Developing a Web Services Client for VMware vCenter Orchestrator

vCenter Orchestrator 4.2.1

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.
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
executeWorkflow Operation 37
find Operation 38
findForId Operation 39
findRelation Operation 40
getAllPlugins Operation 42
getAllWorkflows Operation 42
getWorkflowForId Operation 43
getWorkflowsWithName Operation 43
getWorkflowTokenForId Operation 44
getWorkflowTokenResult Operation 44
getWorkflowTokenStatus Operation 45
hasChildrenInRelation Operation 46
hasRights Operation 47
sendCustomEvent Operation 48
simpleExecuteWorkflow Operation 49

Index 51
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 a Web services API so that you can develop applications to access workflows through Web services. The main purpose of the Orchestrator Web service is to start workflows and to retrieve their output parameters through a network or the Web.

**Intended Audience**

This information is intended for Web service application developers who want to access the Orchestrator processes across a network, through technologies such as simple object access protocol (SOAP) and Web services definition language (WSDL).
VMware vCenter Orchestrator provides a Web services API so that you can develop applications to access workflows through Web services. The main purpose of the Orchestrator Web service is to start workflows and to retrieve their output parameters through a network or the Web.

The Web service API provides a set of 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. The Web service API also allows you to implement tree viewers, based on the relations between objects obtained from plug-ins. The API has few complex object types and relatively few operations.

Note  To help understand how Orchestrator implements Web services, familiarize yourself with the Web services API for your development framework, for example Java or .Net.
Most applications that use the Orchestrator Web Service API have a common structure. To create Orchestrator Web service client applications, you perform a standard sequence of tasks.

This chapter includes the following topics:

- “Process for Creating an Orchestrator Web Service Client Application,” on page 9
- “Web Service Endpoint,” on page 11
- “Generating the Orchestrator Web Service Stubs,” on page 11
- “Accessing the Server from Web Service Clients,” on page 12
- “Create a Web Service Client,” on page 12
- “Time Zones and Running Workflows Through Web Services,” on page 23
- “Web Service Application Examples,” on page 23

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 \textit{orchestrator\_server} is the IP address or host name of the host on which the Orchestrator server is running.

\[
\text{http://\textit{orchestrator\_server}:8280/vmware-vmo-webcontrol/webservice}
\]

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.

\textbf{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.

\[
\text{http://\textit{orchestrator\_server}:8280/vmware-vmo-webcontrol/webservice?WSDL}
\]

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.

\textbf{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 \textit{VSOWebControlService} service as the point of access to the other Web service objects.

\textbf{Table 2-1. Java classes generated with Axis 1.4}

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSOWebControl</td>
<td>The Web service defines a WSDL port type named VSOWebControl, through which you access all the Orchestrator Web service operations.</td>
</tr>
<tr>
<td>WebServiceStub</td>
<td>The Web service defines client and server side stubs that the application uses to start the Web service.</td>
</tr>
<tr>
<td>VSOWebControlProxy</td>
<td>The Web service provides access to the Orchestrator Web service operations through a proxy.</td>
</tr>
<tr>
<td>VSOWebControlService</td>
<td>The VSOWebControlService service is a remote procedure call (RPC) Service implementation. The VSOWebControlService service is the point of access to the other Web service objects.</td>
</tr>
</tbody>
</table>
| VSOWebControlServiceLocator | The VSOWebControlServiceLocator service extends VSOWebControlService to provide the following operations.  
  - getwebserviceAddress obtains the endpoint URL for the Web service.  
  - getwebservice obtains the client-side stub for the Web service application and instantiates the VSOWebControl port type object with the appropriate endpoint URL. |
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. See *Administering VMware vCenter Orchestrator* for information about setting system properties.

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 13
   
   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 14
   
   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 14
   
   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 15
   
   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 16
   
   You can use the `findRelation` operation to locate the children of a particular object.

6. **Find Workflows in the Orchestrator Server** on page 17
   
   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 18
   
   The `getAllWorkflows` operation lists all workflows that a user can access as an array of `Workflow` objects.

8. **Find Workflows by Using the getWorkflowsWithName Operation** on page 18
   
   If you know the name of a particular workflow, as it is defined in the Orchestrator client, the Web service application can obtain this workflow using its name or part of its name.

9. **Retrieve the ID of a Workflow** on page 19
   
   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.
10 Find Workflows by Using the getWorkflowForID Operation on page 19
If you know a particular workflow ID, a Web service application can obtain this workflow by using the getWorkflowForID operation.

11 Run Workflows from a Web Service Client on page 19
The main purpose of a Web services client is to run workflows across a network.

12 Interact with a Workflow While it Runs on page 20
After the workflow starts, the Web services client can perform various actions in response to events while the workflow is running.

13 Obtain Workflow Results on page 22
After the workflow completes its run, you can retrieve the results by calling the getWorkflowTokenResult( ) operation.

Connect to the Orchestrator Web Service
Web service applications establish connections to the Orchestrator server through simple object access protocol (SOAP) binding, using either the HTTP or HTTPS protocols.

Prerequisites
You must have generated the Orchestrator Web service client and server stubs from the Orchestrator WSDL definition. You must create a Web service client application class that implements the VS0WebControl interface.

Procedure
1 In your Web service client application class, create a VS0WebControl instance that connects to the Web service endpoint.
   You can either create an unsecured connection using HTTP, or a secure connection using HTTPS. The default HTTP port is 8280 and the default HTTPS port is 8281. The URL is also a default.
   - The following example shows how to create an HTTP connection to the Web service.
     String urlprefix = "http://10.0.0.1:8280" ;
     URL url = new URL(urlprefix + "/vmware-vmo-webcontrol/webservice");
     VS0WebControl vs0WebControl = new VS0WebControlServiceLocator().getwebservice(url);
   - The following example shows how to create an HTTPS connection to the Web service.
     String urlprefix = "https://10.0.0.1:8281" ;
     URL url = new URL(urlprefix + "/vmware-vmo-webcontrol/webservice");
     vs0WebControl = new VS0WebControlServiceLocator().getwebservice(url);

2 Check the server connections by calling the echo operation.
   The following example shows how you can call the echo operation.
   vs0WebControl.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.
   ModuleInfo[] modules = vs0WebControl.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);
   }
   ```
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.

  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.

   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 = vs0WebControl.findRelation("VC:ComputeResource", "vcenter/domain-s114", "getResourcePool()");
   
   The preceding example returns an array of FinderResult objects that match the following criteria.
   
   - The parent element is of the type VC:ComputeResource.
   - The parent element's ID is vchost/domain-s114.
   - The returned children are related to the parent by the getResourcePool relation, defined by the Orchestrator vCenter Server 4 plug-in.

2. Extract the results from a FinderResult object.

   To show, interpret, or process the results in the FinderResult objects that the find operation returns, you must convey these results to the Web service application.

   ```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.

```java
Workflow[] workflows = vsoWebControl.getAllWorkflows(username, password);
```

The preceding code example calls getAllWorkflows to get an array of Workflow objects that the Web service client can run.

You found workflows in the Orchestrator server that the Web service client can run on objects.

What to do next
Implement operations in the Web services client to run the workflows it finds.

Find Workflows by Using the getWorkflowsWithName Operation

If you know the name of a particular workflow, as it is defined in the Orchestrator client, the Web service application can obtain this workflow using its name or part of its name.

The getWorkflowsWithName operation returns an array of workflows, so you can use it to match several workflows by using wildcards.

Prerequisites
You must have implemented Web service operations in your client application to find objects in the Orchestrator server.

Procedure
- Create an array of Workflow objects by calling the getWorkflowsWithName operation.

```java
Workflow[] workflows =
    vsoWebControl.getWorkflowsWithName("Simple user interaction", username, password);
```

The preceding code example calls the getWorkflowsWithName operation to obtain all workflows for which the name, or part of the name, is Simple user interaction.

You found workflows in the Orchestrator server that the Web service client can run on objects.

What to do next
Implement operations in the Web services client to run the workflows it finds.
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 getWorkflowForID Operation

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

The getWorkflowForID operation returns a single Workflow instance, because all workflow IDs are unique.

Prerequisites
You must have implemented Web service operations in your client application to find objects in the Orchestrator server.

Procedure
1. Create a Workflow object by calling the getWorkflowForID operation.

   ```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.

   ```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.

   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.

   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.

   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("answer2");
     attributes[1] = attribute;
   }
attribute.setValue("123");
attributes[1] = attribute;
vsowebcontrol.answerWorkflowInput tokenId, 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 tokenId, 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() + ");

Developing a Web Services Client for VMware vCenter Orchestrator

VMware, Inc.
Emit the workflow token’s result attributes for display or for use by other applications.

```java
for (int ii = 0; ii < retAttributes.length; ii++) {
    System.out.println("\tName:'" + 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.SimpleDateFormat 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 serve as WSDL operations.

This chapter includes the following topics:
- “FinderResult Object,” on page 25
- “ModuleInfo Object,” on page 26
- “Property Object,” on page 27
- “QueryResult Object,” on page 27
- “Workflow Object,” on page 28
- “WorkflowParameter Object,” on page 29
- “WorkflowToken Object,” on page 29
- “WorkflowTokenAttribute Object,” on page 32

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><code>type</code></td>
<td>Type of object found.</td>
</tr>
<tr>
<td>String</td>
<td><code>id</code></td>
<td>ID of the discovered object.</td>
</tr>
<tr>
<td>Array of properties</td>
<td><code>properties</code></td>
<td>A list of the discovered object's properties.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The format of the <code>properties</code> values is defined by each plug-in in its <code>vso.xml</code> file, under the <code>FinderResult</code> description.</td>
</tr>
<tr>
<td>String</td>
<td><code>dunesUri</code></td>
<td>A string representation of the object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If a <em>FinderResult</em> object is accessed through a plug-in, it is identified by a <code>dunesUri</code> string, rather than by another type of string or ID. The format of the <code>dunesUri</code> is as follows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dunes://service.dunes.ch/CustomSDKObject?id='object_ID'&amp;dunesName='&lt;plug-in_name&gt;:&lt;object_type&gt;'</code></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><code>moduleName</code></td>
<td>Name of the plug-in, used as a prefix in object names.</td>
</tr>
<tr>
<td>String</td>
<td><code>moduleVersion</code></td>
<td>Plug-in version.</td>
</tr>
<tr>
<td>String</td>
<td><code>moduleDescription</code></td>
<td>Description of the plug-in.</td>
</tr>
<tr>
<td>String</td>
<td><code>moduleDisplayName</code></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.

```
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.

```
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 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 FinderResults 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.

- 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>
<tbody>
<tr>
<td>String</td>
<td>globalState</td>
<td>The state of the workflow as a whole, with the following possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- running: the workflow is running</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- waiting: the workflow is waiting for runtime parameters, which can be provided by answerWorkflowInput</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- waiting-signal: the workflow is waiting for an external event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- canceled: the workflow was canceled by a user or by an application</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- completed: the workflow has finished</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- failed: the workflow encountered an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The globalState is the state of the workflow as a whole.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You must run getWorkflowTokenForId every time you update this value.</td>
</tr>
<tr>
<td>String</td>
<td>startDate</td>
<td>The date and time that this workflow token started</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The startDate value is set at the moment the workflow starts. When you obtain a token, its startDate has already been initialized.</td>
</tr>
<tr>
<td>Type</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>String</td>
<td>endDate</td>
<td>Date and time that this workflow token ended, if the workflow token has finished. The <code>endDate</code> value is filled in at the moment the workflow reaches the end of its run. The <code>endDate</code> is only set when the workflow finishes in one of the completed, failed or canceled states.</td>
</tr>
<tr>
<td>String</td>
<td>xmlContent</td>
<td>Defines input parameters, output parameters, attributes, and the content of error messages. The values of the attributes and parameters are set in CDATA elements and error messages are set in <code>&lt;exception&gt;</code> tags, as the following example shows.</td>
</tr>
</tbody>
</table>

```
<token>
  <atts>
    <stack>
      <att n='attstr' t='string' e='n'>
        <![CDATA[attribute]]>Attribute value</att>
      <att n='instr' t='string' e='n'>
        <![CDATA[[]]>Input parameter value</att>
      <att n='outstr' t='string' e='n'>
        <![CDATA[[]]>Output parameter value</att>
    </stack>
  </atts>
  <exception encoded='n'>Error message</exception>
</token>
```

**WorkflowTokenAttribute Object**

A `WorkflowTokenAttribute` object represents an input or output parameter of a running instance of a workflow.

A `WorkflowTokenAttribute` is a value that you pass to a predefined `WorkflowParameter` when a `WorkflowToken` begins, or in some cases, at runtime. When you run a workflow, you supply the input parameters for that particular workflow as `WorkflowTokenAttribute` objects. The `executeWorkflow` operation takes an array of `WorkflowTokenAttribute` objects as an argument when you call it, as the following example shows.

```java
public WorkflowToken executeWorkflow(String workflowId, String username, String password, WorkflowTokenAttribute[] attributes);
```

Workflows also use `WorkflowTokenAttribute` as the output parameter of a run workflow. `WorkflowTokenAttribute` contains the results of a completed `WorkflowToken` created by running `executeWorkflow`. You can collect the result of a `WorkflowToken`, in the form of a `WorkflowTokenAttribute`, by calling `getWorkflowTokenResult`, as the following example shows.

```java
public WorkflowTokenAttribute[] getWorkflowTokenResult(String workflowTokenId, String username, String password);
```
You can also pass an array of WorkflowTokenAttribute objects to the answerWorkflowInput operation to provide input that a workflow token needs while it runs.

```java
public void answerWorkflowInput(String workflowTokenId, WorkflowTokenAttribute[] answerInputs, String username, String password);
```

The following table shows the properties of the WorkflowTokenAttribute object.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>name</td>
<td>Name of the input or output parameter</td>
</tr>
<tr>
<td>String</td>
<td>type</td>
<td>Type of input or output parameter</td>
</tr>
</tbody>
</table>
| String | value | The value property represents either the input or output parameter value for this particular workflow token, in the form of a string. If the type is an array of objects, the value is a string of the following format: "#{<type1>#<value1>#{type2>#<value2>#{...}}#" If the value property specifies an object obtained from a plug-in, then the input or output parameter value is a dunesUri string that points to the object in question. The following example shows the format of the dunesUri.  
dunes://service.dunes.ch/Custom SDKObject?id='<object_ID>'&dunesName='plugin_name:object_type' |

VMware, Inc.  33
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 36
- “cancelWorkflow Operation,” on page 36
- “echo Operation,” on page 37
- “echoWorkflow Operation,” on page 37
- “executeWorkflow Operation,” on page 37
- “find Operation,” on page 38
- “findForId Operation,” on page 39
- “findRelation Operation,” on page 40
- “getAllPlugins Operation,” on page 42
- “getAllWorkflows Operation,” on page 42
- “getWorkflowForId Operation,” on page 43
- “getWorkflowsWithName Operation,” on page 43
- “getWorkflowTokenForId Operation,” on page 44
- “getWorkflowTokenResult Operation,” on page 44
- “getWorkflowTokenStatus Operation,” on page 45
- “hasChildrenInRelation Operation,” on page 46
- “hasRights Operation,” on page 47
- “sendCustomEvent Operation,” on page 48
- “simpleExecuteWorkflow Operation,” on page 49
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);
```
Type | Value | Description
--- | --- | ---
String | workflowTokenId | The identifier of the running workflow to cancel
String | username | Orchestrator user name
String | password | Orchestrator password

**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);
```

Type | Value | Description
--- | --- | ---
String | echo | An arbitrary String. If the Web service connection is working correctly, it returns the String.

**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);
```

Type | Value | Description
--- | --- | ---
Workflow | workflow | The `echoWorkflow` operation takes a `Workflow` object as a parameter. If the connection and serialization are working correctly, it returns the same workflow.

**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 4-1. 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>
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 <code>QueryResult</code> 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. The parentType property can be the name of a plug-in, or it can specify a more narrowly defined parent. For example, you can specify the parentType as &quot;VC:&quot; to obtain the objects at the root of VMware vCenter Server plug-in, or you can a specific folder, such as &quot;VC:VmFolder&quot;.</td>
</tr>
<tr>
<td>String</td>
<td>parentId</td>
<td>The ID of a particular parent object. The parentId parameter allows you to find the children of a specific parent object, if you know its ID.</td>
</tr>
<tr>
<td>String</td>
<td>relation</td>
<td>The name of the relation. Calling findRelation returns all children elements under a parent identified by its parentId. If you omit the parentId the parentType is not the root type of the inventory, the findRelation operation returns null. See “Relation Types,” on page 40 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 4-2. 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>■ IpAddress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ IPV4Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ MacAddressPool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ NetworkDomain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Proxy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Subnet</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Range</td>
</tr>
<tr>
<td>SSH</td>
<td></td>
<td>■ File</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Folder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ RootFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ SshConnection</td>
</tr>
<tr>
<td>vCenter Server</td>
<td></td>
<td>■ getComputeResource_ClusterComputeResource() ClusterComputeResource</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getComputeResource_ComputeResource() ComputeResource</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getDatacenter() Datacenter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getDatadstore() Datastore</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getDatadstoreFolder() DatastoreFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getFolder() DatacenterFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getFolder() DatastoreFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getFolder() DatacenterFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getFolder() HostFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getFolder() NetworkFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getHost() VmFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getHostFolder() HostFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getNetwork() HostSystem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getNetworkFolder() Network</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getNetwork_DistributedVirtualPortgroup() NetworkFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getNetwork_Network() DistributedVirtualPortgroup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getOwner() ComputeResource</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getOwner() VmFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getParentFolder() VirtualApp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getPortgroup() VirtualApp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getRecentTask() VmFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getRecentTask() VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getRecentTask() VirtualMachineSnapshot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getSdkConnections() VmFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getSdkConnections() VirtualApp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getVm() DatacenterFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getVm() SdkConnection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getVmFolder() VirtualMachine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getVmFolder() VmFolder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getVmSnapshot() VirtualMachineSnapshot</td>
</tr>
<tr>
<td>XML</td>
<td>No relations</td>
<td>No relations</td>
</tr>
</tbody>
</table>
The relation property can also reference relation types specified in each plug-in's vso.xml file. The following example is an excerpt from the networking plug-in vso.xml file.

```xml
<relations>
  <relation name="Subnet" type="Class:Subnet"/>
  <relation name="Range" type="Class:Range"/>
  <relation name="NetworkDomain" type="Class:NetworkDomain"/>
  <relation name="MacAddressPool" type="Class:MacAddressPool"/>
</relations>
```

In addition to the relation types listed in Table 4-2, 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.

**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.
**Type** | **Value** | **Description**
--- | --- | ---
String | workflowName | Name of the workflow to find. 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 *Clone or Clone* as the workflowName, this returns all workflows with names that contain the word Clone.

String | username | Orchestrator user name.
String | password | Orchestrator password.

**Return Value**

Returns an array of Workflow objects that correspond to the provided name or name fragment. Workflows are returned in an array even if only one workflow is found. Returns null if you pass it an invalid parameter.

**getWorkflowTokenForId Operation**

The getWorkflowTokenForId operation finds the WorkflowToken object for a specific workflow token.

The getWorkflowTokenForId operation is declared as follows.

```java
public WorkflowToken getWorkflowTokenForId(String workflowTokenId, String username, String password);
```

Individual threads or functions can run multiple workflows. The getWorkflowTokenForId operation allows a central process or thread to track the progress of each workflow. Using getWorkflowTokenForId provides access to all the information about a specific WorkflowToken because, although checking the token status only requires the ID, it is often useful to obtain all the information about a given token.

**Type** | **Value** | **Description**
--- | --- | ---
String | workflowTokenId | ID of this run of the workflow
String | username | Orchestrator user name.
String | password | Orchestrator password.

**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);
```
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>
</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 40.

### 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    | - `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: “name1=value1\nname2=value2\nname3=value3” 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>workflowId</td>
<td>ID of the Workflow to be run.</td>
</tr>
<tr>
<td>String</td>
<td>username</td>
<td>Orchestrator user name.</td>
</tr>
<tr>
<td>String</td>
<td>password</td>
<td>Orchestrator password.</td>
</tr>
<tr>
<td>String</td>
<td>attributes</td>
<td>The format for the attributes parameter is a list of attributes separated by commas. Because commas are used as separators, attribute name strings containing commas are not processed correctly. Each attribute is represented by its name, type, and value, as shown in the following examples. Name1,Type1,Value1,Name2,Type2,Value2</td>
</tr>
</tbody>
</table>

**Return Value**

Runs a workflow. Returns a WorkflowToken object.
Index

A
answerWorkflowInput 20, 36
audience 5

C
cancelWorkflow 20, 36

D
dunesUri 25, 32

E
echo operation 13, 37
echoWorkflow 37
executeWorkflow 37
executeWorkflow operation 19

F
find 27, 38
find operation 14
FinderResult 14–16, 25
findForId 39
findForId operation 14, 15
findRelation 14, 16, 40

G
getAllPlugins 42
getAllPlugins operation 13
getAllWorkflows 17, 18, 42
getWorkflowForId 17, 19
getWorkflowsWithName 17, 18, 43
getWorkflowTokenForId 20
getWorkflowTokenForId 44
getWorkflowTokenResult 22, 44
globalStatus 45

H
hasChildrenInRelation 46
hasRights 47
hasRights operation 19

M
ModuleInfo 26

P
Property 27

Q
QueryResult 14, 27

S
sendCustomEvent 48
simpleExecuteWorkflow 49

V
vsoWebControl 13

W
waitForCustomEvent 48
Web service
access server 12
client creation process 9
download examples 23
endpoint 11
generating stubs 11
HTTP connection 13
HTTPS connection 13
obtain results 22
operation arguments 11
running workflows 19
WSDL description 11
Web service API
objects 25
operations 35
Web services
create client 12
finding objects 14–16
finding workflows 17–19
time zones 23
workflow interaction 20
writing client application 9
workflow ID 19
Workflow object 28
WorkflowParameter 29
WorkflowToken 29
WorkflowTokenAttribute 19
WorkflowTokenAttribute 32
workflowTokenId 45