Operation,” on page 78
- “getAllPlugins Operation,” on page 80
- “getAllWorkflows Operation,” on page 80
- “getWorkflowForId Operation,” on page 81
- “getWorkflowInputForId Operation,” on page 81
- “getWorkflowInputForWorkflowTokenId Operation,” on page 82
- “getWorkflowsWithName Operation,” on page 82
- “getWorkflowTokenBusinessState Operation,” on page 83
- “getWorkflowTokenForId Operation,” on page 83
- “getWorkflowTokenResult Operation,” on page 84
- “getWorkflowTokenStatus Operation,” on page 84
- “hasChildrenInRelation Operation,” on page 85
- “hasRights Operation,” on page 86
- “sendCustomEvent Operation,” on page 87
- “simpleExecuteWorkflow Operation,” on page 88
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
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.

**echo Operation**

The echo operation tests the connection to the Web service by returning a String message.

The echo operation is declared as follows.

```java
public String echo(String echo);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>echo</td>
<td>An arbitrary String. If the Web service connection is working correctly, it returns the String.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns the same String as you provide as an input 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
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 echoWorkflow operation takes a Workflow 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);
```

The plug-in that contains the objects that you are looking for parses the query. The plug-in defines the query language that the `find` operation uses. 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.

The following table describes the `find` operation `query` parameter syntax and behavior for each of the supported Orchestrator plug-ins.

**Table 5-1. 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. Orchestrator sets the search string in a SQL <code>WHERE</code> 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>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>Stores nothing in the inventory.</td>
</tr>
<tr>
<td>JDBC</td>
<td>Not applicable</td>
<td>Stores nothing in the inventory.</td>
</tr>
<tr>
<td>Library</td>
<td>Not applicable</td>
<td>Stores nothing in the inventory.</td>
</tr>
<tr>
<td>Mail</td>
<td>Not applicable</td>
<td>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>Stores nothing in the inventory.</td>
</tr>
</tbody>
</table>
Table 5-1. 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>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>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 that the query returns. If the value of totalCount exceeds the number of objects in the array of FinderResult objects, the array does not include all of 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 type parameter with a name that matches the query string is returned in the QueryResult object.</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. The find operation 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>parentType</code> property can be the name of a plug-in, or it can specify a more narrowly defined parent. For example, you can specify the <code>parentType</code> as &quot;VC:&quot; to obtain the objects at the root of VMware vCenter Server 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 <code>parentId</code> 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Calling <code>findRelation</code> returns all children elements under a parent identified by its <code>parentId</code>. If you omit the <code>parentId</code> the <code>parentType</code> is not the root type of the inventory, the <code>findRelation</code> operation returns null. See “Relation Types,” on page 78 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 5-2. Standard Orchestrator Relation Types

<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>IpAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IPV4Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MacAddressPool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NetworkDomain</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subnet</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Range</td>
<td></td>
</tr>
<tr>
<td>SSH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>File</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Folder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RootFolder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SshConnection</td>
<td></td>
</tr>
<tr>
<td>vCenter Server</td>
<td>getComputeResource_ClusterComputeResource()</td>
<td>ClusterComputeResource</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>getDatastore()</td>
<td>Datastore</td>
</tr>
<tr>
<td></td>
<td>getDatastoreFolder()</td>
<td>DatastoreFolder</td>
</tr>
<tr>
<td></td>
<td>getFolder()</td>
<td>DatacenterFolder</td>
</tr>
<tr>
<td></td>
<td>getFolder()</td>
<td>DatastoreFolder</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>getHostSystem</td>
<td>HostSystem</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>
</tbody>
</table>

XML: No relations No relations
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 5-2, 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.

### 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(String username, String password);
```

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.

**getWorkflowInputForId Operation**

The `getWorkflowInputForId` operation retrieves the answer to a user interaction for an `interactionId` object.

The `getWorkflowInputForId` operation is declared as follows.

```java
public WorkflowInput getWorkflowInputForId(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>id</td>
<td>ID of the workflow input 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 a WorkflowInput object for a specific workflow input that corresponds to the provided workflow input ID.

getWorkflowInputForWorkflowTokenId Operation

The getWorkflowInputForWorkflowTokenId operation retrieves the answer to a user interaction for a workflowTokenId object.

The getWorkflowInputForWorkflowTokenId operation is declared as follows.

```java
public WorkflowInput getWorkflowInputForWorkflowTokenId(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 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 WorkflowInput object for a specific workflow token that corresponds to the provided workflow token ID.

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.

**IMPORTANT** The getWorkflowsWithName operation is a convenient means of obtaining workflows, but you should not use it in production applications because workflow names can change. Use the getWorkflowForId operation rather than the getWorkflowsWithName operation in production applications.

<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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value of the workflowName property can be a full name or a wildcard (*), which returns all the workflows available to the user. You can also search for partial names. For example, if you enter <em>Clone or Clone</em> as the workflowName, this returns all workflows with names that contain the word Clone.</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 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.

getWorkflowTokenBusinessState Operation

The getWorkflowTokenBusinessState operation retrieves the business state of a workflow token by a workflowTokenId object.

Activities that are part of the workflow’s schema can change the current business state of the workflow.

The getWorkflowTokenBusinessState operation is declared as follows.

```java
public WorkflowToken getWorkflowTokenBusinessState(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 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 the business state of a WorkflowToken object for a specific workflow token that corresponds to the provided workflow token ID.

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 that a `WorkflowToken` object produces 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. 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, `getWorkflowTokenResult` returns the correct dunesURI value.

If the object that `getWorkflowTokenResult` obtains is a plain Java object, you can deserialize 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 `o11nplugin-vsphere41.jar` in the classpath. You find the `o11nplugin-vsphere41.jar` file in `install-directory\VMware\Orchestrator\app-server\server\vmo\tmp\dars\o11nplugin-vsphere41.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.

**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 hasChildrenInRelation 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 parentId 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 findRelation operation returns all of that parent's children, regardless of the relation type. hasChildrenInRelation checks for the presence of only the children related by the desired relation type. This value can be null if you call hasChildrenInRelation on the root object of the hierarchy of objects.</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 vs0.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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes, children of the specified relation type are present</td>
</tr>
<tr>
<td>-1</td>
<td>No, children of the specified relation type are not present</td>
</tr>
<tr>
<td>0</td>
<td>Unknown, or an input parameter is invalid</td>
</tr>
</tbody>
</table>

**Related Information**

For more information, see “findRelation Operation,” on page 78.

**hasRights Operation**

The hasRights operation checks whether a user has permissions to view, edit, and run workflows.

To check the rights that you have on a workflow, you must have permission to view that workflow. If you have only edit or run permission on a workflow, you cannot view what rights you have on this workflow, and hasRights returns False.

A 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.

hasRights(workflowId, username, password, 'r')
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>workflowId</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    | n  a: The administrator can change the rights of the object.  
|           |          | 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.

```java
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.

- **internal / external**: The awaited event comes from another workflow (internal) or from a Web service application (external).
- **customEventKey**: The name of the awaited event.
- **myDate**: The date until which waitCustomEventUntil waits for a message from sendCustomEvent.

**simpleExecuteWorkflow Operation**

The simpleExecuteWorkflow operation uses string attributes to start a workflow.

**IMPORTANT** This operation is deprecated since Orchestrator 4.0. Do not use simpleExecuteWorkflow.
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.
Index

A
action
delete permissions 42
deleting 38
download 35
export 35
get permissions 41, 42
import 35
retrieve permissions 41, 42
upload 35
answerWorkflowInput 58, 74
API documentation 16
audience 7
authentication
LDAP 15
solution user name 15
vCenter Single Sign On 14
vco solution user name 15

C
cancelWorkflow 58, 74
configuration element
delete permissions 45
deleting 39
download 37
export 37
get permissions 44, 45
import 37
retrieve permissions 44, 45
upload 37

D
deleting objects 38
dunesUri 63, 70

E
echo operation 51, 75
echoWorkflow 75
executeWorkflow 75
executeWorkflow operation 57
export 34

F
filters 34
find
by ID 32
by type 32
objects 33
find objects 32
find operation 52
FinderResult 52–54, 63
findForId 77
findForId operation 52, 53
findRelation 52, 54, 78

G
getAllPlugins 80
getAllPlugins operation 51
getAllWorkflows 55, 56, 80
getWorkflowForId 81
getWorkflowForId 55, 57
getWorkflowInputForId 81
getWorkflowInputForWorkflowTokenId 82
getWorkflowsWithName 55, 56, 82
getWorkflowTokenBusinessState 83
getWorkflowTokenForId 58
getWorkflowTokenForId 83
getWorkflowTokenResult 60, 84
getWorkflowTokenStatus 58, 84
globalStatus 84

H
hasChildrenInRelation 85
hasRights 86
hasRights operation 57

I
import 34
inventory, search 32

J
Java REST SDK 16

L
LDAP
authentication 14
vCenter Single Sign On 14
ModuleInfo 64

object, find 32

package
delete permissions 43
deleting 38
download 36
export 36
get permissions 42, 43
import 36
retrieve permissions 42, 43
upload 36
permissions 39, 40
Property 65

QueryResult 52, 65

reference 16
resource
delete permissions 44
deleting 39
download 37
export 37
get permissions 43, 44
import 36
retrieve permissions 43, 44
upload 36
REST API, permissions 39, 40

schedule workflow 30
SDK, Java REST 16
sendCustomEvent 87
simpleExecuteWorkflow 88

task
cancel 29
delete 29
delete permissions 41
deleting 38
download 35
executions 24
export 35
filter 17
find 17
get executions 24
get permissions 40, 41
get runs 24
import 34
interact 24
interact while running 24
retrieve permissions 40, 41
run 19, 21, 29
runs 24
upload 34
user interaction 25, 27
validate 21
validate and run 21

vCO API 13
svoWebControl 51
waitForCustomEvent 87
Web service
client creation process 47
download examples 61
endpoint 49
generating stubs 49
HTTP connection 51
HTTPS connection 51
obtain results 60
operation arguments 49
running workflows 57
WSDL description 49
Web service API
objects 63
operations 73
Web services
create client 50
finding objects 52–54
finding workflows 55–57
time zones 61
workflow interaction 58
writing client application 47
workflow
answer to user interaction 25
cancel 29
delete 29
delete permissions 41
deleting 38
download 35
eexecutions 24
export 35
filter 17
find 17
get executions 24
get permissions 40, 41
get runs 24
import 34
interact 24
interact while running 24
retrieve permissions 40, 41
run 19, 21, 29
runs 24
upload 34
user interaction 25, 27
validate 21
validate and run 21
workflow ID 56
Workflow object 66
WorkflowParameter 67
WorkflowToken 67
WorkflowTokenAttribute 57
WorkFlowTokenAttribute 70
workflowTokenId 84