Developing a Web Services Client for VMware vCenter Orchestrator

vRealize Orchestrator 5.5.1

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

If you have comments about this documentation, submit your feedback to

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Copyright © 2008–2014 VMware, Inc. All rights reserved. Copyright and trademark information.

Contents

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*.

Revision	Description
EN-001342-01	Added information about the location of Java REST SDK artifacts in Using the Java REST SDK.
EN-001342-00	Initial release.

Developing a Web Services Client

1

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 2

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 programing language.

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

This section includes the following topics:

- Performing Server Configuration Operations
- Performing Tagging Operations

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 issued by the vCenter Single Sign-On Server.

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*/vco/api/.

Note The default port number is 8281.

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*/vco/api/inventory/System/
- https://*vcoHost*:*port*/vco/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*/vco/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 `VCOAuthorization` 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.

Example: Obtain a Session in vCenter Single Sign-On Mode

The following example code obtains a session in vCenter Single Sign-On mode.

```
URI uri = URI.create("https://vco-server:8283/vco/api");
VcoSessionFactory sessionFactory = new DefaultVcoSessionFactory(uri);

//provide the address of the vCenter Single Sign-On server
URI ssoUri = URI.create("https://sso-server:7444/ims/STSService?wsdl");

//set the tokens to be valid for an hour
long lifeTimeSeconds = 60 * 60;

//create a factory for vCenter Single Sign-On tokens
SsoAuthenticator sso = new SsoAuthenticator(ssoUri, sessionFactory, lifeTimeSeconds);

//provide vCenter Single Sign-On credentials
SsoAuthentication authentication = sso.createSsoAuthentication("username", "password");

VcoSession session = sessionFactory.newSession(authentication);
//use session here
```

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}/vco/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 if 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 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.

Example: Obtain a Session in LDAP Mode

The following example code obtains a session in LDAP mode.

```
URI uri = URI.create("https://vco-server:8283/vco/api");
VcoSessionFactory sessionFactory = new DefaultVcoSessionFactory(uri);

//provide LDAP credentials
```

```
Authentication auth = new UsernamePasswordAuthentication("username", "password");

VcoSession session = sessionFactory.newSession(auth);
//use session here
```

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*/vco/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 REST SDK artifacts are available at the following locations.

Note You can only access the artifacts if you have deployed the Orchestrator Appliance.

- https://*orchestrator_host*:*port*/vco-repo/com/vmware/o11n/o11n-rest-client/
- https://*orchestrator_host*:*port*/vco-repo/com/vmware/o11n/o11n-rest-client-examples/
- https://*orchestrator_host*:*port*/vco-repo/com/vmware/o11n/o11n-rest-client-services/
- https://*orchestrator_host*:*port*/vco-repo/com/vmware/o11n/o11n-rest-client-stubs/

Example: Run a Workflow and Wait for Its Completion

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

```
//start a new session to Orchestrator by using specified credentials
VcoSession session = DefaultVcoSessionFactory.newLdapSession(new URI("https://orchestrator-server:
8281/vco/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
ExecutionContext context = new ExecutionContextBuilder().addParam("name", "Jerry").addParam("age",
18).build();

//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}/vco/api/workflows?conditions=name={workflowFullName}

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

     GET https://{vcoHost}:{port}/vco/api/inventory/System/Workflows/
     ```

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

```
GET https://{vcoHost}:{port}/vco/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}/vco/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/vco/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/vco/api/catalog/System/Workflow/CF808080808080808080808080808080E6808080013086668236014a0614d1
6e1/">
       <attributes>
          <attribute name="id"
value="CF808080808080808080808080808080E6808080013086668236014a0614d16e1"/>
          <attribute name="canExecute" value="true" />
          <attribute name="description" value="" />
          <attribute name="name" value="Interactive Hello World" />
          <attribute name="type" value="Workflow"/>
          <attribute name="canEdit" value="true"/>
      </attributes>
    </link>
    <link rel="down"
      href="https://localhost:
8281/vco/api/catalog/System/Workflow/CF808080808080808080808080808080DA808080013086668236014a0614d1
6e1/">
      <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>
   </link>
</inventory-items>
```

2 Make a GET request at the URL of the inventory item of the Send Hello workflow:

```
GET https://localhost:
8281/vco/api/catalog/System/Workflow/CF808080808080808080808080808080DA808080013086668236014a0614d1
6e1/
```

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

```
<xml version="1.0" encoding="UTF-8" standalone="yes">
<inventory-item xmlns="http://www.vmware.com/vco"
   href="https://localhost:
8281/vco/api/catalog/System/Workflow/CF808080808080808080808080808080DA808080013086668236014a0614d1
6e1/">
   <relations>
      <link rel="down"
       href="https://localhost:
8281/vco/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>
```

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

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

You receive the definition of the Send Hello workflow in the response body:

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

```
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/presentati
on/" />
        <link rel="down"
            href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/tasks/" />
        <link rel="down"
            href="https://localhost:
8281/vco/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}/vco/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}/vco/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/vco/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/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/">
   <relations>
      <link rel="up"
        href="https://localhost:8281/vco/api/inventory/System/Workflows/Samples/HelloWorld/" />
      <link rel="add"
        href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/execution
s/" />
      <link rel="down"
        href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/execution
s/" />
      <link rel="down"
        href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/presentati
on/" />
      <link rel="down"
           href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/tasks/" />
      <link rel="down"
        href="https://localhost:
8281/vco/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>
```

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

```
POST https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/execution
s/
```

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}/vco/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://{vco host}:{port}/vco/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}/vco/api/workflows/{workflowID}/presentation/instances/
```

5 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.

6 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}/vco/api/workflows/{workflowID}/presentation/instances/{executionID}/
```

7 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.

8 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}/vco/api/workflows/{workflowID}/executions/
```

9 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.

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

```
GET https://localhost:
8281/vco/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/vco/api/workflows/CF808080808080808080808080808080DA80800013086668236014a0614d16e1/">
   <relations>
      <link rel="up"
          href="https://localhost:8281/vco/api/inventory/System/Workflows/Samples/HelloWorld/" />
      <link rel="add"
          href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA80800013086668236014a0614d16e1/execution
s/" />
      <link rel="down"
          href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA80800013086668236014a0614d16e1/execution
s/" />
      <link rel="down"
          href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA80800013086668236014a0614d16e1/presentati
on/" />
      <link rel="down"
             href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA80800013086668236014a0614d16e1/tasks/" />
      <link rel="down"
          href="https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA80800013086668236014a0614d16e1/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>
```

2 Make a `GET` request at the URL that holds the definition of the workflow presentation:

```
GET https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA80800013086668236014a0614d16e1/presentati
on/
```

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

```
POST https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA80800013086668236014a0614d16e1/presentati
on/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>
    <constraints>
       <number-range max="15.0" min="3.0" />
    </constraints>
.......
```

4 Make a `POST` request at the URL that holds the particular presentation instance:

```
POST https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/presentati
on/instances/888080808080808080808080808080803F8080800132145338690643f66a027ec/
```

In the request body, provide values for the input parameters:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 <execution-context xmlns="http://www.vmware.com/vco">
  <parameters>
    <parameter name="name" type="string">
      <string>John Smith</string>
    </parameter>
  </parameters>
</execution-context>
```

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/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/execution
s/
```

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}/vco/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}/vco/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}/vco/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}/vco/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}/vco/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/vco/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/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/execution
s/
```

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

```
GET https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/execution
s/8880808080808080808080808080808803A808080013214533869064 3f66a027ec/
```

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>
<name>Send Hello</name>
......
```

Answer to a Waiting User Interaction

You can answer to a waiting user interaction of a workflow run 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 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:

URL	Description
https://*vcoHost*:*port*/vco/api/catalog/System/UserInteraction	Holds the available user interaction objects in Orchestrator
https://*vcoHost*:*port*/vco/api/catalog/System/UserInteraction?status=0	Filters only the waiting user interaction objects.

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}/vco/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}/vco/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:

```
GET https://localhost:8281/vco/api/catalog/System/UserInteraction?status=0
```

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/vco/api/catalog/System/UserInteraction/888080808080808080808080808080805A808080013214533869064
3f66a027ec/
```

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

```
POST https://localhost:
8281/vco/api/workflows/CF808080808080808080808080808080E6808080013086668236014a0614d16e1/execution
s/88808080808080808080808080808080578080800132145338690643f66a027ec/interaction/
```

Provide a value for the input parameter 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>
```

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:

URL	Description
https://*vco host*:*port*/vco/api/catalog/System/UserInteraction	Holds the available user interaction objects in Orchestrator.
https://*vco host*:*port*/vco/api/catalog/System/UserInteraction?status=0	Filters only the waiting user interaction objects.

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}/vco/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}/vco/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}/vco/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 interation presentation by making a POST request at the URL where the instances of the presentation reside:

```
POST https://{vcoHost}:
{port}/vco/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}/vco/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:

```
GET https://localhost:8281/vco/api/catalog/System/UserInteraction?status=0
```

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/vco/api/catalog/System/UserInteraction/8880808080808080808080808080805A808080013214533869064
3f66a027ec/
```

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

```
GET https://localhost:
8281/vco/api/catalog/System/UserInteraction/8880808080808080808080808080808080805A808080013214533869064
3f66a027ec/interaction/
```

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

```
GET https://localhost:
8281/vco/api/catalog/System/UserInteraction/8880808080808080808080808080808080805A808080013214533869064
3f66a027ec/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:

```
POST https://localhost:
8281/vco/api/catalog/System/UserInteraction/8880808080808080808080808080808080805A808080013214533869064
3f66a027ec/interaction/presentation/instances/
```

Provide a value for the input parameter 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>
```

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:

```
............
<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/vco/api/catalog/System/UserInteraction/8880808080808080808080808080808080805A808080013214533869064
3f66a027ec/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}/vco/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}/vco/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}/vco/api/workflows/{workflowID}/executions/{executionID}/
   ```

Retrieve a Workflow's Interactions

You can retrieve the list of all user interactions for 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}/vco/api/workflows/{workflowID}/
   ```

2. Get the list of workflow interactions by making a GET request to the URL of the workflow's interactions:

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

If the GET request is successful, you receive the status code 200 and a list of all user interactions available for the workflow.

Access a Workflow's Schema

You can access the schema image 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}/vco/api/workflows/{workflowID}/
   ```

2. Get the workflow's schema image by making a `GET` request to the URL of the workflow's schema:

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

If the `GET` request is successful, you receive the status code 200 and the icon's schema image binary data. The response content type is set to a correct media type, for example `Content-Type:image/png`.

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.

The maximum number of scheduled tasks supported by Orchestrator is 50.

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}/vco/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}/vco/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/vco/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/vco/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/vco/api/workflows/CF808080808080808080808080808080DA808080013086668236014a0614d16e1/">
        <name>Send Hello</name>
    </workflow>
    <start-mode>normal</start-mode>
</task>
```

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}/vco/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. 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 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 instancess 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}/vco/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}/vco/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 requets.

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}/vco/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}/vco/api/catalog/{plug-in namespace}/
    ```

-
  ```
  https://{vcoHost}:{port}/vco/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}/vco/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}/vco/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/vco/api/catalog/
```

2 Make a GET request at the URL where all system objects in Orchestrator are located:

```
GET https://localhost:8281/vco/api/catalog/System/
```

3 Make GET request at the URL where all workflows reside:

```
GET https://localhost:8281/vco/api/catalog/Workflow/
```

4 Make GET request at the URL of the Send Hello workflow:

```
GET https://localhost:
8281/vco/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}/vco/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/vco/api/inventory/
   ```

2. Make a GET request at the URL where the system objects in Orchestrator reside:

   ```
   GET https://localhost:8281/vco/api/inventory/System/
   ```

3. Make GET request at the URL where all workflows in Orchestrator reside:

   ```
   GET https://localhost:8281/vco/api/inventory/System/Workflows/
   ```

4. Make a GET request at the URL of the Samples workflow category:

   ```
   GET https://localhost:8281/vco/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/vco/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/vco/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 by using 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}/vco/api/workflows/
```

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

Export a Workflow

You can export a workflow by using 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}/vco/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 by using 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}/vco/api/actions/
```

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

Export an Action

You can export an action by using 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}/vco/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 by using 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.

By default, if you import an Orchestrator package with a duplicate name, the existing package is not overwritten. You can specify whether to overwrite existing packages by using a parameter in the request.

By default, Orchestrator packages are imported with the attribute values of configuration elements. You can import a package without attribute values by using a parameter in the request.

By default, tags contained in Orchestrator packages are imported, but if the same tags already exist on the Orchestrator server, the values of existing tags are preserved. You can specify whether existing tag values are preserved by using parameters in the request.

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}/vco/api/packages/
```

3 (Optional) To import a package and overwrite an existing package with the same name, use the `overwrite` parameter in the `POST` request:

```
POST http://{vcoHost}:{port}/vco/api/packages/?overwrite=true
```

4 (Optional) To import a package without the attribute values of the configuration elements from the package, use the `importConfigurationAttributeValues` parameter in the `POST` request:

```
POST http://{vcoHost}:{port}/vco/api/packages/?importConfigurationAttributeValues=false
```

5 (Optional) To import a package without the tags that it contains, use the `tagImportMode` parameter in the `POST` request:

```
POST http://{vcoHost}:{port}/vco/api/packages/?tagImportMode=DoNotImport
```

6 (Optional) To import a package with the tags that it contains and overwrite existing tag values, use the `tagImportMode` parameter in the `POST` request:

```
POST http://{vcoHost}:{port}/vco/api/packages/?tagImportMode=ImportAndOverwriteExistingValue
```

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

Export a Package

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

By default, Orchestrator packages are exported with attribute values of configuration elements and global tags. You can export a package without attribute values or global tags by using parameters in the request. You can also specify a custom name for the package file that you download.

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}/vco/api/packages/{package_name}/
```

3 (Optional) To set a custom name for the exported package, use the `packageName` parameter in the `GET` request:

```
GET http://{vcoHost}:{port}/vco/api/packages/{package_name}/?packageName={custom_name}
```

4 (Optional) To export a package without the attribute values of the configuration elements from the package, use the `exportConfigurationAttributeValues` parameter in the `GET` request:

```
GET http://{vcoHost}:{port}/vco/api/packages/{package_name}/?
exportConfigurationAttributeValues=false
```

5 (Optional) To export a package without global tags, use the `exportGlobalTags` parameter in the `GET` request:

```
GET http://{vcoHost}:{port}/vco/api/packages/{package_name}/?exportGlobalTags=false
```

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 by using 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}/vco/api/resources/
```

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

Export a Resource

You can export a resource by using 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}/vco/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 by using 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}/vco/api/configurations/
```

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

Export a Configuration Element

You can export a configuration element by using 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}/vco/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}/vco/api/workflows/
```

2 Make a `DELETE` request at the URL of the workflow:

```
DELETE http://{vcoHost}:{port}/vco/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}/vco/api/actions/
```

2 Make a `DELETE` request at the URL of the action:

```
DELETE http://{vcoHost}:{port}/vco/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}/vco/api/packages/
```

2 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}/vco/api/packages/{package_name}/?option={parameter}
```

Parameter	Description
`deletePackage`	Only the package is deleted, while its content is retained.
`deletePackageWithContent`	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.
`deletePackageKeepingShared`	The package and the content that is not shared is deleted. Elements that are shared with other packages are not deleted.

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}/vco/api/resources/
   ```

2. Make a DELETE request at the URL of the resource:

   ```
   DELETE http://{vcoHost}:{port}/vco/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}/vco/api/configurations/
   ```

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

   ```
   DELETE http://{vcoHost}:{port}/vco/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

Character	Description
`r`	Gives view permissions.
`x`	Gives execute permissions.
`i`	Gives inspect permissions.
`c`	Gives edit permissions.
`a`	Gives administrative permissions.

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.

```
<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}/vco/api/workflows/
```

2 Make a `GET` request at the URL of the workflow's permissions:

```
GET http://{vcoHost}:{port}/vco/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}/vco/api/workflows/
```

2 Make a `DELETE` request at the URL of the workflow's permissions:

```
DELETE http://{vcoHost}:{port}/vco/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.

Procedure

1 Make a `GET` request and retrieve the ID of the workflow from the list of returned workflows:

```
GET http://{vcoHost}:{port}/vco/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}/vco/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}/vco/api/actions/
```

2 Make a GET request at the URL of the action's permissions:

```
GET http://{vcoHost}:{port}/vco/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}/vco/api/actions/
```

2 Make a DELETE request at the URL of the action's permissions:

```
DELETE http://{vcoHost}:{port}/vco/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.

Procedure

1 Make a GET request and retrieve the ID of the action from the list of returned actions:

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

2 Make a `GET` request at the URL of the package's permissions:

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

2 Make a `DELETE` request at the URL of the package's permissions:

```
DELETE http://{vcoHost}:{port}/vco/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.

Procedure

1 Make a GET request and retrieve the name of the package from the list of returned packages:

```
GET http://{vcoHost}:{port}/vco/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}/vco/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}/vco/api/resources/
```

2 Make a GET request at the URL of the resource's permissions:

```
GET http://{vcoHost}:{port}/vco/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}/vco/api/resources/
```

2 Make a DELETE request at the URL of the resource's permissions:

```
DELETE http://{vcoHost}:{port}/vco/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.

Procedure

1 Make a GET request and retrieve the ID of the resource from the list of returned resources:

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

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

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

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

```
DELETE http://{vcoHost}:{port}/vco/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.

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}/vco/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}/vco/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.

Performing Operations with Plug-Ins

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

Retrieve Information About Plug-Ins

You can retrieve metadata information for all installed plug-ins by using the Orchestrator REST API.

Procedure

1 In a REST client application, add request headers to define the properties of the plug-ins.

2 Make a GET request at the URL of the plug-in objects:

```
GET http://{vcoHost}:{port}/vco/api/plugins/
```

If the GET request is successful, you receive the status code 200.

Import a Plug-In

You can import a plug-in by using the Orchestrator REST API.

Depending on the library of your REST client application, you can use a custom code that defines the properties of the plug-in.

Note You cannot import a plug-in if a plug-in with the same name is already installed.

Prerequisites

The plug-in 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 plug-in that you want to import.

2 Make a POST request at the URL of the plug-in objects:

```
POST http://{vcoHost}:{port}/vco/api/plugins/
```

If the POST request is successful, you receive the status code 200.

Export a Plug-In

You can export a plug-in by using the Orchestrator REST API.

Procedure

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

- **Name**: **`accept`**
- **Value**: **`application/dar`**

2 Make a GET request at the URL of the plug-in that you want to export:

```
GET http://{vcoHost}:{port}/vco/api/plugins/{plug-in_name}/
```

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

Enable or Disable a Plug-In

You can enable or disable a plug-in by using the Orchestrator REST API.

You can change the state of a plug-in from enabled to disabled, or from disabled to enabled, by making a PUT request at the URL of the plug-in. You can check the current state of a plug-in by retrieving information about the Orchestrator plug-ins. See Retrieve Information About Plug-Ins.

Prerequisites

The plug-in 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 plug-in that you want to enable or disable.

2 Make a PUT request at the URL of the plug-in that you want to enable or disable:

```
PUT http://{vcoHost}:{port}/vco/api/plugins/{plug-in_name}/state/
```

If the PUT request is successful, you receive the status code 200.

Performing Server Configuration Operations

The Orchestrator REST API provides Web services that you can use to perform various operations related to the Orchestrator server configuration.

Retrieve Information About the Orchestrator Server Configuration

You can retrieve information about the Orchestrator server configuration by using the Orchestrator REST API.

Procedure

1. In a REST client application, add request headers to define the properties of the server for which you want to retrieve information.
2. Make a GET request at the URL of the plug-in objects:

   ```
   GET http://{vcoHost}:{port}/vco/api/server-configuration/
   ```

If the GET request is successful, you receive the status code 200.

Import Orchestrator Server Configuration

You can import a saved configuration by using the Orchestrator REST API.

Prerequisites

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

Procedure

1. In a REST client application, add a request header with the following values.
   - **Name**: **`content-type`**
   - **Value**: **`multipart/form-data`**
2. Make a POST request at the URL of the server configuration:

   ```
   POST http://{vcoHost}:{port}/vco/api/server-configuration/
   ```

If the POST request is successful, you receive the status code 200.

Export Orchestrator Server Configuration

You can export the server configuration by using the Orchestrator REST API.

Prerequisites

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

Procedure

1. In a REST client application, add a request header with the following values.
   - **Name**: `content-type`
   - **Value**: `multipart/form-data`
2. Add another request header with the following values.
   - **Name**: `accept`
   - **Value**: `*/*`
3. Make a `POST` request at the URL of the server configuration:

   ```
   POST http://{vcoHost}:{port}/vco/api/server-configuration/
   ```

If the `POST` request is successful, you receive the status code 200.

Performing Tagging Operations

The Orchestrator REST API provides Web services that you can use to perform various operations to make objects more searchable by using tags in Orchestrator.

You can make objects more searchable by attaching tags to them. Tags are strings with length between 3 and 64 characters and must contain no whitespace characters.

You can add global and private tags. Global tags are visible to all Orchestrator users and private tags are visible only to the user who created them. Global tags can be created and removed only by users with administrative privileges.

Tag an Object

You can assign tags to an object by using the Orchestrator REST API.

You can create both private and global tags. You specify whether the tag is private or global in the body of the request.

Note To create global tags, you must be logged in as a user with administrative privileges.

You can also assign a value to the tag that you create. The value is an optional parameter that you can use to filter tags.

Procedure

1 Define the request body by using the following syntax.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tag-instance xmlns="http://www.vmware.com/vco" global="false">
  <name>tag_name</name>
  <value>tag_value</value>
</tag-instance>
```

Note You can create a global tag by setting the **`global`** variable to **`"true"`**.

2 Make a `POST` request at the URL of the object:

```
POST http://{vcoHost}:{port}/vco/api/catalog/{namespace}/{objectType}/{objectId}/tags
```

If the `POST` request is successful, you receive the status code 200.

Untag an Object

You can remove tags assigned to an object by using the Orchestrator REST API.

You can remove both private and global tags.

Note To remove global tags, you must be logged in as a user with administrative privileges.

Procedure

◆ Make a `DELETE` request to remove private or global tags.

- To remove a private tag, make a `DELETE` request at the URL of the object by using the following syntax:

  ```
  DELETE http://{vcoHost}:
  {port}/vco/api/catalog/{namespace}/{objectType}/{objectId}/tag/{tag_name}
  ```

- To remove a global tag, make a `DELETE` request at the URL of the object by using the following syntax:

  ```
  DELETE http://{vcoHost}:{port}/vco/api/catalog/{namespace}/{objectType}/{objectId}/tag/:
  {tag_name}
  ```

If the `DELETE` request is successful, you receive the status code 200.

List Object Tags

You can retrieve a list of tags assigned to an object by using the Orchestrator REST API.

Procedure

- Make a GET request at the URL of the object:

  ```
  GET http://{vcoHost}:{port}/vco/api/catalog/{namespace}/{objectType}/{objectId}/tags
  ```

If the GET request is successful, you receive the status code 200.

List Tagged Objects by Type

You can use the Orchestrator REST API to retrieve a list of objects tagged with a specific tag and filter them by object type.

Procedure

- Make a GET request at the URL of the object type:

  ```
  GET http://{vcoHost}:{port}/vco/api/catalog/{namespace}/{objectType}/?tags=tag1&tags=:tag2=value
  ```

If the GET request is successful, you receive the status code 200.

List Tag Owners

You can retrieve a list of tag owners by using the Orchestrator REST API. Tag owners are users who have created at least one tag.

Procedure

- Make a GET request at the following URL:

  ```
  GET http://{vcoHost}:{port}/vco/api/tags
  ```

If the GET request is successful, you receive the status code 200. The list that you retrieve contains users who have created at least one tag. Global tags are listed under the system user name `__GLOBAL__`.

List Tags by Users

You can use the Orchestrator REST API to retrieve a list of tags created by a specific user.

You can also retrieve global tags. Global tags are listed under the system user name `__GLOBAL__`.

Procedure

- Make a GET request at the URL of the user.
  - To retrieve a list of the tags created by a specific user, make a GET request by using the following syntax:

    ```
    GET http://{vcoHost}:{port}/vco/api/tags/{user_name}
    ```

  - To retrieve a list of global tags, make a GET request by using the following syntax:

    ```
    GET http://{vcoHost}:{port}/vco/api/tags/__GLOBAL__
    ```

If the GET request is successful, you receive the status code 200.

List Tags by Users Filtered by Tag Name

You can use the Orchestrator REST API to retrieve a list of tag instances created by a specific user and filter the tags by tag name.

You can also retrieve global tag instances. Global tags are listed under the system user name __GLOBAL__.

Procedure

- Make a GET request at the URL of the user.
  - To retrieve a filtered list of the tag instances created by a specific user, make a GET request by using the following syntax:

    ```
    GET http://{vcoHost}:{port}/vco/api/tags/{user_name}/{tag_name}
    ```

  - To retrieve a filtered list of global tag instances, make a GET request by using the following syntax:

    ```
    GET http://{vcoHost}:{port}/vco/api/tags/__GLOBAL__/{tag_name}
    ```

If the GET request is successful, you receive the status code 200. The information that you retrieve contains a reference to the tagged object, tag name, tag value, and an indication whether the tag instance is global or private.

Remove Tags by Users

You can use the Orchestrator REST API to remove all tags created by a specific user.

You can also remove global tags. Global tags are listed under the system user name __GLOBAL__.

Note To remove global tags, you must be logged in as a user with administrative privileges.

Procedure

- Make a `DELETE` request at the URL of the user.
  - To remove the tags created by a specific user, make a `DELETE` request by using the following syntax:

    ```
    DELETE http://{vcoHost}:{port}/vco/api/tags/{user_name}
    ```

  - To remove the global tags, make a `DELETE` request by using the following syntax:

    ```
    DELETE http://{vcoHost}:{port}/vco/api/tags/__GLOBAL__
    ```

If the `DELETE` request is successful, you receive the status code 200.

Writing a Client Application for the Orchestrator SOAP Service

3

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.

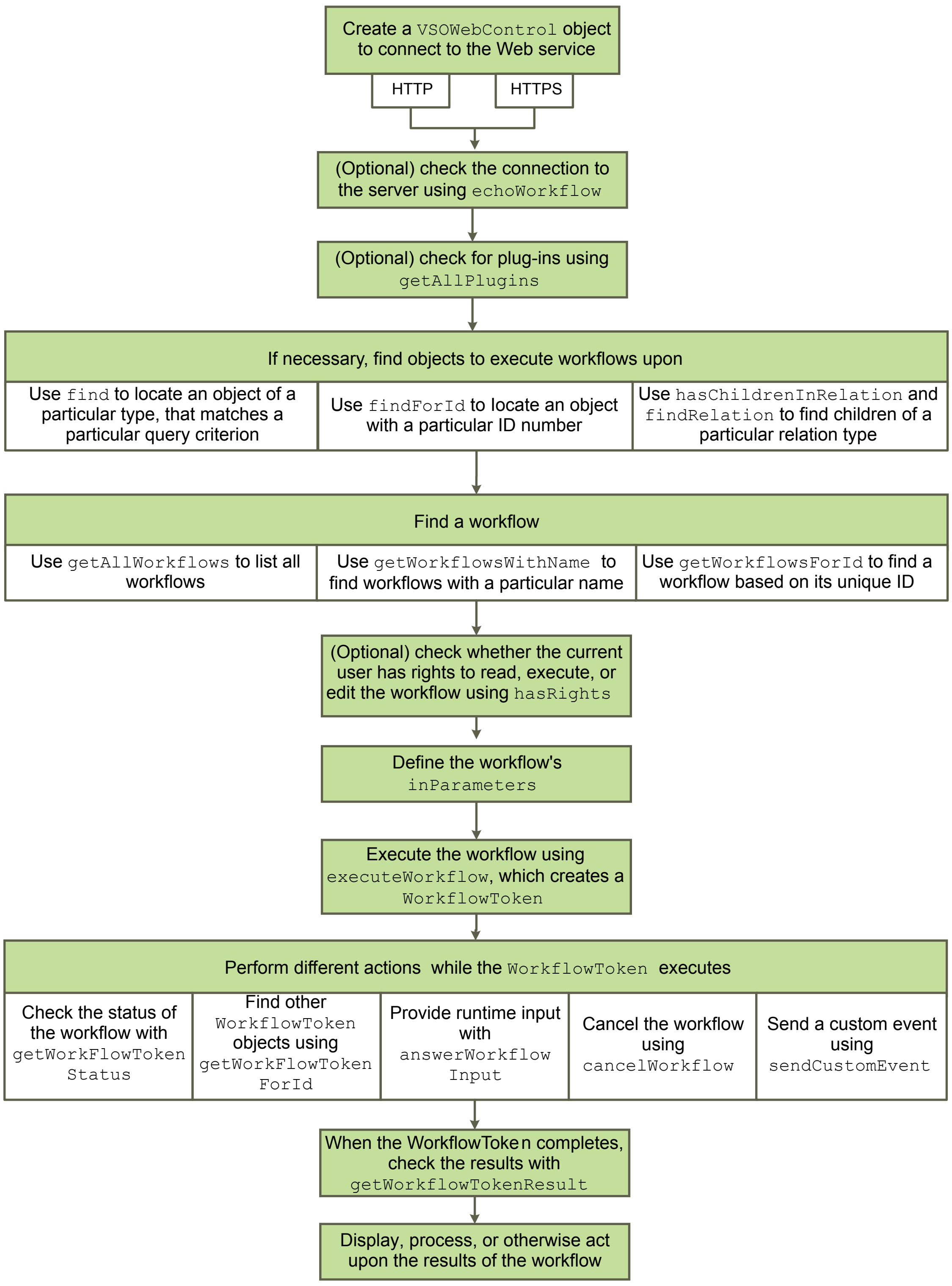
This section includes the following topics:

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.

Figure 3-1. Process for Creating Orchestrator Web Service Applications


Create a VSOWebControl object to connect to the Web service
HTTP
HTTPS
(Optional) check the connection to the server using echoWorkflow
(Optional) check for plug-ins using getAllPlugins
If necessary, find objects to execute workflows upon
Use find to locate an object of a particular type, that matches a particular query criterion
Use findForId to locate an object with a particular ID number
Use hasChildrenInRelation and findRelation to find children of a particular relation type
Find a workflow
Use getAllWorkflows to list all workflows
Use getWorkflowsWithName to find workflows with a particular name
Use getWorkflowsForId to find a workflow based on its unique ID
(Optional) check whether the current user has rights to read, execute, or edit the workflow using hasRights
Define the workflow's inParameters
Execute the workflow using executeWorkflow, which creates a WorkflowToken
Perform different actions while the WorkflowToken executes
Check the status of the workflow with getWorkFlowToken Status
Find other WorkflowToken objects using getWorkFlowToken ForId
Provide runtime input with answerWorkflow Input
Cancel the workflow using cancelWorkflow
Send a custom event using sendCustomEvent
When the WorkflowToke n completes, check the results with getWorkflowTokenResult
Display, process, or otherwise act upon the results of the workflow

Follow the broad stages of development illustrated to create Orchestrator Web services client applications that satisfy most of your requirements.

Web Service Endpoint

The Web service endpoint is the port upon which you connect a Web service client to the Orchestrator server.

You connect to the Orchestrator Web service's endpoint at the following URL, in which *orchestrator_server* is the IP address or host name of the host on which the Orchestrator server is running.

https://*orchestrator_server*:8281/vco/vmware-vmo-webcontrol/webservice

By default, the Web service runs over HTTPS on port 8281 of the Orchestrator server. Access to the Web service API requires a valid user name and password on the Orchestrator server.

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.

https://*orchestrator_server*:8281/vco/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.

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` service as the point of access to the other Web service objects.

Table 3-1. Java classes generated with Axis 1.4

Class	Description
`VSOWebControl`	The Web service defines a WSDL port type named `VSOWebControl`, through which you access all the Orchestrator Web service operations.
`WebServiceStub`	The Web service defines client and server side stubs that the application uses to start the Web service.
`VSOWebControlProxy`	The Web service provides access to the Orchestrator Web service operations through a proxy.

Table 3-1. Java classes generated with Axis 1.4 (Continued)

Class	Description
`VSOWebControlService`	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.
`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. 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

Web service applications use the HTTPS protocol to establish connections to the Orchestrator server through simple object access protocol (SOAP) binding.

2 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.

3. 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.

4. 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.

5. Find Objects by Using the findRelation Operation

   You can use the `findRelation` operation to locate the children of a particular object.

6. 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.

7. Find Workflows by Using the getAllWorkflows Operation

   The `getAllWorkflows` operation lists all workflows that a user can access as an array of `Workflow` objects.

8. 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.

9. 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.

10. 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.

11. Run Workflows from a Web Service Client

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

12. 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.

13. Obtain Workflow Results

    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 use the HTTPS protocol to establish connections to the Orchestrator server through simple object access protocol (SOAP) binding.

Prerequisites

- Verify that you have generated the Orchestrator Web service client and server stubs from the Orchestrator WSDL definition.
- Verify that you have created 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.

   The default HTTPS port is 8281. The URL is also a default.

   The following example shows how to create a connection to the Web service.

   ```
   String urlprefix = "https://10.0.0.1:8281/vco" ;
   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.

   ```
   vsoWebControl.echo(string);
   ```

   The call to the `echo` operation returns the String object that you provided as an argument.

3. (Optional) To check which plug-ins are running on the Orchestrator server, call the `getAllPlugins` operation.

   The following example shows how you can call the `getAllPlugins` operation.

   ```
   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.

```
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);
            }
            else {
                System.out.println("Found nothing");
            }
```

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.

```
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.

```
FinderResult[] results = vsoWebControl.findRelation("VC:ComputeResource",
    "vcenter/domain-s114", "getResourcePool()", "username", "password");
```

The preceding example returns an array of `FinderResult` objects that match the following criteria.

- The parent element is of the type `VC:ComputeResource`.
- The parent element's ID is `vchost/domain-s114`.
- The returned children are related to the parent by the `getResourcePool` relation, defined by the Orchestrator vCenter Server 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.

```
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 Workflows in the Orchestrator Server

When you have found the objects with which to interact, you must find the workflows that perform these interactions.

The Orchestrator Web service API includes the following operations to find all the workflows running in a given environment, to find a workflow with a particular name, or to find workflows with a particular ID.

- `getAllWorkflows`
- `getWorkflowsWithName`
- `getWorkflowForID`

Find Workflows by Using the getAllWorkflows Operation

The `getAllWorkflows` operation lists all workflows that a user can access as an array of `Workflow` objects.

Because the `getAllWorkflows` operation returns `Workflow` objects that contain all the information about a workflow, it is useful for applications that require full information about workflows, such as the workflow's name, ID, description, parameters, and attributes.

Prerequisites

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

Procedure

- Create an array of `Workflow` objects by calling the `getAllWorkflows` operation.

  ```
  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.

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.

  ```
  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.

  ```
  String workflowId = "1880808080808080808080808080808087808080011713796199469943be4c882";
  Workflow workflow = vsoWebControl.getWorkflowForID(workflowId, username, password);
  ```

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

What to do next

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

Run Workflows from a Web Service Client

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

Prerequisites

You must have implemented Web service operations in the client to find workflows in the Orchestrator server.

Procedure

1 (Optional) Check the workflow user permissions by calling the hasRights operation.

You can verify if a user has rights to read, run, or edit a particular workflow using the hasRights operation. This operation is not mandatory, but checking user rights before you run a workflow can help prevent exceptions.

```
String workflowId = "1880808080808080808080808080808087808080011713796199469943be4c882";
Boolean rights = vsoWebControl.hasRights(workflowId, username, password, 'x');
```

The preceding code example calls the hasRights operation to discover whether the user has the right to run the workflow identified by workflowId.

If the user has the right to run the workflow, hasRights returns true. Otherwise, hasRights returns false.

2 Set the workflow attributes in a WorkflowTokenAttribute object.

The Web services client passes WorkflowTokenAttributes arrays to a WorkflowToken object, which runs the workflow.

```
WorkflowTokenAttribute[] attributes = new WorkflowTokenAttribute[1];
WorkflowTokenAttribute attribute = new WorkflowTokenAttribute();
attribute.setName("vm");
attribute.setType(finderResult.getType());
attribute.setValue(finderResult.getDunesUri());
attributes[0] = attribute;
```

The preceding example creates a WorkflowTokenAttribute object, then populates it with the following information:

- The name of the attribute, in this case, vm.
- The type of attribute, as discovered in a FinderResult object defined elsewhere in the code.
- The attribute value, which in this case is a dunesUri string, signifying that the value specifies an object accessed through a plug-in.

3 Run the workflow by calling the executeWorkflow operation.

To run a workflow, you pass the workflow attributes to the executeWorkflow operation in the form of a WorkflowTokenAttribute array.

Running a workflow creates a WorkflowToken object, which represents the instance of the workflow that runs with the specific input parameters that it receives when it starts.

```
WorkflowToken token = vsoWebControl.executeWorkflow(workflowId, username, password, attributes);
```

In the preceding example, the attributes property is the array of WorkflowTokenAttribute objects created in Step 2.

Sometimes, workflows require input parameters during their run. In these cases, you can provide attributes through a user interaction while the workflow is running. You can pass attributes to the workflow during its run using the `answerWorkflowInput` operation.

You implemented operations in the Web service client that check user permissions, pass attributes to a workflow, and run the workflow.

What to do next

Implement operations in the Web services client to interact with workflows while they run.

Interact with a Workflow While it Runs

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

Prerequisites

You must have implemented operations in the Web service client to run workflows in the Orchestrator server.

Procedure

1 Find running workflows by calling the `getWorkflowTokenForId` operation.

   Calling `getWorkflowTokenForId` obtains a `WorkflowToken` object, which contains all of the information about that specific workflow token.

```
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("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("\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.

```
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.

Web Service API Object Reference

4

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 section includes the following topics:

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.

```
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.

Type	Value	Description
String	`type`	Type of object found.
String	`id`	ID of the discovered object.
Array of properties	`properties`	A list of the discovered object's properties. The format of the `properties` values is defined by each plug-in in its `vso.xml` file, under the `FinderResult` description.
String	`dunesUri`	A string representation of the object. 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. `dunes://service.dunes.ch/CustomSDKObject?id='<object_ID>'&dunesName='<plug-in_name>:<object_type>'`

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.

```
public ModuleInfo[] getAllPlugins(username, password);
```

The following table shows the properties of the `ModuleInfo` object.

Type	Value	Description
String	`moduleName`	Name of the plug-in, used as a prefix in object names.
String	`moduleVersion`	Plug-in version.

Type	Value	Description
String	moduleDescription	Description of the plug-in.
String	moduleDisplayName	Plug-in name shown in the Orchestrator inventory.

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.

Type	Value	Description
String	name	Property name.
String	value	Property value. The format of a property's values is defined by each plug-in in its vso.xml file, under the FinderResult description.

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.

Type	Value	Description
Long	`totalCount`	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 `FinderResults` returned does not include all the objects found in the queried inventory.
`FinderResult[]`	`elements`	An array of `FinderResult` objects.

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.

Type	Value	Description
String	`id`	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.
String	`name`	The name of the workflow, as it appears in the workflow's **Name** text box in Orchestrator.
String	`description`	A detailed description of what the workflow does.

Type	Value	Description
`WorkflowParameter[]`	`inParameters`	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.
`WorkflowParameter[]`	`outParameters`	The `outParameters` array is the set of `WorkflowParameter` objects that result from running a workflow. This array allows the workflow to send errors, the names of any created objects, and other information as output. You can set up arbitrary output parameters to generate any information that you need.
`WorkflowParameter[]`	`attributes`	The `attributes` array is a set of `WorkflowParameter` objects that represent constants and preset variables for a given workflow. Attributes differ from `inParameters` because they are intended to represent environmental constants or variables, rather than runtime information. **Note** You cannot retrieve workflow attribute values by using the Web service. You can only retrieve output parameter values.

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.

Type	Value	Description
String	`name`	The parameter name.
String	`type`	The parameter type.

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.

```
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.

Type	Value	Description
String	`id`	The identifier of this particular instance of a completed workflow.
String	`title`	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.
String	`workflowId`	The identifier of the workflow of which this `WorkflowToken` object is a running instance.
String	`currentItemName`	The name of the step in the workflow that is running at the moment when `getWorkflowTokenForId` is called.

Type	Value	Description
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** You should not use `currentItemState`. The `globalState` property makes `currentItemState` redundant.
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 ■ `suspended`: the workflow run is paused 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.
		`<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.

```
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.

```
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.

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

The following table shows the properties of the `WorkflowTokenAttribute` object.

Type	Value	Description
String	`name`	Name of the input or output parameter
String	`type`	Type of input or output parameter

String	`value`	The `value` property represents either the input or output parameter value for this particular workflow token, in the form of a string.
		If the `type` is an array of objects, the `value` is a string of the following format:
		`"#{#<type1>#<value1>#;#<type2>#<value2>#...}#"`
		If the `value` property specifies an object obtained from a plug-in, then the input or output parameter value is a `dunesUri` string that points to the object in question. The following example shows the format of the `dunesUri`.
		`dunes://service.dunes.ch/CustomSDKObject?id='<object_ID>'&dunesName='<plug-in_name>:<object_type>'`

Web Service API Operation Reference

5

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 section includes the following topics:

- hasRights Operation
- sendCustomEvent Operation
- simpleExecuteWorkflow Operation

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.

```
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.

Type	Value	Description
String	`workflowTokenId`	The ID of a running workflow that is waiting for input from a user interaction or external application
Array of `WorkflowTokenAttribute` objects	`answerInputs`	The result of the user interaction or external application, passed as input to the waiting workflow
String	`username`	Orchestrator user name
String	`password`	Orchestrator password

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.

```
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.

```
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.

```
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.

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

Type	Value	Description
String	`workflowId`	The identifier of the workflow to run
String	`username`	Orchestrator user name
String	`password`	Orchestrator password
Array of `WorkflowTokenAttribute` instances	`workflowInputs`	Array of input parameters required to run the workflow

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.

```
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

Orchestrator Plug-In	Query Parameter Syntax	Query Behavior
Database, for example Lifecycle Manager	String	Searches for object names in SQL database tables. Orchestrator sets the search string in a SQL `WHERE` keyword search. It searches the primary keys, then the object IDs in the database.
Enumeration	Not applicable	Stores nothing in the inventory. You can find enumerations on each data type that contains enumeration types.
Jakarta common set	Not applicable	Stores nothing in the inventory.
JDBC	Not applicable	Stores nothing in the inventory.
Library	Not applicable	Stores nothing in the inventory.
Mail	Not applicable	Stores nothing in the inventory.
SSH	If you have configured Orchestrator to use SSH connections, you can make queries SSH commands.	Stores nothing in the inventory.
vCenter Server	String or null	Ignores the query string and returns all objects of the specified type.
XML	Not applicable	Stores nothing in the inventory.

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.

Type	Value	Description
String	`type`	Type of object looked for.
String	`query`	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.
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

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.

```
public FinderResult findForId(String type, String id, String username,
String password);
```

Type	Value	Description
String	`type`	Type of object looked for.
String	`id`	ID of the object looked for.

Type	Value	Description
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

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.

```
public FinderResult[] findRelation(String parentType, String parentId,
String relation, String username, String password);
```

Type	Value	Description
String	`parentType`	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 `"VC:"` to obtain the objects at the root of VMware vCenter Server plug-in, or you can a specific folder, such as `"VC:VmFolder"`.
String	`parentId`	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.
String	`relation`	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 for more information.
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

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

Plug-In	Relation Names	Relation Types
Enumerations	No relations	No relations
Jakarta Commons Net	No relations	No relations
JDBC	No relations	No relations
Library	No relations	No relations
Mail	No relations	No relations
Networking		■ `IpAddress` ■ `IPV4Address` ■ `MacAddressPool` ■ `NetworkDomain` ■ `Proxy` ■ `Subnet` ■ `Range`
SSH		■ `File` ■ `Folder` ■ `RootFolder` ■ `SshConnection`

Table 5-2. Standard Orchestrator Relation Types (Continued)

Plug-In	Relation Names	Relation Types
vCenter Server	■ getComputeResource_ClusterComputeResource() ■ getComputeResource_ComputeResource() ■ getDatacenter() ■ getDatastore() ■ getDatastoreFolder() ■ getFolder() ■ getFolder() ■ getFolder() ■ getFolder() ■ getFolder() ■ getHost() ■ getHostFolder() ■ getNetwork() ■ getNetworkFolder() ■ getNetwork_DistributedVirtualPortgroup() ■ getNetwork_Network() ■ getOwner() ■ getParentFolder() ■ getPortgroup() ■ getRecentTask() ■ getResourcePool() ■ getResourcePool_ResourcePool() ■ getResourcePool_VirtualApp() ■ getRootFolder() ■ getSdkConnections() ■ getVm() ■ getVmFolder() ■ getVmSnapshot()	■ ClusterComputeResource ■ ComputeResource ■ Datacenter ■ Datastore ■ DatastoreFolder ■ DatacenterFolder ■ DatastoreFolder ■ HostFolder ■ NetworkFolder ■ VmFolder ■ HostSystem ■ HostFolder ■ Network ■ NetworkFolder ■ DistributedVirtualPortgroup ■ Network ■ ComputeResource ■ VmFolder ■ DistributedVirtualPortgroup ■ Task ■ ResourcePool ■ ResourcePool ■ VirtualApp ■ DatacenterFolder ■ SdkConnection ■ VirtualMachine ■ VmFolder ■ VirtualMachineSnapshot
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.

```
[...]
<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.

```
public ModuleInfo[] getAllPlugins(username, password);
```

The following table describes the `getAllPlugins` operation properties.

Type	Value	Description
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

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.

```
public Workflow[] getAllWorkflows(String username, String password);
```

Type	Value	Description
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

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.

```
String workflowId = workflows[0].getId();
```

The `getWorkflowForId` operation is declared as follows.

```
public Workflow getWorkflowForId(String workflowId, String username, String  password);
```

Type	Value	Description
String	`workflowId`	ID of the workflow to retrieve.
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

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.

```
public WorkflowInput getWorkflowInputForId(String id, String username, String password);
```

Type	Value	Description
String	`id`	ID of the workflow input to retrieve.
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

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.

```
public WorkflowInput getWorkflowInputForWorkflowTokenId(String workflowTokenId, String username,
String password);
```

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 `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.

```
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.

getWorkflowTokenBusinessState Operation

The `getWorkflowTokenBusinessState` operation retrieves the business state of a workflow token for 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.

```
public WorkflowToken getWorkflowTokenBusinessState(String workflowTokenId, String username, String
password);
```

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

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 ID.

The `getWorkflowTokenForId` operation is declared as follows.

```
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.

```
public WorkflowTokenAttribute[] getWorkflowTokenResult(String workflowTokenId,
String username, String password);
```

Type	Value	Description
String	`workflowTokenId`	ID of this specific run of the workflow
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

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
- `suspended`: the workflow run is paused

The `getWorkflowTokenStatus` operation is declared as follows.

```
public String[] getWorkflowTokenStatus(String[] workflowTokenID, String username,
String password);
```

Type	Value	Description
Array of strings	`workflowTokenId`	List of workflow token IDs.
String	`username`	Orchestrator user name.
String	`password`	Orchestrator 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.

```
public int hasChildrenInRelation(String parentType, String parentId, String relation, String username,
String password);
```

Type	Value	Description
String	`parentType`	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.
String	`parentId`	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.
String	`relation`	The type of relation by which children are related to their parents. Relation types are specified in the `vso.xml` file for each plug-in.
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.

Return Value

Returns one of the following values:

`1`	Yes, children of the specified relation type are present
`–1`	No, children of the specified relation type are not present
`0`	Unknown, or an input parameter is invalid

Related Information

For more information, see findRelation Operation.

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')
```

Type	Value	Description
String	`workflowId`	The ID of the workflow for which you are checking a user's rights.
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.
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.

```
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.

Type	Value	Description
String	`eventName`	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.
String	`serializedProperties`	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.

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`.

Type	Value	Description
String	`workflowId`	ID of the Workflow to be run.
String	`username`	Orchestrator user name.
String	`password`	Orchestrator password.
String	`attributes`	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`

Return Value

Runs a workflow. Returns a `WorkflowToken` object.