

Developing with VMware vRealize Orchestrator

vRealize Orchestrator 7.6

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

<https://docs.vmware.com/>

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

Copyright © 2008-2019 VMware, Inc. All rights reserved. [Copyright and trademark information.](#)

Contents

Developing with VMware vRealize Orchestrator 10

1 Developing Workflows 11

Key Concepts of Workflows 13

Workflow Parameters 13

Workflow Attributes 14

Workflow Schema 14

Workflow Presentation 14

Workflow Tokens 14

Phases in the Workflow Development Process 15

Best Practices for Developing Workflows 15

Access Rights for the Orchestrator Client 16

Testing Workflows During Development 16

Creating and Editing a Workflow 16

Create a Workflow 17

Edit a Workflow 17

Edit a Workflow from the Standard Library 18

Workflow Editor Tabs 18

Provide General Workflow Information 19

Defining Attributes and Parameters 20

Define Workflow Parameters 21

Define Workflow Attributes 22

Attribute and Parameter Naming Restrictions 23

Workflow Schema 24

View Workflow Schema 25

Building a Workflow in the Workflow Schema 25

Schema Elements 29

Schema Element Properties 32

Links and Bindings 35

Decisions 42

Exception Handling 45

Using Error Handlers 46

Foreach Elements and Composite Types 48

Add a Switch Activity to a Workflow 51

Developing Plug-Ins 52

Overview of Plug-Ins 52

Contents and Structure of a Plug-In 61

Orchestrator Plug-In API Reference 66

Elements of the vso.xml Plug-In Definition File	77
Best Practices for Orchestrator Plug-In Development	95
Obtaining Input Parameters from Users When a Workflow Starts	111
Creating the Input Parameters Dialog Box In the Presentation Tab	111
Setting Parameter Properties	113
Requesting User Interactions While a Workflow Runs	117
Add a User Interaction to a Workflow	118
Set the User Interaction security.group Attribute	119
Set the timeout.date Attribute to an Absolute Date	120
Calculate a Relative Timeout for User Interactions	121
Set the timeout.date Attribute to a Relative Date	122
Define the External Inputs for a User Interaction	123
Define User Interaction Exception Behavior	124
Create the Input Parameters Dialog Box for the User Interaction	126
Respond to a Request for a User Interaction	127
Calling Workflows Within Workflows	128
Workflow Elements that Call Workflows	128
Call a Workflow Synchronously	131
Call a Workflow Asynchronously	132
Schedule a Workflow	133
Prerequisites for Calling a Remote Workflow from Within Another Workflow	134
Call Several Workflows Simultaneously	135
Running a Workflow on a Selection of Objects	136
Implement the Start Workflows in a Series and Start Workflows in Parallel Workflows	137
Developing Long-Running Workflows	139
Set a Relative Time and Date for Timer-Based Workflows	139
Create a Timer-Based Long-Running Workflow	140
Create a Trigger Object	142
Create a Trigger-Based Long-Running Workflow	144
Configuration Elements	145
Create a Configuration Element	145
Validating Workflows	146
Validate a Workflow and Fix Validation Errors	147
Debugging Workflows	148
Debug a Workflow	148
Example Workflow Debugging	149
Running Workflows	150
Run a Workflow in the Workflow Editor	150
Run a Workflow	151
Resuming a Failed Workflow Run	153
Set the Behavior for Resuming a Failed Workflow Run	153

Set Custom Properties for Resuming Failed Workflow Runs	154
Resume a Failed Workflow Run	154
Generate Workflow Documentation	155
Use Workflow Version History	155
Develop a Simple Example Workflow	156
Create the Simple Workflow Example	158
Create the Schema of the Simple Workflow Example	159
Create the Simple Workflow Example Zones	162
Define the Parameters of the Simple Workflow Example	163
Define the Simple Workflow Example Decision Bindings	164
Bind the Action Elements of the Simple Workflow Example	165
Bind the Simple Workflow Example Scripted Task Elements	169
Define the Simple Workflow Example Exception Bindings	177
Set the Read-Write Properties for Attributes of the Simple Workflow Example	178
Set the Simple Workflow Example Parameter Properties	179
Set the Layout of the Simple Workflow Example Input Parameters Dialog Box	180
Validate and Run the Simple Workflow Example	182
Develop a Complex Workflow	183
Create the Complex Workflow Example	185
Create a Custom Action for the Complex Workflow Example	186
Create the Schema of the Complex Workflow Example	187
Create the Complex Workflow Example Zones	189
Define the Parameters of the Complex Workflow Example	191
Define the Bindings for the Complex Workflow Example	192
Set the Complex Workflow Example Attribute Properties	203
Create the Layout of the Complex Workflow Example Input Parameters	204
Validate and Run the Complex Workflow Example	205

2 Scripting 207

Orchestrator Elements that Require Scripting	207
Limitations of the Mozilla Rhino Implementation in Orchestrator	208
Using the Orchestrator Scripting API	209
Access the Scripting Engine from the Workflow Editor	210
Access the Scripting Engine from the Action or Policy Editor	210
Access the Orchestrator API Explorer	211
Use the Orchestrator API Explorer to Find Objects	211
Writing Scripts	212
Add Parameters to Scripts	214
Accessing the Orchestrator Server File System from JavaScript and Workflows	215
Accessing Java Classes from JavaScript	215
Accessing Operating System Commands from JavaScript	216

Using XPath Expressions with the vCenter Server Plug-In	216
Using XPath Expressions with the vCenter Server Plug-In	216
Exception Handling Guidelines	217
Orchestrator JavaScript Examples	218
Basic Scripting Examples	219
Email Scripting Examples	221
File System Scripting Examples	222
LDAP Scripting Examples	223
Logging Scripting Examples	223
Networking Scripting Examples	224
Workflow Scripting Examples	224
3 Developing Actions	226
Reusing Actions	226
Access the Actions View	227
Components of the Actions View	227
Creating Actions	227
Create an Action	228
Find Elements That Implement an Action	228
Action Coding Guidelines	229
Use Action Version History	230
Restore Deleted Actions	231
4 Creating Resource Elements	233
View a Resource Element	233
Import an External Object to Use as a Resource Element	234
Edit the Resource Element Information	234
Save a Resource Element to a File	235
Update a Resource Element	236
Add a Resource Element to a Workflow	236
5 Creating Packages	238
Create a Package	238
6 Developing Plug-Ins	240
Overview of Plug-Ins	240
Structure of an Orchestrator Plug-In	241
Exposing an External API to Orchestrator	243
Components of a Plug-In	243
Role of the vso.xml File	245
Roles of the Plug-In Adapter	245

Roles of the Plug-In Factory	246
Role of Finder Objects	247
Role of Scripting Objects	248
Role of Event Handlers	248
Contents and Structure of a Plug-In	249
Defining the Application Mapping in the vso.xml File	250
Format of the vso.xml Plug-In Definition File	251
Naming Plug-In Objects	251
Plug-In Object Naming Conventions	252
File Structure of the Plug-In	253
Orchestrator Plug-In API Reference	254
IAop Interface	254
IDynamicFinder Interface	255
IPluginAdaptor Interface	255
IPluginEventPublisher Interface	256
IPluginFactory Interface	257
IPluginNotificationHandler Interface	258
IPluginPublisher Interface	258
WebConfigurationAdaptor Interface	259
PluginTrigger Class	259
PluginWatcher Class	260
QueryResult Class	261
SDKFinderProperty Class	262
PluginExecutionException Class	263
PluginOperationException Class	263
HasChildrenResult Enumeration	264
ScriptingAttribute Annotation Type	265
ScriptingFunction Annotation Type	265
ScriptingParameter Annotation Type	265
Elements of the vso.xml Plug-In Definition File	265
module Element	266
description Element	266
deprecated Element	267
url Element	267
installation Element	267
action Element	268
finder-datasources Element	268
finder-datasource Element	269
inventory Element	270
finders Element	270
finder Element	271

properties Element	272
property Element	272
relations Element	273
relation Element	273
id Element	274
inventory-children Element	274
relation-link Element	274
events Element	275
trigger Element	275
trigger-properties Element	275
trigger-property Element	276
gauge Element	276
scripting-objects Element	277
object Element	277
constructors Element	278
constructor Element	278
Constructor parameters Element	278
Constructor parameter Element	278
attributes Element	279
attribute Element	279
methods Element	280
method Element	280
example Element	281
code Element	282
Method parameters Element	282
Method parameter Element	282
singleton Element	282
enumerations Element	283
enumeration Element	283
entries Element	284
entry Element	284
Best Practices for Orchestrator Plug-In Development	284
Approaches for Building Orchestrator Plug-Ins	285
Types of Orchestrator Plug-Ins	287
Plug-In Implementation	290
Recommendations for Orchestrator Plug-In Development	295
Documenting Plug-In User Interface Strings and APIs	298

7 Creating Plug-Ins by Using Maven 300

Create an Orchestrator Plug-In with Maven from an Archetype	300
Maven Archetypes	301

[Maven-Based Plug-In Development Best Practices](#) 302

Developing with VMware vRealize Orchestrator

Developing with VMware vRealize Orchestrator provides information and instructions for developing custom VMware® vRealize Orchestrator workflows and actions.

In addition, the documentation contains information about the Orchestrator elements that require scripting and provides JavaScript examples. *Developing with VMware vRealize Orchestrator* also provides instructions about how to create resources and packages.

Intended Audience

This information is intended for developers who want to create custom Orchestrator workflows and actions, as well as custom building blocks.

Note The procedures described in this guide are based on the user interface of the vRealize Orchestrator Legacy Client.

Developing Workflows

1

You develop workflows in the Orchestrator client interface. Workflow development involves using the workflow editor, the built-in Mozilla Rhino JavaScript scripting engine, and the Orchestrator and vCenter Server APIs.

- [Key Concepts of Workflows](#)

Workflows consist of a schema, attributes, and parameters. The workflow schema is the main component of a workflow as it defines all the workflow elements and the logical connections between them. The workflow attributes and parameters are the variables that workflows use to transfer data. Orchestrator saves a workflow token every time a workflow runs, recording the details of that specific run of the workflow.

- [Phases in the Workflow Development Process](#)

The process for developing a workflow involves a series of phases. You can follow a different sequence of phases or skip a phase, depending on the type of workflow that you are developing. For example, you can create a workflow without custom scripting.

- [Best Practices for Developing Workflows](#)

VMware recommends several best practices for developing Orchestrator workflows by multiple users and in a clustered environment.

- [Access Rights for the Orchestrator Client](#)

Only Orchestrator group administrator accounts can access the Java Client.

- [Testing Workflows During Development](#)

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

- [Creating and Editing a Workflow](#)

You create workflows in the Orchestrator client and edit them in the workflow editor. The workflow editor is the IDE of the Orchestrator client for developing workflows.

- [Provide General Workflow Information](#)

You provide a workflow name and description, define attributes and certain aspects of workflow behavior, set the version number, and verify the signature, in the **General** tab in the workflow editor.

- [Defining Attributes and Parameters](#)

After you create a workflow, you must define the global attributes, input parameters, and output parameters of the workflow.

- [Workflow Schema](#)

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

- [Developing Plug-Ins](#)

Orchestrator allows integration with management and administration solutions through its open plug-in architecture. You use the Orchestrator client to run and create plug-in workflows and access the plug-in API.

- [Obtaining Input Parameters from Users When a Workflow Starts](#)

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

- [\(Optional\) Requesting User Interactions While a Workflow Runs](#)

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

- [Calling Workflows Within Workflows](#)

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

- [Running a Workflow on a Selection of Objects](#)

You can automate repetitive tasks by running a workflow on a selection of objects. For example, you can create a workflow that takes a snapshot of all the virtual machines in a virtual machine folder, or you can create a workflow that powers off all the virtual machines on a given host.

- [Developing Long-Running Workflows](#)

A workflow in a waiting state consumes system resources because it constantly polls the object from which it requires a response. If you know that a workflow will potentially wait for a long time before it receives the response it requires, you can add long-running workflow elements to the workflow.

- [Configuration Elements](#)

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

- [Validating Workflows](#)

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

- [Debugging Workflows](#)

Orchestrator provides a workflow debugging tool. You can debug a workflow to inspect the input and output parameters and attributes at the start of any activity, replace parameter or attribute values during a workflow run in edit mode, and resume a workflow from the last failed activity.

- [Running Workflows](#)

An Orchestrator workflow runs according to a logical flow of events.

- [Resuming a Failed Workflow Run](#)

If a workflow fails, Orchestrator provides an option to resume the workflow run from the last failed activity.

- [Generate Workflow Documentation](#)

You can export documentation in PDF format about a workflow or a workflow folder that you select at any time.

- [Use Workflow Version History](#)

You can use version history to revert a workflow to a previously saved state. You can revert the workflow state to an earlier or a later workflow version. You can also compare the differences between the current state of the workflow and a saved version of the workflow.

- [Develop a Simple Example Workflow](#)

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

- [Develop a Complex Workflow](#)

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

Key Concepts of Workflows

Workflows consist of a schema, attributes, and parameters. The workflow schema is the main component of a workflow as it defines all the workflow elements and the logical connections between them. The workflow attributes and parameters are the variables that workflows use to transfer data. Orchestrator saves a workflow token every time a workflow runs, recording the details of that specific run of the workflow.

Workflow Parameters

Workflows receive input parameters and generate output parameters when they run.

Input Parameters

Most workflows require a certain set of input parameters to run. An input parameter is an argument that the workflow processes when it starts. The user, an application, another workflow, or an action passes input parameters to a workflow for the workflow to process when it starts.

For example, if a workflow resets a virtual machine, the workflow requires as an input parameter the name of the virtual machine.

Output Parameters

A workflow's output parameters represent the result from the workflow run. Output parameters can change when a workflow or a workflow element runs. While workflows run, they can receive the output parameters of other workflows as input parameters.

For example, if a workflow creates a snapshot of a virtual machine, the output parameter for the workflow is the resulting snapshot.

Workflow Attributes

Workflow elements process data that they receive as input parameters, and set the resulting data as workflow attributes or output parameters.

Read-only workflow attributes act as global constants for a workflow. Writable attributes act as a workflow's global variables.

You can use attributes to transfer data between the elements of a workflow. You can obtain attributes in the following ways:

- Define attributes when you create a workflow.
- Set the output parameter of a workflow element as a workflow attribute.
- Inherit attributes from a configuration element.

Workflow Schema

A workflow schema is a graphical representation that shows the workflow as a flow diagram of interconnected workflow elements. The workflow schema is the most important element of a workflow as it determines its logic.

Workflow Presentation

When users run a workflow, they provide the values for the input parameters of the workflow in the workflow presentation. When you organize the workflow presentation, consider the type and number of input parameters of the workflow.

Workflow Tokens

A workflow token represents a workflow that is running or has run.

A workflow is an abstract description of a process that defines a generic sequence of steps and a generic set of required input parameters. When you run a workflow with a set of real input parameters, you receive an instance of this abstract workflow that behaves according to the specific input parameters you give it. This specific instance of a completed or a running workflow is called a workflow token.

Workflow Token Attributes

Workflow token attributes are the specific parameters with which a workflow token runs. The workflow token attributes are an aggregation of the workflow's global attributes and the specific input and output parameters with which you run the workflow token.

Phases in the Workflow Development Process

The process for developing a workflow involves a series of phases. You can follow a different sequence of phases or skip a phase, depending on the type of workflow that you are developing. For example, you can create a workflow without custom scripting.

Generally, you develop a workflow through the following phases.

- 1 Create a new workflow or create a duplicate of an existing workflow from the standard library.
- 2 Provide general information about the workflow.
- 3 Define the input parameters of the workflow.
- 4 Lay out and link the workflow schema to define the logical flow of the workflow.
- 5 Bind the input and output parameters of each schema element to workflow attributes.
- 6 Write the necessary scripts for scriptable task elements or custom decision elements.
- 7 Create the workflow presentation to define the layout of the input parameters dialog box that the users see when they run the workflow.
- 8 Validate the workflow.

Best Practices for Developing Workflows

VMware recommends several best practices for developing Orchestrator workflows by multiple users and in a clustered environment.

- Each developer has a dedicated test standalone Orchestrator instance for creating and developing workflows.
- Workflows are saved as maven projects on a shared source code control system.
- To ensure optimal performance of the Orchestrator production deployment, it is best to import workflows in a scheduled window.

- When importing workflows to an Orchestrator cluster, connect the Orchestrator client to one of the nodes by using their local host name or IP address, instead of the address of the load balancer virtual server.

Note Any modifications of a workflow take effect with the next workflow run.

Access Rights for the Orchestrator Client

Only Orchestrator group administrator accounts can access the Java Client.

Testing Workflows During Development

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

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

Note Do not forget to reactivate automatic validation when you finish developing the workflow.

Procedure

- 1 In the Orchestrator client menu, click **Tools > User preferences**.
- 2 Click the **Workflows** tab.
- 3 Deselect the **Validate workflow before running it** check box.

Results

You deactivated automatic workflow validation.

Creating and Editing a Workflow

You create workflows in the Orchestrator client and edit them in the workflow editor. The workflow editor is the IDE of the Orchestrator client for developing workflows.

You open the workflow editor by editing an existing workflow.

- [Create a Workflow](#)

You can create workflows in the workflows hierarchical list of the Orchestrator client.

- [Edit a Workflow](#)

You edit a workflow to make changes to an existing workflow or to develop a new empty workflow.

■ [Edit a Workflow from the Standard Library](#)

Orchestrator provides a standard library of workflows that you can use to automate operations in the virtual infrastructure. The workflows in the standard library are locked in the read-only state.

■ [Workflow Editor Tabs](#)

The workflow editor consists of tabs on which you edit the components of the workflows.

Create a Workflow

You can create workflows in the workflows hierarchical list of the Orchestrator client.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Workflows** view.
- 3 (Optional) Right-click the root of the workflows hierarchical list, or a folder in the list, and select **Add folder** to create a new workflow folder.
- 4 (Optional) Type the name of the new folder.
- 5 Right-click the new folder or an existing folder and select **New workflow**.
- 6 Name the new workflow and click **OK**.

Results

A new empty workflow is created in the folder that you chose.

What to do next

You can edit the workflow.

Edit a Workflow

You edit a workflow to make changes to an existing workflow or to develop a new empty workflow.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Workflows** view.
- 3 Expand the workflows hierarchical list to navigate to the workflow that you want to edit.
- 4 To open the workflow for editing, right-click the workflow and select **Edit**.

Results

The workflow editor opens the workflow for editing.

Edit a Workflow from the Standard Library

Orchestrator provides a standard library of workflows that you can use to automate operations in the virtual infrastructure. The workflows in the standard library are locked in the read-only state.

To edit a workflow from the standard library, you must create a duplicate of that workflow. You can edit duplicate workflows or custom workflows.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Workflows** view.
- 3 (Optional) Right-click the root of the hierarchical list of workflow folders and select **New folder** to create a folder to contain the workflow to edit.
- 4 Expand the **Library** hierarchical list of standard workflows to navigate to the workflow to edit.
- 5 Right-click the workflow to edit.

The **Edit** option is dimmed. The workflow is read-only.

- 6 Right-click the workflow and select **Duplicate workflow**.
- 7 Provide a name for the duplicate workflow.
By default, Orchestrator names the duplicate workflow *Copy of workflow_name*.
- 8 Click the **Workflow folder** value to search for a folder in which to save the duplicate workflow.

Select the folder you created in [Step 3](#). If you did not create a folder, select a folder that is not in the library of standard workflows.

- 9 Click **Yes** or **No** to copy the workflow version history to the duplicate.

Option	Description
Yes	The version history of the original workflow is replicated in the duplicate.
No	The version of the duplicate reverts to 0.0.0.

- 10 Click **Duplicate** to duplicate the workflow.
- 11 Right-click the duplicate workflow and select **Edit**.
The workflow editor opens. You can edit the duplicate workflow.

Results

You duplicated a workflow from the standard library. You can edit the duplicate workflow.

Workflow Editor Tabs

The workflow editor consists of tabs on which you edit the components of the workflows.

Table 1-1. Workflow Editor Tabs

Tab	Description
Summary	Provide general information about the workflow, such as the workflow name, description, tags, and version number. The Summary tab can also be used to configure the server restart behavior and workflow run failure behavior.
Variables	Define the variables for your workflow. Variables were previously known as attributes.
Input/Output	Define the input and output parameters of your workflow. The input parameters provide the data that the workflow processes during the workflow run. The output parameters are provided when the workflow run finishes.
Schema	Build the workflow. You build the workflow by dragging workflow schema elements from the workflow palette on the left side of the Schema tab. By clicking an element in the schema diagram, you can define and edit the element's behavior.
Input Form	Define the layout of the user input dialog box that appears when users run a workflow. You arrange the parameters and variables into presentation steps and tabs to ease identification of parameters in the input parameters dialog box. You define the constraints on the input parameters that users can provide in the presentation by setting the parameter properties. You can also add external validation for your workflow by using actions. For more information on the input form designer, see <i>vRealize Orchestrator Input Form Designer</i> in <i>Using the VMware vRealize Orchestrator Client</i> .
Version History	View and manage the version history of the workflow. Compare and restore versions and push and pull workflows to and from your integrated Git repository. For more information on using Git in the vRealize Orchestrator Client, see <i>How Can I Use Git Branching to Manage My vRealize Orchestrator Object Inventory</i> in <i>Using the VMware vRealize Orchestrator Client</i> .
Audit	View information about events related to the workflow such as, when it was saved, when a workflow run was performed, and when a workflow run was finished.

Provide General Workflow Information

You provide a workflow name and description, define attributes and certain aspects of workflow behavior, set the version number, and verify the signature, in the **General** tab in the workflow editor.

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

- 1 Click the **General** tab in the workflow editor.
- 2 Click the **Version** digits to set a version number for the workflow.

The **Version Comment** dialog box opens.

- 3 Type a comment for this version of the workflow and click **OK**.

For example, type **Initial creation** if you just created the workflow.

A new version of the workflow is created. You can later revert the state of the workflow to this version.

- 4 Define how the workflow behaves if the Orchestrator server restarts by setting the **Server restart behavior** value.

- Leave the default value of **Resume workflow run** to make the workflow resume at the point at which its run was interrupted when the server stopped.
- Click **Resume workflow run** and select **Do not resume workflow run (set as FAILED)** to prevent the workflow from restarting if the Orchestrator server restarts.

Prevent the workflow from restarting if the workflow depends on the environment in which it runs. For example, if a workflow requires a specific vCenter Server and you reconfigure Orchestrator to connect to a different vCenter Server, restarting the workflow after you restart the Orchestrator server causes the workflow to fail.

- 5 Type a detailed description of the workflow in the **Description** text box.
- 6 Click **Save** at the bottom of the workflow editor.

A green message at the bottom left of the workflow editor confirms that you saved your changes.

Results

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

What to do next

You must define the workflow attributes and parameters.

Defining Attributes and Parameters

After you create a workflow, you must define the global attributes, input parameters, and output parameters of the workflow.

Workflow attributes store data that workflows process internally. Workflow input parameters are data provided by an outside source, such as a user or another workflow. Workflow output parameters are data that the workflow delivers when it finishes its run.

- [Define Workflow Parameters](#)

You can use input and output parameters to pass data into and out of the workflow.

- [Define Workflow Attributes](#)

Workflow attributes are the data that workflows process.

- [Attribute and Parameter Naming Restrictions](#)

You can use OGNL expressions to determine input parameters dynamically when a workflow runs. The Orchestrator OGNL parser uses certain keywords during OGNL processing that you cannot use in workflow attribute or parameter names.

Define Workflow Parameters

You can use input and output parameters to pass data into and out of the workflow.

You can define the parameters of a workflow in the workflow editor. The input parameters are the initial data that the workflow requires to run. Users provide the values for the input parameters when they run the workflow. The output parameters are the data the workflow returns when it completes its run.

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

1 Click the appropriate tab in the workflow editor.

- Click **Inputs** to create input parameters.
- Click **Outputs** to create output parameters.

2 Right-click inside the parameters tab and select **Add parameter**.

3 Click the parameter name to change it.

The default name is `arg_in_X` for input parameters and `arg_out_X` for output parameters, where `X` is a number.

4 (Optional) To change the value of the parameter type, click the value and select one from the list of available values.

The value for the parameter type is String by default.

5 Add a description for the parameter in the **Description** text box.

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

Results

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

What to do next

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

Define Workflow Attributes

Workflow attributes are the data that workflows process.

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

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

- 1 Click the **General** tab in the workflow editor.

The attributes pane appears in the bottom half of the **General** tab.

- 2 Right-click in the attributes pane and select **Add Attribute**.

A new attribute appears in the attributes list, with String as its default type.

- 3 Click the attribute name to change it.

The default name is attX, where X is a number.

Note Workflow attributes must not have the same name as any of the workflow's parameters.

- 4 Click the attribute type to select a new type from a list of possible values.

The default attribute type is String.

- 5 Click the attribute value to set or select a value according to the attribute type.

- 6 Add a description of the attribute in the **Description** text box.

- 7 If the attribute is a constant rather than a variable, click the check box to the left of the attribute name to make its value read-only.

The lock icon identifies the column of read-only check boxes.

- 8 (Optional) If you decide that the attribute should be an input or output parameter rather than an attribute, right-click the attribute and select **Move as INPUT/OUTPUT parameter** to change the attribute into a parameter.

Results

You defined an attribute for the workflow.

What to do next

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

Attribute and Parameter Naming Restrictions

You can use OGNL expressions to determine input parameters dynamically when a workflow runs. The Orchestrator OGNL parser uses certain keywords during OGNL processing that you cannot use in workflow attribute or parameter names.

Using a reserved OGNL keyword as a prefix to an attribute name does not break OGNL processing. For example, you can name a parameter `trueParameter`. Reserved keywords are not case-sensitive.

You cannot use the following keywords in workflow attribute and parameter names.

Table 1-2. Forbidden Keywords in Attribute and Parameter Names

Forbidden Keyword	Forbidden Keyword	Forbidden Keyword
■ abstract	■ eof	■ _memberAccess
■ back_char_esc	■ esc	■ native
■ back_char_literal	■ exponent	■ package
■ boolean	■ export	■ private
■ byte	■ extends	■ public
■ char	■ false	■ root
■ char_literal	■ final	■ short
■ class	■ flt_literal	■ static
■ _classResolver	■ flt_suff	■ string_esc
■ const	■ ident	■ string_literal
■ context	■ implements	■ synchronized
■ debugger	■ import	■ this
■ dec_digits	■ in	■ _traceEvaluations
■ dec_flt	■ int	■ true
■ default	■ int_literal	■ _typeConverter
■ delete	■ interface	■ volatil
■ digit	■ _keepLastEvaluation	■ with
■ double	■ _lastEvaluation	■ WithinBackCharLiteral
■ dynamic_subscript	■ letter	■ WithinCharLiteral
■ enum	■ long	■ WithinStringLiteral

Workflow Schema

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

- [View Workflow Schema](#)

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

- [Building a Workflow in the Workflow Schema](#)

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

- [Schema Elements](#)

The workflow editor presents the workflow schema elements in menus on the **Schema** tab. You can use the schema elements available in the **Schema** tab to build a workflow.

- [Schema Element Properties](#)

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

- [Links and Bindings](#)

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

- [Decisions](#)

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

- [Exception Handling](#)

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

- [Using Error Handlers](#)

You can use a standard error handler to define the behavior in case an error occurs in a specific workflow schema element. You can use a global error handler to define the behavior in case errors that are not caught by standard error handlers occur.

- [Foreach Elements and Composite Types](#)

You can insert a Foreach element in the workflow that you develop to run a subworkflow that iterates over arrays of parameters or attributes. To improve the understanding and readability of the workflow, you can group several workflow parameters of different types that are logically connected in a single type that is called a composite type.

■ [Add a Switch Activity to a Workflow](#)

You can add a basic switch activity to a workflow schema that defines the switch cases based on workflow attributes or parameters.

View Workflow Schema

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

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Navigate to a workflow in the workflow hierarchical list.
- 3 Click the workflow.
Information about that workflow appears in the right pane.
- 4 Select the **Schema** tab in the right pane.

Results

You see the graphical representation of the workflow.

Building a Workflow in the Workflow Schema

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

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

Edit a Workflow Schema

You build a workflow by creating a sequence of schema elements that define the logical flow of the workflow.

By default, all elements in the workflow schema are linked. Links between the elements are represented as arrows. When you add a new element to the workflow schema, you must drag it onto an arrow or an existing workflow element that is not linked to a next element. After you add workflow elements to the schema, you can delete existing links and create new links to define the logical flow of the workflow.

You can copy an element or a selection of elements from the schema of an existing workflow to the schema of the workflow that you are editing. See [Copy Workflow Schema Elements](#).

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

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

- 1 Click the **Schema** tab in the workflow editor.
- 2 Drag a schema element from the **Generic** menu in the left pane, to the workflow schema.
- 3 Double-click the element you dragged to the workflow schema, type an appropriate name, and press Enter.

You must provide elements with unique names in the context of the workflow.

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

- 4 (Optional) Right-click an element in the schema and select **Copy**.
- 5 (Optional) Right-click at an appropriate position in the schema and select **Paste**.

Copying and pasting existing schema elements is a quick way of adding similar elements to the schema. All of the settings of the copied element appear in the pasted element, except for the business state. Adjust the pasted element settings accordingly.

- 6 Drag schema elements from the **Basic**, **Log**, or **Network** menus to the workflow schema.

You can edit the names of the elements in the **Basic**, **Log**, or **Network** menus. You cannot edit their scripting.

- 7 Drag schema elements from the **Generic** menu to the workflow schema.

When you drag actions or workflows to the workflow schema, a dialog box in which you can search for the action or workflow to insert appears.

- 8 In the **Filter** text box, type the name or part of the name of the workflow or action to insert in the workflow.

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

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

You inserted the workflow or action in the workflow schema.

- 10 Repeat this procedure until you have added all of the required schema elements to the workflow schema.

What to do next

Define the properties of the elements you added to the workflow schema and link and bind them all together.

Copy Workflow Schema Elements

You can copy an element or a selection of elements from the schema of an existing workflow to the schema of the workflow that you are editing.

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

- 1 Click the **Schema** tab in the workflow editor.
- 2 From the left pane, select the workflow from which you want to copy schema elements.
 - Click **All Workflows** and select the workflow from the hierarchical list of workflows.
 - Type the name of the workflow in the search text box and press Enter.
- 3 Right-click the selected workflow and select **Open**.
A window displaying the workflow's properties appears.
- 4 In the workflow's window, click the **Schema** tab.
- 5 Select one or more workflow schema elements, right-click the selection, and select **Copy**.
- 6 In the **Schema** tab of the workflow that you are editing, right-click and select **Paste**.

Results

You copied workflow schema elements from one workflow to another.

What to do next

You must link and bind the copied schema elements to the existing workflow schema.

Promote Input and Output Parameters

You can promote the input and output parameters of a child element to the parent workflow.

You can promote a custom attribute that you have defined on the **General** tab of the workflow editor. You can promote predefined attributes only by replacing an input parameter with an attribute of matching type.

Note If you promote a predefined attribute and assign a custom value to it, a duplicate attribute is created to avoid overwriting the value of the original attribute. The duplicate attribute retains the name of the original attribute and increments the numerical value at the end of the attribute's name.

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

- 1 Click the **Schema** tab in the workflow editor.
- 2 Add a workflow or an action element to the workflow schema.
The following notification appears at the top of the schema pane.

Do you want to add the activity's parameters as input/output to the current workflow?

- 3 On the notification, click **Setup**.

A pop-up window with the available options appears.

- 4 Select the mapping type for each input parameter.

Option	Description
Input	The argument is mapped to an input workflow parameter.
Skip	The argument is mapped to a NULL value.
Value	The argument is mapped to an attribute with a value that you can set from the Value column.

- 5 Select the mapping type for each output parameter.

Option	Description
Output	The argument is mapped to an output workflow parameter.
Skip	The argument is mapped to a NULL value.
Local variable	The argument is mapped to an attribute.

- 6 Click **Promote**.

Results

You promoted parameters to the parent workflow.

Modify Search Results

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

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

Procedure

- 1 (Optional) If you are editing a workflow in the workflow editor, click **Save and Close** to exit the editor.
- 2 From the Orchestrator client menu, select **Tools > User preferences**.
- 3 Click the **General** tab.
- 4 Type the number of results for searches to return in the **Finder Maximum Size** text box.
- 5 Click **Save and Close** in the User Preferences dialog box.

Results

You modified the number of results that searches return.

Schema Elements

The workflow editor presents the workflow schema elements in menus on the **Schema** tab. You can use the schema elements available in the **Schema** tab to build a workflow.

Table 1-3. Schema Elements and Icons






Schema Element Name	Description	Icon	Location in Workflow Editor
Start Workflow	The starting point of the workflow. All workflows contain this element. A workflow can have only one start element. Start elements have one output and no input, and cannot be removed from the workflow schema.		Always present on the Schema tab
Scriptable task	General-purpose tasks you define. You write JavaScript functions in this element.		The Generic workflow palette
Decision	A boolean function. Decision elements take one input parameter and return either <code>true</code> or <code>false</code> . The type of decision that the element makes depends on the type of the input parameter. Decision elements let the workflow branch into different directions, depending on the input parameter the decision element receives. If the received input parameter corresponds to an expected value, the workflow continues along a certain route. If the input is not the expected value, the workflow continues on an alternative path.		The Generic workflow palette
Custom decision	A boolean function. Custom decisions can take several input parameters and process them according to custom scripts. Returns either <code>true</code> or <code>false</code> .		The Generic workflow palette
Decision activity	A boolean function. A decision activity runs a workflow and binds its output parameters to a <code>true</code> or a <code>false</code> path.		The Generic workflow palette

Table 1-3. Schema Elements and Icons (continued)






Schema Element Name	Description	Icon	Location in Workflow Editor
User interaction	Lets users pass new input parameters to the workflow. You can design how the user interaction element presents the request for input parameters and place constraints on the parameters that users can provide. When a running workflow arrives at a user interaction element, it enters a passive state and prompts the user for input. You can set a timeout period within which the users must provide input. The workflow resumes according to the data the user passes to it, or returns an exception if the timeout period expires. While it is waiting for the user to respond, the workflow token is in the waiting state.		The Generic workflow palette
Waiting timer	Used by long-running workflows. When a running workflow arrives at a Waiting Timer element, it enters a passive state. You set an absolute date at which the workflow resumes running. While it is waiting for the date, the workflow token is in the waiting-signal state.		The Generic workflow palette
Waiting event	Used in long-running workflows. When a running workflow arrives at a Waiting Event element, it enters a passive state. You define a trigger event that the workflow awaits before it resumes running. While it is waiting for the event, the workflow token is in the waiting-signal state.		The Generic workflow palette
End workflow	The end point of a workflow. You can have multiple end elements in a schema, to represent the various possible outcomes of the workflow. End elements have one input with no output. When a workflow reaches an End Workflow element, the workflow token enters the completed state.		The Generic workflow palette
Thrown exception	Creates an exception and stops the workflow. Multiple occurrences of this element can be present in the workflow schema. Exception elements have one input parameter, which can only be of the String type, and have no output parameter. When a workflow reaches an Exception element, the workflow token enters the failed state.		The Generic workflow palette

Table 1-3. Schema Elements and Icons (continued)











Schema Element Name	Description	Icon	Location in Workflow Editor
Workflow note	Lets you annotate sections of the workflow. You can stretch notes to delineate sections of the workflow. You can change the background color of the notes to differentiate workflow zones. Workflow notes provide only visual information, to help you understand the schema.		The Generic workflow palette
Action element	Calls on an action from the Orchestrator libraries of actions. When a workflow reaches an action element, it calls and runs that action.		The Generic workflow palette
Workflow element	Starts another workflow synchronously. When a workflow reaches a Workflow element in its schema, it runs that workflow as part of its own process. The original workflow continues only after the called workflow completes its run.		The Generic workflow palette
Foreach element	Runs a workflow on every element from an array. For example, you can run the Rename Virtual Machine workflow on all virtual machines from a folder.		The Generic workflow palette
Asynchronous workflow	Starts a workflow asynchronously. When a workflow reaches an asynchronous workflow element, it starts that workflow and continues its own run. The original workflow does not wait for the called workflow to complete.		The Generic workflow palette
Schedule workflow	Creates a task to run the workflow at a set time, and then the workflow continues its run.		The Generic workflow palette
Nested workflows	Starts several workflows simultaneously. You can decide to nest local workflows and remote workflows that are in a different Orchestrator server. You can also run workflows with different credentials. The workflow waits for all the nested workflows to complete before continuing its run.		The Generic workflow palette
Handle error	Handles an error for a specific workflow element. The workflow can handle the error by creating an exception, calling another workflow, or running a custom script.		The Generic workflow palette
Default error handler	Handles workflow errors that are not caught by standard error handlers. You can use any available schema elements to handle errors.		The Generic workflow palette

Table 1-3. Schema Elements and Icons (continued)

Schema Element Name	Description	Icon	Location in Workflow Editor
Switch	Switches to alternative workflow paths, based on a workflow attribute or parameter.		The Generic workflow palette
Pre-Defined Task	<p>Non-editable scripted elements that perform standard tasks that workflows commonly use. The following tasks are predefined:</p> <p>Basic</p> <ul style="list-style-type: none"> ■ Sleep ■ Change credential ■ Wait until date ■ Wait for custom event ■ Send custom event ■ Increase counter ■ Decrease counter <p>Log</p> <ul style="list-style-type: none"> ■ System log ■ System warning ■ System error ■ Server log ■ Server warning ■ Server error ■ System+Server log ■ System+Server warning ■ System+Server error <p>Network</p> <ul style="list-style-type: none"> ■ HTTP post ■ HTTP get 		The Basic , Log , and Network workflow palettes

Schema Element Properties

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

Edit the Global Properties of a Schema Element

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

Prerequisites

Verify that the **Schema** tab of the workflow editor contains elements.

Procedure

- 1 Click the **Schema** tab in the workflow editor.

- 2 Select an element to edit by clicking the **Edit** icon (✎).

A dialog box that lists the properties of the element appears.

- 3 Click the **Info** tab.

- 4 Provide a name for the schema element in the **Name** text box.

This is the name that appears in the schema element in the workflow schema diagram.

- 5 From the **Interaction** drop-down menu, select a description.

The **Interaction** property allows you to select between standard descriptions of how this element interacts with objects outside of the workflow. This property is for information only.

- 6 (Optional) Provide a business status description in the **Business Status** text box.

The **Business Status** property is a brief description of what this element does. When a workflow is running, the workflow token shows the Business Status of each element as it runs. This feature is useful for tracking workflow status.

- 7 (Optional) In the **Description** text box, type a description of the schema element.

Schema Element Properties Tabs

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

Different schema elements have different properties tabs.

Table 1-4. Properties Tabs per Schema Element

Schema Element Property Tab	Description	Applies to Schema Element Type
Attributes	Attributes that elements require from an external source, such as the user, an event, or a timer. The attributes can be a timeout limit, a time and date, a trigger, or user credentials.	<ul style="list-style-type: none"> ■ User Interaction ■ Waiting Event ■ Waiting Timer
Decision	Defines the decision statement. The input parameter that the decision element receives either matches or does not match the decision statement, resulting two possible courses of action.	Decision
End Workflow	Stops the workflow, either because the workflow completed successfully, or because it encountered an error and returned an exception.	<ul style="list-style-type: none"> ■ End ■ Exception

Table 1-4. Properties Tabs per Schema Element (continued)

Schema Element Property Tab	Description	Applies to Schema Element Type
Exception	How this schema element behaves in the event of an exception.	<ul style="list-style-type: none"> ■ Action ■ Asynchronous Workflow ■ Exception ■ Nested Workflows ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ User Interaction ■ Waiting Event ■ Waiting Timer ■ Workflow
External Inputs	Input parameters that the user must provide at a certain moment while the workflow runs.	User Interaction
IN	The IN binding for this element. The IN binding defines the way in which the schema element receives input from the element that precedes it in the workflow.	<ul style="list-style-type: none"> ■ Action ■ Asynchronous Workflow ■ Custom Decision ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ Workflow
Info	The schema element's general properties and description. The information the Info tab displays depends on the type of schema element.	<ul style="list-style-type: none"> ■ Action ■ Asynchronous Workflow ■ Custom Decision ■ Decision ■ Nested Workflows ■ Note ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ User Interaction ■ Waiting Event ■ Waiting Timer ■ Workflow
OUT	The OUT binding for this element. The OUT binding defines the way in which the schema element binds output parameters to the workflow attributes or to the workflow output parameters.	<ul style="list-style-type: none"> ■ Action ■ Asynchronous Workflow ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ Workflow

Table 1-4. Properties Tabs per Schema Element (continued)

Schema Element Property Tab	Description	Applies to Schema Element Type
Presentation	Defines the layout of the input parameters dialog box the user sees if the workflow needs user input while it is running.	User Interaction
Scripting	Shows the JavaScript function that defines the behavior of this schema element. For Asynchronous Workflow, Schedule Workflow, and Action elements this scripting is read-only. For scriptable task and custom decision elements, you edit the JavaScript in this tab.	<ul style="list-style-type: none"> ■ Action ■ Asynchronous Workflow ■ Custom Decision ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task
Visual Binding	Shows a graphical representation of how the parameters and attributes of this schema element bind to the parameters and attributes of the elements that come before and after it in the workflow. This is another representation of the element's IN and OUT bindings.	<ul style="list-style-type: none"> ■ Action ■ Asynchronous Workflow ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ Workflow
Workflows	Selects the workflows to nest.	Nested Workflows

Links and Bindings

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

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

Logical Flow of a Workflow

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

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

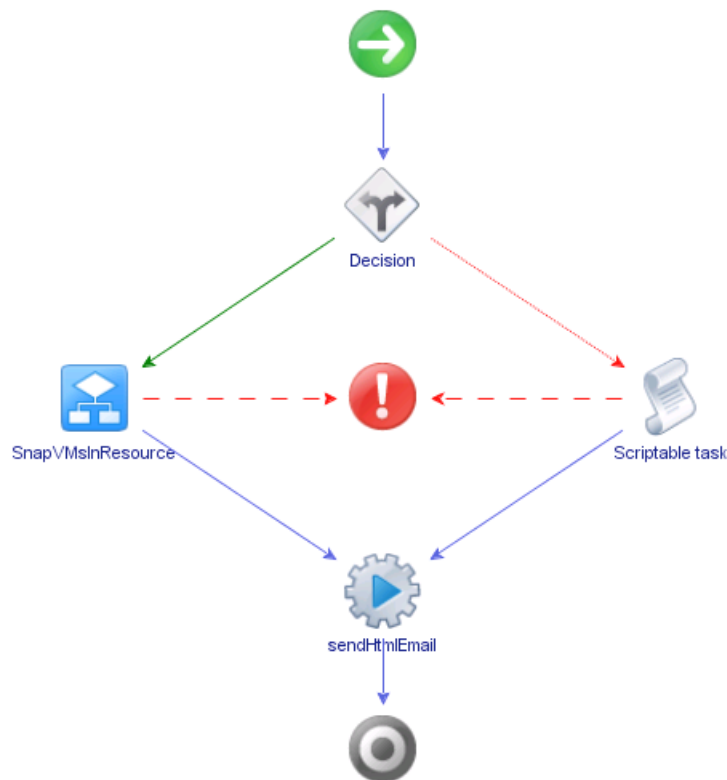
Different styles of arrows in the workflow schema denote the different paths that the workflow can take through its logical flow.

- A blue arrow denotes the standard path that the workflow takes from one element to the next.

- A green arrow denotes the path that the workflow takes if a Boolean decision element returns true.
- A red dotted arrow denotes the path that the workflow takes if a Boolean decision element returns false.
- A red dashed arrow denotes the exception path that the workflow takes if a workflow element does not run correctly.

The following figure shows an example workflow schema that demonstrates the different paths that workflows can take.

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



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

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

- c The `sendHtmlEmail` action runs successfully.
 - d The workflow ends successfully in the `completed` state.
- `true` decision result, exception.
 - a The decision element returns `true`.
 - b The `SnapVMsInResourcePool` workflow encounters an error.
 - c The workflow returns an exception and stops in the `failed` state.
- `false` decision result, exception.
 - a The decision element returns `false`.
 - b The operation the Scriptable task element defines encounters an error.
 - c The workflow returns an exception and stops in the `failed` state.

Element Links

Links connect schema elements and define the logical flow of the workflow from one element to the next.

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

The following elements are exceptions to the preceding statements.

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

Create Standard Path Links

Standard path links determine the normal run of the workflow.

When you link one element to another, you always link the elements in the order in which they run in the workflow. You always start from the element that runs first to create a link between two elements.

Prerequisites

- Open a workflow for editing in the workflow editor.
- Verify that the **Schema** tab of the workflow editor contains elements.

Procedure

- 1** Place the pointer on the element that you want to connect to another element.
A blue and a red arrow appear on the element's right.
- 2** Place the pointer on the blue arrow.
The blue arrow enlarges.
- 3** Left-click the blue arrow, hold down the left mouse button, and move the pointer to the target element.
A blue arrow appears between the two elements and a green rectangle appears around the target element.
- 4** Release the left mouse button.
The blue arrow remains between the two elements.

Results

A standard path now links the elements.

What to do next

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

Data Flow of a Workflow

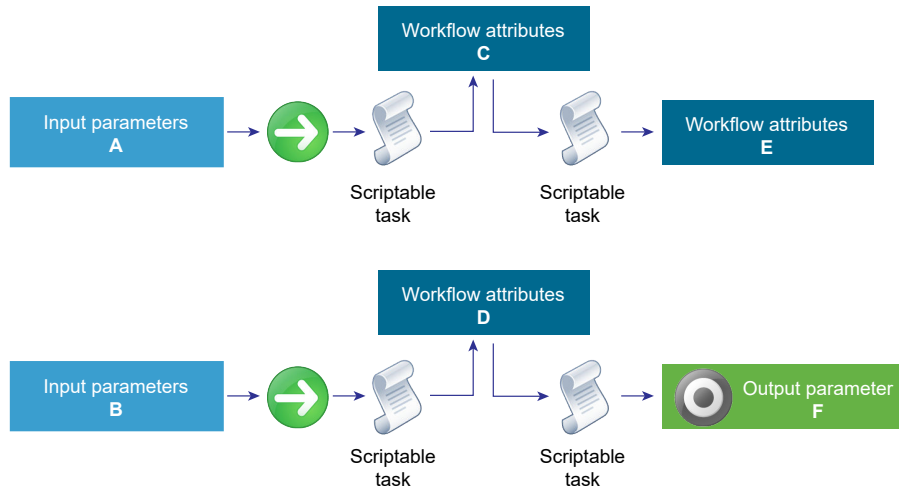
The data flow of a workflow is the manner in which workflow element input and output parameters bind to workflow attributes as each element of the workflow runs. You define the data flow of a workflow by using schema element bindings.

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

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

The following figure shows a very simple workflow. The blue arrows represent the element linking and the logical flow of the workflow. The red lines show the data flow of the workflow.

Figure 1-2. Example of Workflow Data Flow



The data flows through the workflow as follows.

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

Element Bindings

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

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

IN bindings

Set a schema element's incoming data. You bind the element's local input parameters to source workflow attributes. The **IN** tab lists the element's input parameters in the Local Parameter column. The **IN** tab lists the workflow attributes to which the local parameter binds

in the Source Parameter column. The tab also displays the parameter type and a description of the parameter.

OUT bindings

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

Exception bindings

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

IN bindings read values from the bound source parameter. OUT bindings write values into the bound source parameter.

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

A common mistake when creating workflows is to not bind output parameter values to reflect the changes that the element makes to the workflow attributes.

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

Define Element Bindings

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

Prerequisites

Verify that you have a workflow schema in the **Schema** tab of the workflow editor, and that you have created links between the elements.

Procedure

- 1 Click the **Edit** icon (✎) of the element on which to set the bindings.

A dialog box that lists the properties of the element appears.

2 Click the **IN** tab.

The contents of the **IN** tab depend on the type of element you selected.

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

3 If an appropriate parameter exists, choose an input parameter to bind, and click the **Not set** button in the **Source Parameter** text box.

A list of possible source parameters and attributes to bind to appears.

4 Choose a source parameter to bind to the local input parameter from the list proposed.**5** (Optional) If you have not defined the source parameter to which to bind, you can create it by clicking the **Create parameter/attribute in workflow** link in the parameter selection dialog box.**6** Click the **OUT** tab.

The contents of the **OUT** tab depend on the type of element you selected.

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

7 Choose a parameter to bind.**8** Click the **Source Parameter > Not set** button.**9** Choose a source parameter to bind to the input parameter.**10** (Optional) If you did not define the parameter to which to bind, you can create it by clicking the **Create parameter/attribute in workflow** button in the parameter selection dialog box.**Results**

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

What to do next

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

Decisions

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

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

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

Custom Decisions

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

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

Create Decision Element Links

Decision elements differ from other elements in a workflow. They have only `true` or `false` output parameters. Decision elements have no exception linking.

Prerequisites

Verify that the **Schema** tab of the workflow editor contains elements, including at least one decision element that is not linked to other elements.

Procedure

- 1 Place the mouse pointer on a decision element to link it to two other elements that define two possible branches in the workflow.

A blue arrow and a red arrow appear on the element's right.

- 2 Place the pointer on the blue arrow, and while keeping the left mouse button pressed, move the pointer to the target element.

A green arrow appears between the two elements and the target element turns green. The green arrow represents the **true** path the workflow takes if the input parameter or attribute received by the decision element matches the decision statement.

- 3 Release the left mouse button.

The green arrow remains between the two elements. You have defined the path the workflow takes when the decision element receives the expected value.

- 4 Place the pointer on the decision element, hold down the left mouse button, and move the pointer to the target element.

A dotted red arrow appears between the two elements and the target element turns green. The red arrow represents the **false** path that the workflow takes if the input parameter or attribute received by the decision element does not match the decision statement.

- 5 Release the left mouse button.

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

Results

You have defined the possible **true** or **false** paths that the workflow takes depending on the input parameter or attribute the decision element receives.

What to do next

Define the decision statement. See [Create Workflow Branches Using Decisions](#).

Delete a Linked Decision Element

When you delete a linked decision element from a workflow schema, you must specify which workflow paths to delete.

Prerequisites

Verify that the **Schema** tab of the workflow editor contains elements, including at least one decision element with **true** and **false** paths.

Procedure

- 1 Select the decision element and press Delete.

A dialog box with available options appears.

- 2 Select which decision branch to delete.

Option	Description
Success branch	The decision element and all elements that follow the true decision path are deleted from the workflow schema.
Failure branch	The decision element and all elements that follow the false decision path are deleted from the workflow schema.
Both branches	The decision element and all elements that follow both decision paths are deleted from the workflow schema.
None	Only the decision element and its links are deleted from the workflow schema. All elements that follow both decision paths remain in the workflow schema.

- 3 Click **OK**.


Create Workflow Branches Using Decisions

Decision elements are simple Boolean functions that you use to create branches in workflows. Decision elements determine whether the input received matches the decision statement you set. As a function of this decision, the workflow continues its course along one of two possible paths.

Prerequisites

Verify that you have a decision element linked to two other elements in the schema in the workflow editor before you define the decision.

Procedure

- 1 Click the **Edit** icon () of the decision element.
A dialog box that lists the properties of the decision element appears.
- 2 Click the **Decision** tab in the dialog box.
- 3 Click the **Not Set (NULL)** link to select the source input parameter for this decision.
A dialog box that lists all the attributes and input parameters defined in this workflow appears.
- 4 Select an input parameter from the list by double-clicking it.
- 5 If you did not define the source parameter to which to bind, create it by clicking the **Create attribute/parameter in workflow** link in the parameter selection dialog box.
- 6 Select a decision statement from the drop-down menu.
The statements that the menu proposes are contextual, and differ according to the type of input parameter selected.

- 7 Add a value that you want the decision statement to match.

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

Results

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

What to do next

You must set how the workflow handles exceptions.

Exception Handling

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

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

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

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

Create Exception Bindings

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

Prerequisites

Verify that the **Schema** tab of the workflow editor contains elements.



Procedure


- 1 Place the pointer on the element for which you want to define exception binding.

A red arrow appears on the element's right.

- 2 Place the pointer on the red arrow until it enlarges, hold down the left mouse button, and drag the red arrow to the target element.

A red dashed arrow links the two elements. The target element defines the behavior of the workflow if the element that links to it encounters an error.

- 3 Click the **Edit** icon () of the element that links to the exception handling element.
- 4 Click the **Exception** tab in the schema element properties tabs.
- 5 To set the **Output exception binding** value, click **Not set**.
 - Select a parameter to bind to the exception output parameter from the exception attribute binding dialog box and click **Select**.
 - Click **Create parameter/attribute in workflow** to create an exception output parameter.
- 6 Click the target element that defines the exception handling behavior.
- 7 Click the **IN** tab in the schema element properties tabs.
- 8 Click the **Bind to workflow parameter/attribute** icon ().

The dialog box for selecting the input parameter appears.
- 9 Select the exception output parameter and click **Select**.
- 10 Click the **OUT** tab for the exception handling element in the schema element properties tabs.
- 11 Define the behavior of the exception handling element.
 - Click the **Bind to workflow parameter/attribute** icon () to select an output parameter for the exception handling element to generate.
 - Click the **Scripting** tab and use JavaScript to define the behavior of the exception handling element.

Results

You defined how the element handles exceptions.

What to do next

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

Using Error Handlers

You can use a standard error handler to define the behavior in case an error occurs in a specific workflow schema element. You can use a global error handler to define the behavior in case errors that are not caught by standard error handlers occur.

Add an Error Handler to a Workflow

You can define how errors in a specific workflow element are handled during a workflow run by adding an error handler to the workflow element. You can add an error handler only to workflow elements that do not have a specified error path.

Important Workflows that contain a **Handle error** element are not compatible with Orchestrator 5.5.x or earlier.

Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.
- Add some elements to the workflow schema.

Procedure

- 1 Drag a **Handle error** element to the appropriate element in the workflow schema.
A dialog box appears.
- 2 From the drop-down menu in the dialog box, select how errors should be handled.

Option	Description
Throw exception	When an error occurs, an exception is thrown. You can modify the exception binding.
Call a workflow	When an error occurs, a selected workflow runs.
Custom script	When an error occurs, a custom script runs.

- 3 Click **Select**.

Results

You added an error handler to a workflow. When the workflow reaches this element, it performs the selected action before ending its run.

Add a Global Error Handler to a Workflow

You can define how errors, which are not caught by standard error handlers, are handled during a workflow run by adding a global error handler to the workflow schema. You can add one global error handler to a workflow schema.

Important Workflows that contain a **Default error handler** element are not compatible with Orchestrator 5.5.x or earlier.

Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.

- Add some elements to the workflow schema.

Procedure

- 1 Drag a **Default error handler** element to the workflow schema.
- 2 (Optional) Add schema elements between the **Default error handler** element and the **Throw exception** element to specify how global workflow errors are handled.

Results

You added a global error handler to a workflow. When an error that is not caught by standard error handlers in the workflow occurs, the global error handler performs the specified actions before ending the workflow run.

Foreach Elements and Composite Types

You can insert a Foreach element in the workflow that you develop to run a subworkflow that iterates over arrays of parameters or attributes. To improve the understanding and readability of the workflow, you can group several workflow parameters of different types that are logically connected in a single type that is called a composite type.

Using Foreach Elements

A Foreach element runs a subworkflow iteratively over an array of input parameters or attributes. You can select the arrays over which the subworkflow is run, and can pass the values for the elements of such an array when you run the workflow. The subworkflow runs as many times as the number of elements that you have defined in the array.

If you have a configuration element that contains an array of attributes, you can run a workflow that iterates over these attributes in a Foreach element.

For example, suppose that you have 10 virtual machines in a folder that you want to rename. To do this, you must insert a Foreach element in a workflow and define the Rename virtual machine workflow as a subworkflow in the element. The Rename virtual machine workflow takes two input parameters, a virtual machine and its new name. You can promote these parameters as input to the current workflow, and as a result, they become arrays over which the Rename virtual machine workflow will iterate. When you run your workflow, you can specify the 10 virtual machines in the folder and their new names. Every time the workflow runs, it takes an element from the array of the virtual machines and an element from the array of the new names for the virtual machines.

Using Composite Types

A composite type is a group of more than one input parameter or attribute that are connected logically but are of different types. In a Foreach element, you can bind a group of parameters as a composite value. In this way, the Foreach element takes the values for the grouped parameters at once in every subsequent run of the workflow.

For example, suppose that you are about to rename a virtual machine. You need the virtual machine object and its new name. If you have to rename multiple virtual machines, you need two arrays, one for the virtual machines and one for their names. These two arrays are not explicitly connected. A composite type lets you have one array where each element contains both the virtual machine and its new name. In this way, the connection between those two parameters in case of multiple values is specified explicitly and not implied by the workflow schema.

Note You cannot run a workflow that contains composite types from the vSphere Web Client.

Define a Foreach Element

If you want to run a subworkflow multiple times by passing different values for its parameters or attributes in every subsequent run, you can insert a Foreach element in the parent workflow.

When you insert a Foreach element, you must select at least one array over which the Foreach element iterates. An array element can have different values for each subsequent workflow run.

If the subworkflow has output parameters, you should select the output parameters of the Foreach element in which to accumulate workflow outputs, so that the subworkflow can iterate over them as well.

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

- 1 In the workflow editor, select the **Schema** tab.
- 2 From the **Generic** menu, drag a Foreach element in the workflow schema.
- 3 Select a workflow from the Chooser dialog box.

The following notification appears at the top of the schema pane.

Do you want to add the activity's parameters as input/output to the current workflow?

- 4 On the notification, click **Setup**.

A pop-up window with the available options appears.

- 5 Select the mapping type for each input parameter.

Option	Description
Input	The argument is mapped to an input workflow parameter.
Skip	The argument is mapped to a NULL value.
Value	The argument is mapped to an attribute with a value that you can set from the Value column.

- 6 Select the mapping type for each output parameter.

Option	Description
Output	The argument is mapped to an output workflow parameter.
Skip	The argument is mapped to a NULL value.
Local variable	The argument is mapped to an attribute.

- 7 Click **Promote**.
- 8 Right-click the Foreach element and select **Synchronize > Synchronize presentation**.
A confirmation dialog box appears.
- 9 Click **Ok** to propagate the presentation of the Foreach element to the current workflow.
A dialog box displays information about the outcome of the operation.
- 10 On the **Inputs** tab, verify that the subworkflow's parameters are added as elements of type array.
- 11 On the **Outputs** tab, verify that the subworkflow's parameters are added as elements of type array.

Results

You defined a Foreach element in your workflow. The Foreach element runs a workflow that takes as parameters every element from the array of parameters or attributes that you have defined.

For parameters or attributes that are not defined as arrays, the workflow takes the same value in every subsequent run.

Example: Rename Virtual Machines by Using a Foreach Element

You can use a Foreach element to rename several virtual machines at once. You have to insert a Foreach element in a workflow and promote the `vm` and the `newName` parameters as input to the current workflow. In this way, when you run the workflow, you specify the virtual machines to rename and the new names for the virtual machines. The virtual machines are included as elements in the array that you created for the `vm` parameter. The new names for the virtual machines are included in the array that you created for the `newName` parameter.

Define a Composite Type in a Foreach Element

You can group multiple workflow parameters that are connected logically in a new type that is called a composite type. You can use a Foreach element to bind a group of parameters as a composite value to connect several arrays of parameters in a single array.

Prerequisites

- Open a workflow for editing in the workflow editor.
- Verify that you have a Foreach element in your workflow.

Procedure

- 1 Select the **IN** or the **OUT** tab of the Foreach element.
- 2 Select a local parameter that you want to group with other local parameters in a composite type.
- 3 Click **Bind a group of parameters as composite value** at the top of the **IN** or the **OUT** tab.
- 4 In the Bindings pane, select the parameters that you want to group as a composite type.
- 5 Select **Bind as iterator**.

You have set the Foreach element to iterate over an array of the composite type.

- 6 Click **Accept**.

Results

You defined a composite type and made sure that the workflow will iterate over an array of this composite type. Parameters that are grouped as a composite type are named *composite_type_name.parameter_name*. For example, if you create a `snapshots` composite type, the parameters that are group in the type can be `snapshots.vm[in-parameter]` or `snapshots.name[in-parameter]`. Every element from the array of the composite type contains a single instance of every parameter that you grouped in the composite type.

Example: Rename Virtual Machines

Suppose that you want to rename 10 virtual machines at a time. For this, you insert a Foreach element in a workflow and select the Rename virtual machine workflow in the element. You create a composite type to connect the `vm` and the `newName` parameters explicitly. You bind the composite type as an iterator, thus creating a single array that contains both the `vm` and the `newName` parameter.

Add a Switch Activity to a Workflow

You can add a basic switch activity to a workflow schema that defines the switch cases based on workflow attributes or parameters.

Every switch activity can have multiple switch cases. Every switch case is defined by a condition related to an attribute or a parameter. If the condition is fulfilled, the workflow run switches to a corresponding workflow element that you define. If none of the specified conditions are fulfilled, the workflow run switches to a default workflow element that you define.

Important Workflows that contain a **Switch** element are not compatible with Orchestrator 5.5.x or earlier.

Prerequisites

Verify that the **Schema** tab of the workflow editor contains elements.

Procedure

- 1 Drag a **Switch** element to the appropriate element in the workflow schema.
- 2 Click the **Edit** icon (✎) of the **Switch** element.
- 3 In the **Cases** tab, add or delete switch cases.
You can change the priority of switch cases.
- 4 Define the condition for each switch case.
- 5 Select the corresponding workflow element for each switch case.
- 6 Select the default workflow element to switch to.
- 7 Click **Close**.
- 8 Click **Save**.

Results

You defined the switch case conditions and workflow paths.

Developing Plug-Ins

Orchestrator allows integration with management and administration solutions through its open plug-in architecture. You use the Orchestrator client to run and create plug-in workflows and access the plug-in API.

Overview of Plug-Ins

Orchestrator plug-ins must include a standard set of components and must adhere to a standard architecture. These practices help you to create plug-ins for the widest possible variety of external technologies.

■ [Structure of an Orchestrator Plug-In](#)

Orchestrator plug-ins have a common structure that consists of various types of layers that implement specific functionality.

■ [Exposing an External API to Orchestrator](#)

You expose an API from an external product to the Orchestrator platform by creating an Orchestrator plug-in. You can create a plug-in for any technology that exposes an API that you can map into JavaScript objects that Orchestrator can use.

■ [Components of a Plug-In](#)

Plug-ins are composed of a standard set of components that expose the objects in the plugged-in technology to the Orchestrator platform.

- **Role of the vso.xml File**

You use the `vso.xml` file to map the objects, classes, methods, and attributes of the plugged-in technology to Orchestrator inventory objects, scripting types, scripting classes, scripting methods, and attributes. The `vso.xml` file also defines the configuration and start-up behavior of the plug-in.

- **Roles of the Plug-In Adapter**

The plug-in adapter is the entry point of the plug-in to the Orchestrator server. The plug-in adapter serves as the datastore for the plugged-in technology in the Orchestrator server, creates the plug-in factory, and manages events that occur in the plugged-in technology.

- **Roles of the Plug-In Factory**

The plug-in factory defines how Orchestrator finds objects in the plugged-in technology and performs operations on the objects.

- **Role of Finder Objects**

Finder objects identify and locate specific instances of managed object types in the plugged-in technology. Orchestrator can modify and interact with objects that it finds in the plugged-in technology by running workflows on the finder objects.

- **Role of Scripting Objects**

Scripting objects are JavaScript representations of objects from the plugged-in technology. Scripting objects from plug-ins appear in the Orchestrator Javascript API and you can use them in scripted elements in workflows and actions.

- **Role of Event Handlers**

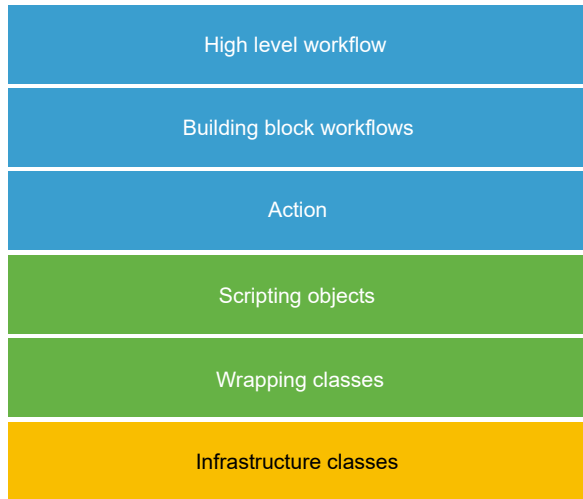
Events are changes in the states or attributes of the objects that Orchestrator finds in the plugged-in technology. Orchestrator monitors events by implementing event handlers.

Structure of an Orchestrator Plug-In

Orchestrator plug-ins have a common structure that consists of various types of layers that implement specific functionality.

The bottom three layers of a Orchestrator plug-in, which are the infrastructure classes, wrapping classes, and scripting objects, implement the connection between the plugged-in technology and Orchestrator.

The user-visible parts of a Orchestrator plug-in are the top three layers, which are actions, building blocks, and high-level workflows.

Figure 1-3. Structure of an Orchestrator Plug-In**Infrastructure classes**

A set of classes that provide the connection between the plugged-in technology and Orchestrator. The infrastructure classes include the classes to implement according to the plug-in definition, such as plug-in factory, plug-in adaptor, and so on. The infrastructure classes also include the classes that provide functionality for common tasks and objects such as helpers, caching, inventory, and so on.

Wrapping classes

A set of classes that adapt the object model of the plugged-in technology to the object model that you want to expose inside Orchestrator.

Scripting objects

JavaScript object types that provide access to the wrapping classes, methods, and attributes in the plugged-in technology. In the `vso.xml` file, you define which wrapping classes, attributes, and methods from the plugged-in technology will be exposed to Orchestrator.

Actions

A set of JavaScript functions that you can use directly in workflows and scripting tasks. Actions can take multiple input parameters and have a single return value.

Building block workflows

A set of workflows that cover all generic functionality that you want to provide with the plug-in. Typically, a building block workflow represents an operation in the user interface of the orchestrated technology. The building block workflows can be used directly or can be included inside high-level workflows.

High-level workflows

A set of workflows that cover specific functionality of the plug-in. You can provide high-level workflows to meet concrete requirements or to show complex examples of the plug-in usage.

Exposing an External API to Orchestrator

You expose an API from an external product to the Orchestrator platform by creating an Orchestrator plug-in. You can create a plug-in for any technology that exposes an API that you can map into JavaScript objects that Orchestrator can use.

Plug-ins map Java objects and methods to JavaScript objects that they add to the Orchestrator scripting API. If an external technology exposes a Java API, you can map the API directly to JavaScript for Orchestrator to use in workflows and actions.

You can create plug-ins for applications that expose an API in a language other than Java by using WSDL (Web service definition language), REST (Representational state transfer), or a messaging service to integrate the exposed API with Java objects. You then map the integrated Java objects to JavaScript for Orchestrator to use.

The plugged-in technology is independent from Orchestrator. You can create Orchestrator plug-ins for external products even if you only have access to binary code, for example in Java archives (JAR files), rather than source code.

Components of a Plug-In

Plug-ins are composed of a standard set of components that expose the objects in the plugged-in technology to the Orchestrator platform.

The main components of a plug-in are the plug-in adapter, factory, and event implementations. You map the objects and operations defined in the adapter, factory, and event implementations to Orchestrator objects in an XML definition file named `vso.xml`. The `vso.xml` file maps objects and functions from the plugged in technology to JavaScript scripting objects that appear in the Orchestrator JavaScript API. The `vso.xml` file also maps object types from the plugged-in technology to finders, that appear in the Orchestrator **Inventory** tab.

Plug-ins are composed of the following components.

Plug-In Module

The plug-in itself, as defined by a set of Java classes, a `vso.xml` file, and packages of the workflows and actions that interact with the objects that you access through the plug-in. The plug-in module is mandatory.

Plug-In Adapter

Defines the interface between the plugged-in technology and the Orchestrator server. The adapter is the entry point of the plug-in to the Orchestrator platform. The adapter creates the plug-in factory, manages the loading and unloading of the plug-in, and manages the events that occur on the objects in the plugged-in technology. The plug-in adapter is mandatory.

Plug-In Factory

Defines how Orchestrator finds objects in the plugged-in technology and performs operations on them. The adapter creates a factory for the client session that opens between Orchestrator and a plugged-in technology. The factory allows you either to share a session between all client connections or to open one session per client connection. The plug-in factory is mandatory.

Configuration

Orchestrator does not define a standard way for the plug-in to store its configuration. You can store configuration information by using Windows Registries, static configuration files, storing information in a database, or in XML files. Orchestrator plug-ins can be configured by running configuration workflows in the Orchestrator client.

Finders

Interaction rules that define how Orchestrator locates and represents the objects in the plugged-in technology. Finders retrieve objects from the set of objects that the plugged-in technology exposes to Orchestrator. You define in the `vso.xml` file the relations between objects to allow you to navigate through the network of objects. Orchestrator represents the object model of the plugged-in technology in the **Inventory** tab. Finders are mandatory if you want to expose objects in the plugged-in technology to Orchestrator.

Scripting Objects

JavaScript object types that provide access to the objects, operations, and attributes in the plugged-in technology. Scripting objects define how Orchestrator accesses the object model of the plugged-in technology through JavaScript. You map the classes and methods of the plugged-in technology to JavaScript objects in the `vso.xml` file. You can access the JavaScript objects in the Orchestrator scripting API and integrate them into Orchestrator scripted tasks, actions, and workflows. Scripting objects are mandatory if you want to add scripting types, classes, and methods to the Orchestrator JavaScript API.

Inventory

Instances of objects in the plugged-in technology that Orchestrator locates by using finders appear in the **Inventory** view in the Orchestrator client. You can perform operations on the objects in the inventory by running workflows on them. The inventory is optional. You can create a plug-in that only adds scripting types and classes to the Orchestrator JavaScript API and does not expose any instances of objects in the inventory.

Events

Changes in the state of an object in the plugged-in technology. Orchestrator can listen passively for events that occur in the plugged-in technology. Orchestrator can also actively trigger events in the plugged-in technology. Events are optional.

Role of the `vso.xml` File

You use the `vso.xml` file to map the objects, classes, methods, and attributes of the plugged-in technology to Orchestrator inventory objects, scripting types, scripting classes, scripting

methods, and attributes. The `vso.xml` file also defines the configuration and start-up behavior of the plug-in.

The `vso.xml` file performs the following principal roles.

Start-Up and Configuration Behavior

Defines the manner in which the plug-in starts and locates any configuration implementations that the plug-in defines. Loads the plug-in adapter.

Inventory Objects

Defines the types of objects that the plug-in accesses in the plugged-in technology. The finder methods of the plug-in factory implementation locate instances of these objects and display them in the Orchestrator inventory.

Scripting Types

Adds scripting types to the Orchestrator JavaScript API to represent the different types of object in the inventory. You can use these scripting types as input parameters in workflows.

Scripting Classes

Adds classes to the Orchestrator JavaScript API that you can use in scripted elements in workflows, actions, policies, and so on.

Scripting Methods

Adds methods to the Orchestrator JavaScript API that you can use in scripted elements in workflows, actions, policies, and so on.

Scripting Attributes

Adds the attributes of the objects in the plugged-in technology to the Orchestrator JavaScript API that you can use in scripted elements in workflows, actions, policies, and so on.

Roles of the Plug-In Adapter

The plug-in adapter is the entry point of the plug-in to the Orchestrator server. The plug-in adapter serves as the datastore for the plugged-in technology in the Orchestrator server, creates the plug-in factory, and manages events that occur in the plugged-in technology.

To create a plug-in adapter, you create a Java class that implements the `IPluginAdaptor` interface.

The plug-in adapter class that you create manages the plug-in factory, events, and triggers in the plugged-in technology. The `IPluginAdaptor` interface provides methods that you use to perform these tasks.

The plug-in adapter performs the following principal roles.

Creates a factory

The most important role of the plug-in adapter is to load and unload one plug-in factory instance for every connection from Orchestrator to the plugged-in technology. The plug-in adapter class calls the `IPluginAdaptor.createPluginFactory()` method to create an instance of a class that implements the `IPluginFactory` interface.

Manages events

The plug-in adapter is the interface between the Orchestrator server and the plugged-in technology. The plug-in adapter manages the events that Orchestrator performs or watches for on the objects in the plugged-in technology. The adapter manages events through event publishers. Event publishers are instances of the `IPluginEventPublisher` interface that the adapter creates by calling the `IPluginAdaptor.registerEventPublisher()` method. Event publishers set triggers and gauges on objects in the plugged-in technology, to allow Orchestrator to launch defined actions if certain events occur on the object, or if the object's values pass certain thresholds. Similarly, you can define `PluginTrigger` and `PluginWatcher` instances that define events that Wait Event elements in long-running workflows await.

Sets the plug-in name

You provide a name for the plug-in in the `vso.xml` file. The plug-in adapter gets this name from the `vso.xml` file and publishes it in the Orchestrator client **Inventory** view.

Installs licenses

You can call methods to install any license files that the plugged-in technology requires in the adapter implement.

For full details of the `IPluginAdaptor` interface, all of its methods, and all of the other classes of the plug-in API, see [Orchestrator Plug-In API Reference](#).

Roles of the Plug-In Factory

The plug-in factory defines how Orchestrator finds objects in the plugged-in technology and performs operations on the objects.

To create the plug-in factory, you must implement and extend the `IPluginFactory` interface from the Orchestrator plug-in API. The plug-in factory class that you create defines the finder functions that Orchestrator uses to access objects in the plugged-in technology. The factory allows the Orchestrator server to find objects by their ID, by their relation to other objects, or by searching for a query string.

The plug-in factory performs the following principal tasks.

Finds objects

You can create functions that find objects according to their name and type. You find objects by name and type by using the `IPluginFactory.find()` method.

Finds objects related to other objects

You can create functions to find objects that relate to a given object by a given relation type. You define relations in the `vso.xml` file. You can also create finders to find dependent child objects that relate to all parents by a given relation type. You implement the `IPluginFactory.findRelation()` method to find any objects that are related to a given parent object by a given relation type. You implement the `IPluginFactory.hasChildrenInRelation()` method to discover whether at least one child object exists for a parent instance.

Define queries to find objects according to your own criteria

You can create object finders that implement query rules that you define. You implement the `IPluginFactory.findAll()` method to find all objects that satisfy query rules you define when the factory calls this method. You obtain the results of the `findAll()` method in a `QueryResult` object that contains a list of all of the objects found that match the query rules you define.

For more information about the `IPluginFactory` interface, all of its methods, and all of the other classes of the plug-in API, see [Orchestrator Plug-In API Reference](#).

Role of Finder Objects

Finder objects identify and locate specific instances of managed object types in the plugged-in technology. Orchestrator can modify and interact with objects that it finds in the plugged-in technology by running workflows on the finder objects.

Every instance of a given managed object type in the plugged-in technology must have a unique identifier so that Orchestrator finder objects can find them. The plugged-in technology provides the unique identifiers for the object instances as strings. When a workflow runs, Orchestrator sets the unique identifiers of the objects that it finds as workflow attribute values. Workflows that require an object of a given type as an input parameter run on a specific instance of that type of object.

Finder objects that plug-ins add to the Orchestrator JavaScript API have the plug-in name as a prefix. For example, the `VirtualMachine` managed object type from the vCenter Server API appears in Orchestrator as the `VC:VirtualMachine` JavaScript type.

For example, Orchestrator accesses a specific `VC:VirtualMachine` instance through the vCenter Server plug-in by implementing a finder object that uses the `id` attribute of the virtual machine as its unique identifier. You can pass this object instance to workflow elements as attribute values.

An Orchestrator plug-in maps the objects from the plugged-in technology to equivalent Orchestrator finder objects in the `<finder>` elements in the `vso.xml` file. The `<finder>` elements identify the method or function from the plugged-in technology that obtains the unique identifier for a specific instance of an object. The `<finder>` elements also define relations between objects, to find objects by the manner in which they relate to other objects.

Finder objects appear in the Orchestrator **Inventory** tab under the plug-in that contains them.

Role of Scripting Objects

Scripting objects are JavaScript representations of objects from the plugged-in technology. Scripting objects from plug-ins appear in the Orchestrator JavaScript API and you can use them in scripted elements in workflows and actions.

Scripting objects from plug-ins appear in the Orchestrator JavaScript API as JavaScript modules, types, and classes. Most finder objects have a scripting object representation. The JavaScript classes can add methods and attributes to the Orchestrator JavaScript API that represent the methods and attributes from objects from the API of the plugged-in technology. The plugged-in technology provides the implementations of the objects, types, classes, attributes, and methods independently of Orchestrator. For example, the vCenter Server plug-in represents all the objects from the vCenter Server API as JavaScript objects in the Orchestrator JavaScript API, with JavaScript representations of all the classes, methods and attributes that the vCenter Server API defines. You can use the vCenter Server scripting classes and the methods and attributes they define in Orchestrator scripted functions.

For example, the `VirtualMachine` managed object type from the vCenter Server API is found by the `VC:VirtualMachine` finder and appears in the Orchestrator JavaScript API as the `VcVirtualMachine` JavaScript class. The `VcVirtualMachine` JavaScript class in the Orchestrator JavaScript API defines all of the same methods and attributes as the `VirtualMachine` managed object from the vCenter Server API.

An Orchestrator plug-in maps the objects, types, classes, attributes, and methods from the plugged-in technology to equivalent Orchestrator JavaScript objects, types, classes, attributes, and methods in the `<scripting-objects>` element in the `vso.xml` file.

Role of Event Handlers

Events are changes in the states or attributes of the objects that Orchestrator finds in the plugged-in technology. Orchestrator monitors events by implementing event handlers.

Orchestrator plug-ins allow you to monitor events in a plugged-in technology in different ways. The Orchestrator plug-in API allows you to create the following types of event handlers to monitor events in a plugged-in technology.

Listeners

Passively monitor objects in the plugged-in technology for changes in their state. The plugged-in technology or the plug-in implementation defines the events that listeners monitor. Listeners do not initiate events, but notify Orchestrator when the events occur. Listeners detect events either by polling the plugged-in technology or by receiving notifications from the plugged-in technology. When events occur, Orchestrator policies or workflows that are waiting for the event can react by starting operations in the Orchestrator server. Listener components are optional.

Policies

Monitor certain events in the plugged-in technology and start operations in the Orchestrator server if the events occur. Policies can monitor policy triggers and policy gauges. Policy

triggers define an event in the plugged-in technology that, when it occurs, causes a running policy to start an operation in the Orchestrator server, for example running a workflow. Policy gauges define ranges of values for the attributes of an object in the plugged-in technology that, when exceeded, cause Orchestrator to start an operation. Policies are optional.

Workflow triggers

If a running workflow contains a Wait Event element, when it reaches that element it suspends its run and waits for an event to occur in a plugged-in technology. Workflow triggers define the events in the plugged-in technology that Waiting Event elements in workflows await. You register workflow triggers with watchers. Workflow triggers are optional.

Watchers

Watch workflow triggers for a certain event in the plugged-in technology, on behalf of a Waiting Event element in a workflow. When the event occurs, the watchers notify any workflows that are waiting for that event. Watchers are optional.

Contents and Structure of a Plug-In

Orchestrator plug-ins must contain a standard set of components and conform to a standard file structure. For a plug-in to conform to the standard file structure, it must include specific folders and files.

To create an Orchestrator plug-in, you define how Orchestrator accesses and interacts with the objects in the plugged-in technology. And, you map all of the objects and functions of the plugged-in technology to corresponding Orchestrator objects and functions in the `vso.xml` file.

The `vso.xml` file must include a reference to every type of object or operation to expose to Orchestrator. Every object that the plug-in finds in the plugged-in technology must have a unique identifier that you provide. You define the object names in the `finder` elements and in the `object` elements in the `vso.xml` file.

A plug-in can be delivered as a standard Java archive file (JAR) or a ZIP file, but in either case, the file must be renamed with a `.dar` extension.

Note You can use the Orchestrator Control Center to import a DAR file to the Orchestrator server.

- [Defining the Application Mapping in the vso.xml File](#)

Objects that you include in the `vso.xml` file appear as scripting objects in the Orchestrator scripting API, or as finder objects in the Orchestrator **Inventory** tab.

- [Format of the vso.xml Plug-In Definition File](#)

The `vso.xml` file defines how the Orchestrator server interacts with the plugged-in technology. You must include a reference to every type of object or operation to expose to Orchestrator in the `vso.xml` file.

■ Naming Plug-In Objects

You must provide a unique identifier for every object that the plug-in finds in the plugged-in technology. You define the object names in the <finder> elements and in the <object> elements in the `vso.xml` file.

■ Plug-In Object Naming Conventions

You must follow Java class naming conventions when you name all objects in plug-ins.

■ File Structure of the Plug-In

A plug-in must conform to a standard file structure and must include certain specific folders and files. You deliver a plug-in as a standard Java archive (JAR) or ZIP file, that you must rename with the `.dar` extension.

Defining the Application Mapping in the `vso.xml` File

Objects that you include in the `vso.xml` file appear as scripting objects in the Orchestrator scripting API, or as finder objects in the Orchestrator **Inventory** tab.

The `vso.xml` file provides the following information to the Orchestrator server:

- A version, name, and description for the plug-in
- References to the classes of the plugged-in technology and to the associated plug-in adapter
- Initializes the plug-in when the Orchestrator server starts
- Scripting types to represent the types of objects in the plugged-in technology
- The relationships between object types to define how the objects display in the Orchestrator Inventory
- Scripting classes that map the objects and operations in the plugged-in technology to functions and object types in the Orchestrator JavaScript API
- Enumerations to define a list of constant values that apply to all objects of a certain type
- Events that Orchestrator monitors in the plugged-in technology

The `vso.xml` file must conform to the XML schema definition of Orchestrator plug-ins. You can access the schema definition at the VMware support site.

```
http://www.vmware.com/support/orchestrator/plugin-4-1.xsd
```

For descriptions of all of the elements of the `vso.xml` file, see [Elements of the `vso.xml` Plug-In Definition File](#).

Format of the `vso.xml` Plug-In Definition File

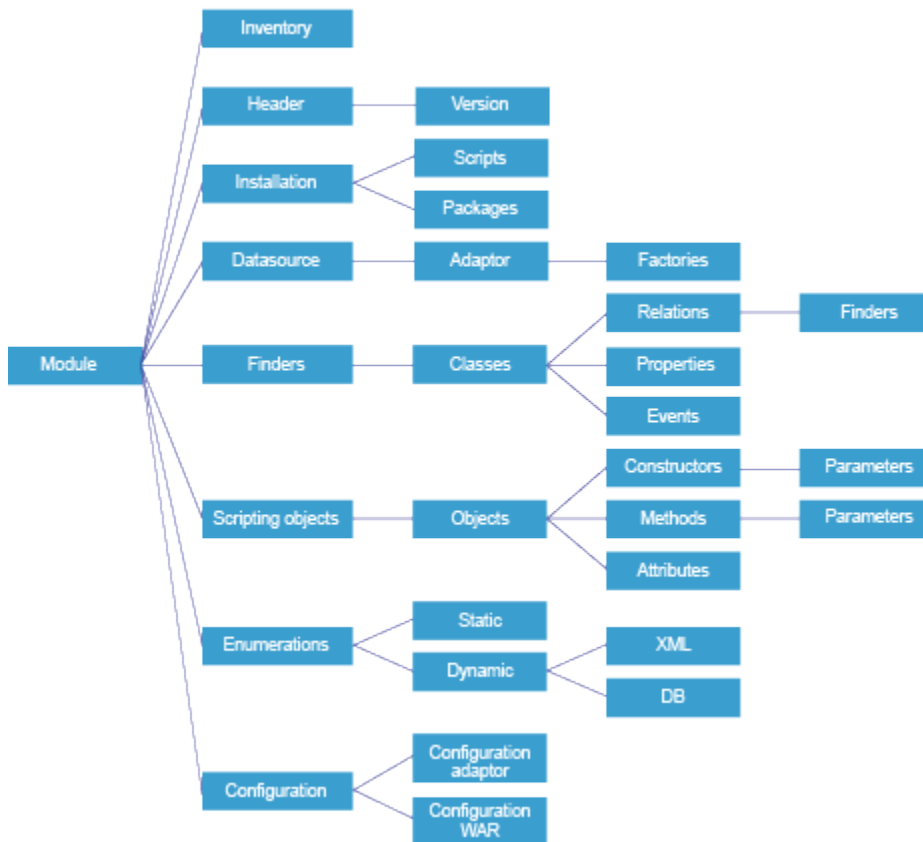
The `vso.xml` file defines how the Orchestrator server interacts with the plugged-in technology. You must include a reference to every type of object or operation to expose to Orchestrator in the `vso.xml` file.

Objects that you include in the `vso.xml` file appear as scripting objects in the Orchestrator scripting API, or as finder objects in the Orchestrator **Inventory** tab.

As part of the open architecture and standardized implementation of plug-ins, the `vso.xml` file must adhere to a standard format.

The following diagram shows the format of the `vso.xml` plug-in definition file and how the elements nest within each other.

Figure 1-4. Format of the `vso.xml` Plug-In Definition File



Naming Plug-In Objects

You must provide a unique identifier for every object that the plug-in finds in the plugged-in technology. You define the object names in the `<finder>` elements and in the `<object>` elements in the `vso.xml` file.

The finder operations that you define in the factory implementation find objects in the plugged-in technology. When the plug-in finds objects, you can use them in Orchestrator workflows and pass them from one workflow element to another. The unique identifiers that you provide for the objects allows them to pass between the elements in a workflow.

The Orchestrator server stores only the type and identifier of each object that it processes, and stores no information about where or how Orchestrator obtained the object. You must name objects consistently in the plug-in implementation so that you can track the objects you obtain from plug-ins.

If the Orchestrator server stops while workflows are running, when you restart the server the workflows resume at the workflow element that was running when the server stopped. The workflow uses the identifiers to retrieve objects that the element was processing when the server stopped.

Plug-In Object Naming Conventions

You must follow Java class naming conventions when you name all objects in plug-ins.

Important Because of the way in which the workflow engine performs data serialization, do not use the following string sequences in object names. Using these character sequences in object identifiers causes the workflow engine to parse workflows incorrectly, which can cause unexpected behavior when you run the workflows.

- #;#
 - #,#
 - #=#
-

Use these guidelines when you name objects in plug-ins.

- Use an initial uppercase letter for each word in the name.
- Do not use spaces to separate words.
- For letters, only use the standard characters A to Z and a to z.
- Do not use special characters, such as accents.
- Do not use a number as the first character of a name.
- Where possible, use fewer than 10 characters.

[Table 1-5. Plug-In Object Naming Rules](#) shows rules that apply to individual object types.

Table 1-5. Plug-In Object Naming Rules

Object Type	Naming Rules
Plug-In	<ul style="list-style-type: none"> ■ Defined in the <code><module></code> element in the <code>vso.xml</code> file. ■ Must adhere to Java class naming conventions. ■ Must be unique. You cannot run two plug-ins with the same name in an Orchestrator server.
Finder object	<ul style="list-style-type: none"> ■ Defined in the <code><finder></code> elements in the <code>vso.xml</code> file. ■ Must adhere to Java class naming conventions. ■ Must be unique in the plug-in. <p>Orchestrator adds the plug-in name and a colon to the finder object names in the finder object types in the Orchestrator scripting API. For example, the <code>VirtualMachine</code> object type from the vCenter Server plug-in appears in the Orchestrator scripting API as <code>VC:VirtualMachine</code>.</p>
Scripting object	<ul style="list-style-type: none"> ■ Defined in the <code><scripting-object></code> elements in the <code>vso.xml</code> file. ■ Must adhere to Java class naming conventions. ■ Must be unique in the Orchestrator server. ■ To avoid confusing scripting objects with finder objects of the same name or with scripting objects from other plug-ins, always prefix the scripting object name with the name of the plug-in, but do not add a colon. For example, the <code>VirtualMachine</code> class from the vCenter Server plug-in appears in the Orchestrator scripting API as the <code>VcVirtualMachine</code> class.

File Structure of the Plug-In

A plug-in must conform to a standard file structure and must include certain specific folders and files. You deliver a plug-in as a standard Java archive (JAR) or ZIP file, that you must rename with the `.dar` extension.

The contents of the DAR archive must use the following folder structure and naming conventions.

Table 1-6. Structure of the DAR Archive

Folders	Description
<i>plug-in_name</i> \VSO-INF\	Contains the <code>vso.xml</code> file that defines the mapping of the objects in the plugged-in technology to Orchestrator objects. The VSO-INF folder and the <code>vso.xml</code> file are mandatory.
<i>plug-in_name</i> \lib\	Contains the JAR files that contain the binaries of the plugged-in technology. Also contains JAR files that contain the implementations of the adapter, factory, notification handlers, and other interfaces in the plug-in. The lib folder and JAR files are mandatory.
<i>plug-in_name</i> \resources\	Contains resource files that the plug-in requires. The resources folder can include the following types of element: <ul style="list-style-type: none"> ■ Image files, to represent the objects of the plug-in in the Orchestrator Inventory tab. ■ Scripts, to define initialization behavior when the plug-in starts. ■ Orchestrator packages, that can contain custom workflows, actions, and other resources that interact with the objects that you access by using the plug-in. You can organize resources in subfolders. For example, <code>resources\images\</code> , <code>resources\scripts\</code> , or <code>resources\packages\</code> . The resources folder is optional.

You use the Orchestrator Control Center to import a DAR file to the Orchestrator server.

Orchestrator Plug-In API Reference

The Orchestrator plug-in API defines Java interfaces and classes to implement and extend when you develop the `IPluginAdaptor` and `IPluginFactory` implementations to create a plug-in.

All classes are contained in the `ch.dunes.vso.sdk.api` package, unless stated otherwise.

IAop Interface

The IAop interface provides methods to obtain and set properties on objects in the plugged-in technology.

```
public interface IAop
```

The IAop interface defines the following methods:

Method	Returns	Description
<code>get(java.lang.String propertyName, java.lang.Object object, java.lang.Object sdkObject)</code>	<code>java.lang.Object</code>	Obtains a property from a given object in the plug-in.
<code>set(java.lang.String propertyName, java.lang.String propertyValue, java.lang.Object object)</code>	<code>Void</code>	Sets a property on a given object in the plug-in.

IDynamicFinder Interface

The IDynamicFinder interface returns the ID and properties of a finder programmatically, instead defining the ID and properties in the `vso.xml` file.

The IDynamicFinder Interface defines the following methods.

Method	Returns	Description
<code>getIdAccessor(java.lang.String type)</code>	<code>java.lang.String</code>	Provides an OGNL expression to obtain an object ID programmatically.
<code>getProperties(java.lang.String type)</code>	<code>java.util.List<SDKFinderProperty></code>	Provides a list of object properties programmatically.

IPluginAdaptor Interface

You implement the IPluginAdaptor interface to manage plug-in factories, events and watchers. The IPluginAdaptor interface defines an adapter between a plug-in and the Orchestrator server.

IPluginAdaptor instances are responsible for session management. The IPluginAdaptor Interface defines the following methods.

Method	Returns	Description
<code>addWatcher(PluginWatcher watcher)</code>	<code>Void</code>	Adds a watcher to monitor for a specific event
<code>createPluginFactory(java.lang.String sessionId, java.lang.String username, java.lang.String password, IPluginNotificationHandler notificationHandler)</code>	<code>IPluginFactory</code>	<p>Creates an IPluginFactory instance. The Orchestrator server uses the factory to obtain objects from the plugged-in technology by their ID, by their relation to other objects, and so on.</p> <p>The session ID allows you to identify a running session. For example, a user could log into two different Orchestrator clients and run two sessions simultaneously.</p> <p>Similarly, starting a workflow creates a session that is independent from the client in which the workflow started. A workflow continues to run even if you close the Orchestrator client.</p>

Method	Returns	Description
<code>installLicenses(PluginLicense[] licenses)</code>	Void	Installs the license information for standard plug-ins that VMware provides
<code>registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)</code>	Void	Sets triggers and gauges on an element in the inventory
<code>removeWatcher(java.lang.String watcherId)</code>	Void	Removes a watcher
<code>setPluginName(java.lang.String pluginName)</code>	Void	Gets the plug-in name from the <code>vso.xml</code> file
<code>setPluginPublisher(IPluginPublisher pluginPublisher)</code>	Void	Sets the publisher of the plug-in
<code>uninstallPluginFactory(IPluginFactory plugin)</code>	Void	Uninstalls a plug-in factory.
<code>unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)</code>	Void	Removes triggers and gauges from an element in the inventory

IPluginEventPublisher Interface

The `IPluginEventPublisher` interface publishes gauges and triggers on an event notification bus for Orchestrator policies to monitor.

You can create `IPluginEventPublisher` instances directly in the plug-in adaptor implementation or you can create them in separate event generator classes.

You can implement the `IPluginEventPublisher` interface to publish events in the plugged-in technology to the Orchestrator policy engine. You create methods to set policy triggers and gauges on objects in the plugged-in technology and event listeners to listen for events on those objects.

Policies can implement either gauges or triggers to monitor objects in the plugged-in technology. Policy gauges monitor the attributes of objects and push an event in the Orchestrator server if the values of the objects exceed certain limits. Policy triggers monitor objects and push an event in the Orchestrator server if a defined event occurs on the object. You register policy gauges and triggers with `IPluginEventPublisher` instances so that Orchestrator policies can monitor them.

The `IPluginEventPublisher` Interface defines the following methods.

Type	Returns	Description
<code>pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)</code>	Void	<p>Publish a gauge for policies to monitor. Takes the following parameters:</p> <ul style="list-style-type: none"> ■ <code>type</code>: Type of the object to monitor. ■ <code>id</code>: Identifier of the object to monitor. ■ <code>gaugeName</code>: Name for this gauge. ■ <code>deviceName</code>: Name for the type of attribute that the gauge monitors. ■ <code>gaugeValue</code>: Value for which the gauge monitors the object.
<code>pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)</code>	Void	<p>Publish a trigger for policies to monitor. Takes the following parameters:</p> <ul style="list-style-type: none"> ■ <code>type</code>: Type of the object to monitor. ■ <code>id</code>: Identifier of the object to monitor. ■ <code>triggerName</code>: Name for this trigger. ■ <code>additionalProperties</code>: Any additional properties for the trigger to monitor.

IPluginFactory Interface

The IPluginAdaptor returns IPluginFactory instances. IPluginFactory instances run commands in the plugged-in application, and finds objects upon which to perform Orchestrator operations.

The IPluginFactory interface defines the following field:

```
static final java.lang.String RELATION_CHILDREN
```

The IPluginFactory interface defines the following methods.

Method	Returns	Description
<code>executePluginCommand(java.lang.String cmd)</code>	Void	Use the plug-in to run a command. VMware recommends that you do not use this method.
<code>find(java.lang.String type, java.lang.String id)</code>	<code>java.lang.Object</code>	Use the plug-in to find an object. Identify the object by its ID and type.
<code>findAll(java.lang.String type, java.lang.String query)</code>	<code>QueryResult</code>	Use the plug-in to find objects of a certain type and that match a query string. You define the syntax of the query in the IPluginFactory implementation of the plug-in. If you do not define query syntax, <code>findAll()</code> returns all objects of the specified type.

Method	Returns	Description
<code>findRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)</code>	<code>java.util.List</code>	Determines whether an object has children.
<code>hasChildrenInRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)</code>	<code>HasChildrenResult</code>	Finds all children related to a given parent by a certain relation.
<code>invalidate(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	Invalidate objects by type and ID.
<code>void invalidateAll()</code>	<code>Void</code>	Invalidate all objects in the cache.

IPluginNotificationHandler Interface

The `IPluginNotificationHandler` defines methods to notify Orchestrator of different types of event that occur on the objects Orchestrator accesses through the plug-in.

The `IPluginNotificationHandler` Interface defines the following methods.

Method	Returns	Description
<code>getSessionID()</code>	<code>java.lang.String</code>	Returns the current session ID
<code>notifyElementDeleted(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	Notifies the system that an object with the given type and ID has been deleted
<code>notifyElementInvalidate(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	Notifies the system that an object's relations have changed. You can use the <code>notifyElementInvalidate()</code> method to notify Orchestrator of all changes in relations between objects, not only for relation changes that invalidate an object. For example, adding a child object to a parent represents a change in the relation between the two objects.
<code>notifyElementUpdated(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	Notifies the system that an object's attributes have been modified
<code>notifyMessage(ch.dunes.vso.sdk.api.ErrorLevel severity, java.lang.String type, java.lang.String id, java.lang.String message)</code>	<code>Void</code>	Publishes an error message related to the current module

IPluginPublisher Interface

The `IPluginPublisher` interface publishes a watcher event on an event notification bus for long-running workflow Wait Event elements to monitor.

When a workflow trigger starts an event in the plugged-in technology, a plug-in watcher that watches that trigger and that is registered with an `IPluginPublisher` instance notifies any waiting workflows that the event has occurred.

The `IPluginPublisher` Interface defines the following method.

Type	Value	Description
<code>pushWatcherEvent(java.lang.String id, java.util.Properties properties)</code>	<code>Void</code>	Publish a watcher event on event notification bus

WebConfigurationAdaptor Interface

The `WebConfigurationAdaptor` interface implements `IConfigurationAdaptor` and defines methods to locate and install a Web application in the configuration tab for a plug-in.

Note The `WebConfigurationAdaptor` interface is deprecated since Orchestrator 4.1. To add a Web application to the configuration, implement `IConfigurationAdaptor` and use the `configuration-war` attribute in the `vso.xml` file to identify the Web application.

The `WebConfigurationAdaptor` interface defines the following methods.

Method	Returns	Description
<code>getWebAppContext()</code>	<code>String</code>	Locates the WAR file of the Web application for the configuration tab. Provide the name and path to the WAR file from the <code>/webapps</code> directory in the DAR file as a string.
<code>setWebConfiguration(boolean webConfiguration)</code>	<code>Boolean</code>	Determine whether the contents of the configuration tab are defined by a Web application.

PluginTrigger Class

The `PluginTrigger` class creates a trigger module that obtains information about objects and events to monitor in the plugged-in technology, on behalf of a Wait Event element in a workflow.

The `PluginTrigger` class defines methods to obtain or set the type and name of the object to monitor, the nature of the event, and a timeout period.

You create implementations of the `PluginTrigger` class exclusively for use by Wait Event elements in workflows. You define policy triggers for Orchestrator policies in classes that define events and implement the `IPluginEventPublisher.pushTrigger()` method.

```
public class PluginTrigger
extends java.lang.Object
implements java.io.Serializable
```

The `PluginTrigger` class defines the following methods:

Method	Returns	Description
getModuleName()	java.lang.String	Obtains the name of the trigger module.
getProperties()	java.util.Properties	Obtains a list of properties for the trigger.
getSdkId()	java.lang.String	Obtains the ID of the object to monitor in the plugged-in technology.
getSdkType()	java.lang.String	Obtains the type of the object to monitor in the plugged-in technology.
getTimeout()	Long	Obtains the trigger timeout period.
setModuleName(java.lang.String moduleName)	Void	Sets the name of the trigger module.
setProperties(java.util.Properties properties)	Void	Sets a list of properties for the trigger.
setSdkId(java.lang.String sdkId)	Void	Sets the ID of the object to monitor in the plugged-in technology.
setSdkType(java.lang.String sdkType)	Void	Sets the type of the object to monitor in the plugged-in technology.
setTimeout(long timeout)	Void	Sets a timeout period in seconds. A negative value deactivates the timeout.

Constructors

- PluginTrigger()
- PluginTrigger(java.lang.String moduleName, long timeout, java.lang.String sdkType, java.lang.String sdkId)

PluginWatcher Class

The PluginWatcher class watches a trigger module for a defined event in the plugged-in technology on behalf of a long-running workflow Wait Event element.

The PluginWatcher class defines a constructor that you can use to create plug-in watcher instances. The PluginWatcher class defines methods to obtain or set the name of the workflow trigger to watch and a timeout period.

```
public class PluginWatcher
extends java.lang.Object
implements java.io.Serializable
```

The PluginWatcher class defines the following methods:

Method	Returns	Description
getId()	java.lang.String	Obtains the ID of the trigger
getModuleName()	java.lang.String	Obtains the trigger module name

Method	Returns	Description
getTimeoutDate()	Long	Obtains the trigger timeout date
getTrigger()	Void	Obtains a trigger
setId(java.lang.String id)	Void	Sets the ID of the trigger
setTimeoutDate()	Void	Sets the trigger timeout date

Constructor

PluginWatcher(PluginTrigger trigger)

QueryResult Class

The QueryResult class contains the results of a find query made on the objects Orchestrator accesses through the plug-in.

```
public class QueryResult
extends java.lang.Object
implements java.io.Serializable
```

The totalCount value can be greater than the number of elements the QueryResult returns, if the total number of results found exceeds the number of results the query returns. The number of results the query returns is defined in the query syntax in the vso.xml file.

The QueryResult class defines the following methods:

Method	Returns	Description
addElement(java.lang.Object element)	Void	Adds an element to the QueryResult
addElements(java.util.List elements)	Void	Adds a list of elements to the QueryResult
getElements()	java.util.List	Obtains elements from the plugged in application
getTotalCount()	Long	Obtains a count of all the elements available in the plugged in technology
isPartialResult()	Boolean	Determines whether the result obtained is complete
removeElement(java.lang.Object element)	Void	Removes an element from the plugged in technology
setElements(java.util.List elements)	Void	Sets elements in the plugged in technology
setTotalCount(long totalCount)	Void	Sets the total number of elements available in the plugged in technology

Constructors

- QueryResult()
- QueryResult(java.util.List ret)

■ `QueryResult(java.util.List elements, long totalCount)`

SDKFinderProperty Class

The `SDKFinderProperty` class defines methods to obtain and set properties in the objects found in the plugged in technology by the Orchestrator finder objects. The `IDynamicFinder.getProperties` method returns `SDKFinderProperty` objects.

```
public class SDKFinderProperty
extends java.lang.Object
```

The `SDKFinderProperty` class defines the following methods:

Method	Returns	Description
<code>getAttributeName()</code>	<code>java.lang.String</code>	Obtains an object attribute name
<code>getBeanProperty()</code>	<code>java.lang.String</code>	Obtains properties from a Java bean
<code>getDescription()</code>	<code>java.lang.String</code>	Obtains an object description
<code>getDisplayName()</code>	<code>java.lang.String</code>	Obtains an object display name
<code>getPossibleResultType()</code>	<code>java.lang.String</code>	Obtains the possible types of result the finder returns
<code>getPropertyAccessor()</code>	<code>java.lang.String</code>	Obtains an object property accessor
<code>getPropertyAccessorTree()</code>	<code>java.lang.Object</code>	Obtains an object property accessor tree
<code>isHidden()</code>	<code>Boolean</code>	Shows or hides the object
<code>isShowInColumn()</code>	<code>Boolean</code>	Shows or hides the object in the database column
<code>isShowInDescription()</code>	<code>Boolean</code>	Shows or hides the object description
<code>setAttributeName(java.lang.String attributeName)</code>	<code>Void</code>	Sets an object attribute name
<code>setBeanProperty(java.lang.String beanProperty)</code>	<code>Void</code>	Sets properties in a Java bean
<code>setDescription(java.lang.String description)</code>	<code>Void</code>	Sets an object description
<code>setDisplayName(java.lang.String displayName)</code>	<code>Void</code>	Sets an object display name
<code>setHidden(boolean hidden)</code>	<code>Void</code>	Show or hide the object
<code>setPossibleResultType(java.lang.String possibleResultType)</code>	<code>Void</code>	Sets the possible types of result the finder returns
<code>setPropertyAccessor(java.lang.String propertyAccessor)</code>	<code>Void</code>	Sets an object property accessor
<code>setPropertyAccessorTree(java.lang.Object propertyAccessorTree)</code>	<code>Void</code>	Sets an object property accessortree

Method	Returns	Description
setShowInColumn(boolean showInTable)	Void	Show or hide the object in the database column
setShowInDescription(boolean showInDescription)	Void	Show or hide the object description

Constructor

SDKFinderProperty(java.lang.String attributeName, java.lang.String displayName, java.lang.String beanProperty, java.lang.String propertyAccessor)

PluginExecutionException Class

The PluginExecutionException class returns an error message if the plug-in encounters an exception when it runs an operation.

```
public class PluginExecutionException
extends java.lang.Exception
implements java.io.Serializable
```

The PluginExecutionException class inherits the following methods from class java.lang.Throwable:

fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString, fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace

Constructor

PluginExecutionException(java.lang.String message)

PluginOperationException Class

The PluginOperationException class handles errors encountered during a plug-in operation.

```
public class PluginOperationException
extends java.lang.RuntimeException
implements java.io.Serializable
```

The PluginOperationException class inherits the following methods from class java.lang.Throwable:

fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Constructor

PluginOperationException(java.lang.String message)

HasChildrenResult Enumeration

The HasChildrenResult Enumeration declares whether a given parent has children. The `IPluginFactory.hasChildrenInRelation` method returns HasChildrenResult objects.

```
public enum HasChildrenResult
extends java.lang.Enum<HasChildrenResult>
implements java.io.Serializable
```

The HasChildrenResult enumeration defines the following constants:

- `public static final HasChildrenResult Yes`
- `public static final HasChildrenResult No`
- `public static final HasChildrenResult Unknown`

The HasChildrenResult enumeration defines the following methods:

Method	Returns	Description
<code>getValue()</code>	<code>int</code>	Returns one of the following values: 1 Parent has children -1 Parent has no children 0 Unknown, or invalid parameter
<code>valueOf(java.lang.String name)</code>	<code>static HasChildrenResult</code>	Returns an enumeration constant of this type with the specified name. The String must match exactly an identifier used to declare an enumeration constant of this type. Do not use whitespace characters in the enumeration name.
<code>values()</code>	<code>static HasChildrenResult[]</code>	Returns an array containing the constants of this enumeration type, in the order they are declared. This method can iterate over constants as follows: <div style="background-color: #f0f0f0; padding: 10px; margin-top: 10px;"> <pre>for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);</pre> </div>

The HasChildrenResult enumeration inherits the following methods from class `java.lang.Enum`: `clone`, `compareTo`, `equals`, `finalize`, `getDeclaringClass`, `hashCode`, `name`, `ordinal`, `toString`, `valueOf`

ScriptingAttribute Annotation Type

The `ScriptingAttribute` annotation type annotates an attribute from an object in the plugged in technology for use as a property in scripting.

```
@Retention(value=RUNTIME)
@Target(value={METHOD, FIELD})
public @interface ScriptingAttribute
```

The `ScriptingAttribute` annotation type has the following value:

```
public abstract java.lang.String value
```

ScriptingFunction Annotation Type

The `ScriptingFunction` annotation type annotates a method for use as a property in scripting.

```
@Retention(value=RUNTIME)
@Target(value={METHOD, CONSTRUCTOR})
public @interface ScriptingFunction
```

The `ScriptingFunction` annotation type has the following value:

```
public abstract java.lang.String value
```

ScriptingParameter Annotation Type

The `ScriptingParameter` annotation type annotates a parameter for use as a property in scripting.

```
@Retention(value=RUNTIME)
@Target(value=PARAMETER)
public @interface ScriptingParameter
```

The `ScriptingParameter` annotation type has the following value:

```
public abstract java.lang.String value
```

Elements of the vso.xml Plug-In Definition File

The `vso.xml` file contains a set of standard elements. Some of the elements are mandatory while others are optional. Each element has attributes that define values for the objects and operations you map to Orchestrator objects and operations.

In addition, elements can have zero or more child elements. A child element further defines the parent element. The same child element can appear in multiple parent elements. For example, the `description` element has no child elements, but appears as a child element for many parent elements: `module`, `example`, `trigger`, `gauge`, `finder`, `constructor`, `method`, `object`, and `enumeration`.

Each element definition that follows lists its attributes, parents and children.

module Element

A module describes a set of plug-in objects to make available to Orchestrator.

The module contains information about how data from the plugged-in technology maps to Java classes, versioning, how to deploy the module, and how the plug-in appears in the Orchestrator inventory.

The `<module>` element is optional. The `<module>` element has the following attributes:

Attributes	Value	Description
name	String	Defines the type of all the <code><finder></code> elements in the plug-in. Mandatory attribute.
version	Number	The plug-in version number, for use when reloading packages in a new version of the plug-in. Mandatory attribute.
build-number	Number	The plug-in build number, for use when reloading packages in a new version of the plug-in. Mandatory attribute.
image	Image file	The icon to display in the Orchestrator Inventory. Mandatory attribute.
display-name	String	The name that appears in the Orchestrator Inventory. Optional attribute.
interface-mapping-allowed	true or false	VMware strongly discourages interface mapping. Optional attribute.

Table 1-7. Element Hierarchy

Parent Element	Child Elements
None	<ul style="list-style-type: none"> ■ <code><description></code> ■ <code><installation></code> ■ <code><configuration></code> ■ <code><finder-datasources></code> ■ <code><inventory></code> ■ <code><finders></code> ■ <code><scripting-objects></code> ■ <code><enumerations></code>

description Element

The `<description>` elements provide descriptions of the elements of the plug-in that appear in the API Explorer documentation.

You add the text that appears in the API Explorer documentation between the `<description>` and `</description>` tags.

The `<description>` element is optional. The `<description>` element has no attributes.

Table 1-8. Element Hierarchy

Parent Elements	Child Elements
■ <module>	None
■ <example>	
■ <trigger>	
■ <gauge>	
■ <finder>	
■ <constructor>	
■ <method>	
■ <object>	
■ <enumeration>	

deprecated Element

The <deprecated> element marks objects and methods that are deprecated in the API Explorer documentation.

You add the text that appears in the API Explorer documentation between the <deprecated> and </deprecated> tags.

The <deprecated> element is optional. The <deprecated> element has no attributes.

Table 1-9. Element Hierarchy

Parent Elements	Child Elements
■ <method>	None
■ <object>	

url Element

The <url> element provides a URL that points to external documentation about an object or enumeration.

You provide the URL between the <url> and </url> tags.

The <url> element is optional. The <url> element has no attributes.

Table 1-10. Element Hierarchy

Parent Elements	Child Elements
■ <enumeration>	None
■ <object>	

installation Element

The <installation> element allows you to install a package or run a script when the server starts.

The <installation> element is optional. The <installation> element has the following attributes:

Attributes	Value	Description
mode	always, never, or version	Setting the mode value results in the following behavior when the Orchestrator server starts: <ul style="list-style-type: none"> ■ The action <i>always</i> runs ■ The action <i>never</i> runs ■ The action runs when the server detects a newer version of the plug-in Mandatory attribute.

Table 1-11. Element Hierarchy

Parent Element	Child Element
<module>	<action>

action Element

The <action> element specifies the action that runs when the Orchestrator server starts.

The <action> element attributes provide the path to the Orchestrator package or script that defines the plug-in's behavior when it starts.

The <action> element is optional. A plug-in can have an unlimited number of <action> elements. The <action> element has the following attributes.

Attributes	Value	Description
resource	String	The path to the Java package or script from the root of the dar file. Mandatory attribute.
type	install-package or execute-script	Either installs the specified Orchestrator package in the Orchestrator server, or runs the specified script. Mandatory attribute.

Table 1-12. Element Hierarchy

Parent Element	Child Elements
<installation>	None

finder-datasources Element

The <finder-datasources> element is the container for the <finder-datasource> elements.

The <finder-datasources> element is optional. The <finder-datasources> element has no attributes.

Table 1-13. Element Hierarchy

Parent Element	Child Elements
<module>	<finder-datasource>

finder-datasource Element

The <finder-datasource> element points to the Java class file of the IPluginAdaptor implementation that you create for the plug-in.

You set how Orchestrator accesses the objects of the plugged-in technology in the <finder-datasource> element. The <finder-datasource> element identifies the Java class of the plug-in adapter that you create. The plug-in adapter class instantiates the plug-in factory that you create. The plug-in factory defines the methods that find objects in the plugged-in technology. You can set timeouts in the <finder-datasource> element for the finder method calls that the factory performs. Different timeouts apply to the different finder methods from the IPluginFactory interface.

The <finder-datasource> element is optional. A plug-in can have an unlimited number of <finder-datasources> elements. The <finder-datasource> element has the following attributes.

Attributes	Value	Description
name	String	Identifies the data source in the <finder> element datasource attributes. Equivalent to an XML id. Mandatory attribute.
adaptor-class	Java class	Points to the IPluginAdaptor implementation you define to create the plug-in adapter, for example, com.vmware.plugins.sample.Adaptor. Mandatory attribute.
concurrent-call	true (default) or false	Allows multiple users to access the adapter at the same time. You must set concurrent-call to false if the plug-in does not support concurrent calls. Optional attribute.
invoker-mode	direct (default) or timeout	Sets a timeout on the finder function. If set to direct, calls to finder functions never time out. If set to timeout, the Orchestrator server applies the timeout period that corresponds to the finder method. Optional attribute.
anonymous-login-mode	never (default) or always	Passes or does not pass the user's username and password to the plug-in. Optional attribute.
timeout-fetch-relation	Number; default 30 seconds	Applies to calls from findRelation(). Optional attribute.
timeout-find-all	Number; default 60 seconds	Applies to calls from findAll(). Optional attribute.

Attributes	Value	Description
timeout-find	Number; default 60 seconds	Applies to calls from find(). Optional attribute.
timeout-has-children-in-relation	Number; default 2 seconds	Applies to calls from findChildrenInRelation(). Optional attribute.
timeout-execute-plugin-command	Number; default 30 seconds	Applies to calls from executePluginCommand(). Optional attribute.

Table 1-14. Element Hierarchy

Parent Element	Child Elements
<finder-datasources>	None

inventory Element

The <inventory> element defines the root of the hierarchical list for the plug-in that appears in the Orchestrator client **Inventory** view and object selection dialog boxes.

The <inventory> element does not represent an object in the plugged-in application, but rather represents the plug-in itself as an object in the Orchestrator scripting API.

The <inventory> element is optional. The <inventory> element has the following attribute.

Attributes	Value	Description
type	An Orchestrator object type	The type of the <finder> element that represents the root of the hierarchy of objects. Mandatory attribute.

Table 1-15. Element Hierarchy

Parent Element	Child Elements
<module>	None

finders Element

The <finders> element is the container for all the <finder> elements.

The <finders> element is optional. The <finders> element has no attributes.

Table 1-16. Element Hierarchy

Parent Element	Child Element
<module>	<finder>

finder Element

The <finder> element represents in the Orchestrator client a type of object found through the plug-in.

The `<finder>` element identifies the Java class that defines the object the object finder represents. The `<finder>` element defines how the object appears in the Orchestrator client interface. It also identifies the scripting object that the Orchestrator scripting API defines to represent this object.

Finders act as an interface between object formats used by different types of plugged-in technologies.

The `<finder>` element is optional. A plug-in can have an unlimited number of `<finder>` elements. The `<finder>` element defines the following attributes:

Attributes	Value	Description
type	An Orchestrator object type	Type of object represented by the finder. Mandatory attribute.
datasource	<code><finder-datasource name></code> attribute	Identifies the Java class that defines the object by using the datasource refid. Mandatory attribute.
dynamic-finder	Java method	Defines a custom finder method you implement in an <code>IDynamicFinder</code> instance, to return the ID and properties of a finder programmatically, instead defining it in the <code>vso.xml</code> file. Optional attribute.
hidden	true or false (default)	If true, hides the finder in the Orchestrator client. Optional attribute.
image	Path to a graphic file	A 16x16 icon to represent the finder in hierarchical lists in the Orchestrator client. Optional attribute.
java-class	Name of a Java class	The Java class that defines the object the finder finds and maps to a scripting object. Optional attribute.
script-object	<code><scripting-object type></code> attribute	The <code><scripting-object></code> type, if any, to which to map this finder. Optional attribute.

Table 1-17. Element Hierarchy

Parent Element	Child Elements
<code><finders></code>	<ul style="list-style-type: none"> ■ <code><id></code> ■ <code><description></code> ■ <code><properties></code> ■ <code><default-sorting></code> ■ <code><inventory-children></code> ■ <code><relations></code> ■ <code><inventory-tabs></code> ■ <code><events></code>

properties Element

The `<properties>` element is the container for `<finder>``<property>` elements.

The `<properties>` element is optional. The `<properties>` element has no attributes.

Table 1-18. Element Hierarchy

Parent Element	Child Element
<code><finder></code>	<code><property></code>

property Element

The `<property>` element maps the found object's properties to Java properties or method calls.

You can call on the methods of the `SDKFinderProperty` class when you implement the plug-in factory to obtain properties for the plug-in factory implementation to process.

You can show or hide object properties in the views in the Orchestrator client. You can also use enumerations to define object properties.

The `<property>` element is optional. A plug-in can have an unlimited number of `<property>` elements. The `<property>` element has the following attributes.

Attributes	Value	Description
<code>name</code>	Finder name	The name the <code>FinderResult</code> uses to store the element. Mandatory attribute.
<code>display-name</code>	Finder name	The displayed property name. Optional attribute.
<code>bean-property</code>	Property name	You use the <code>bean-property</code> attribute to identify a property to obtain using <code>get</code> and <code>set</code> operations. If you identify a property named <code>MyProperty</code> , the plug-in defines <code>getMyProperty</code> and <code>setMyProperty</code> operations. You set one or the other of <code>bean-property</code> or <code>property-accessor</code> , but not both. Optional attribute.
<code>property-accessor</code>	The method that obtains a property value from an object	The <code>property-accessor</code> attribute allows you to define an OGNL expression to validate an object's properties. You set one or the other of <code>bean-property</code> or <code>property-accessor</code> , but not both. Optional attribute.
<code>show-in-column</code>	true (default) or false	If true, this property shows in the Orchestrator client results table. Optional attribute.
<code>show-in-description</code>	true (default) or false	If true, this property shows in the object description. Optional attribute.

Attributes	Value	Description
hidden	true or false (default)	If true, this property is hidden in all cases. Optional attribute.
linked-enumeration	Enumeration name	Links a finder property to an enumeration. Optional attribute.

Table 1-19. Element Hierarchy

Parent Element	Child Elements
<properties>	Child Elements

relations Element

The <relations> element is the container for <finder><relation> elements.

The <relations> element is optional. The <relations> element has no attributes.

Table 1-20. Element Hierarchy

Parent Element	Child Element
<finder>	<relation>

relation Element

The <relation> element defines how objects relate to other objects.

You define the relation name in the <relation> element.

The <relation> element is optional. A plug-in can have an unlimited number of <relation> elements. The <relation> element has the following attributes.

Attributes	Value	Description
name	Relation name	A name for this relation. Mandatory attribute.
type	Orchestrator object type	The type of the object that relates to another object by this relation. Mandatory attribute.
cardinality	to-one or to-many	Defines the relation between the objects as one-to-one or one-to-many. Optional attribute.

Table 1-21. Element Hierarchy

Parent Element	Child Elements
<relations>	None

id Element

The <id> element defines a method to obtain the unique ID of the object that the finder identifies.

The <id> element is optional. The <id> element has the following attributes.

Attributes	Value	Description
accessor	Method name	The accessor attribute allows you to define an OGNL expression to validate an object's properties. Mandatory attribute.

Table 1-22. Element Hierarchy

Parent Element	Child Elements
<finder>	None

inventory-children Element

The <inventory-children> element defines the hierarchy of the lists that show the objects in the Orchestrator client **Inventory** view and object selection boxes.

The <inventory-children> element is optional. The <inventory-children> element has no attributes.

Table 1-23. Element Hierarchy

Parent Element	Child Element
<finder>	<relation-link>

relation-link Element

The <relation-link> element defines the hierarchies between parent and child objects in the **Inventory** tab.

The <relation-link> element is optional. A plug-in can have an unlimited number of <relation-link> elements. The <relation-link> element has the following attribute.

Type	Value	Description
name	Relation name	A refid to a relation name. Mandatory attribute.

Table 1-24. Element Hierarchy

Parent Element	Child Elements
<inventory-children>	None

events Element

The <events> element is the container for the <trigger> and <gauge> elements.

The <events> element can contain an unlimited number of triggers or gauges.

The <events> element is optional. The <events> element has no attributes.

Table 1-25. Element Hierarchy

Parent Element	Child Elements
<finder>	<ul style="list-style-type: none"> ■ <trigger> ■ <gauge>

trigger Element

The <trigger> element declares the triggers you can use for this finder. You must implement the `registerEventPublisher()` and `unregisterEventPublisher()` methods of `IPluginAdaptor` to set triggers.

The <trigger> element is optional. The <trigger> element has the following attribute.

Type	Value	Description
name	Trigger name	A name for this trigger. Mandatory attribute.

Table 1-26. Element Hierarchy

Parent Element	Child Elements
<events>	<ul style="list-style-type: none"> ■ <description> ■ <trigger-properties>

trigger-properties Element

The <trigger-properties> element is the container for the <trigger-property> elements.

The <trigger-properties> element is optional. The <trigger-properties> element has no attributes.

Table 1-27. Element Hierarchy

Parent Element	Child Element
<trigger>	<trigger-property>

trigger-property Element

The <trigger-property> element defines the properties that identify a trigger object.

The <trigger-property> element is optional. A plug-in can have an unlimited number of <trigger-property> elements. The <trigger-property> element has the following attributes.

Type	Value	Description
name	Trigger name	A name for the trigger. Optional attribute.
display-name	Trigger name	The name that displays in the Orchestrator client. Optional attribute.
type	Trigger type	The object type that defines the trigger. Mandatory attribute.

Table 1-28. Element Hierarchy

Parent Element	Child Elements
<trigger-properties>	None

gauge Element

The <gauge> element defines the gauges you can use for this finder. You must implement `theregisterEventPublisher()` and `unregisterEventPublisher()` methods of `IPluginAdaptor` to set gauges.

The <gauge> element is optional. A plug-in can have an unlimited number of <gauge> elements. The <gauge> element has the following attributes.

Type	Value	Description
name	Gauge name	A name for the gauge. Mandatory attribute.
min-value	Number	Minimum threshold. Optional attribute.
max-value	Number	Maximum threshold. Optional attribute.
unit	Object type	Object type that defines the gauge. Mandatory attribute.
format	String	The format of the monitored value. Optional attribute.

Table 1-29. Element Hierarchy

Parent Element	Child Element
<events>	<description>

scripting-objects Element

The <scripting-objects> element is the container for the <object> elements.

The <scripting-objects> element is optional. The <scripting-objects> element has no attributes.

Table 1-30. Element Hierarchy

Parent Element	Child Element
<module>	<object>

object Element

The <object> element maps the plugged-in technology's constructors, attributes, and methods to JavaScript object types that the Orchestrator scripting API exposes.

See [Naming Plug-In Objects](#) for object naming conventions.

The <object> element is optional. A plug-in can have an unlimited number of <object> elements. The <object> element has the following attributes.

Type	Value	Description
script-name	JavaScript name	Scripting name of the class. Must be globally unique. Mandatory attribute.
java-class	Java class	The Java class wrapped by this JavaScript class. Mandatory attribute.
create	true (default) or false	If true, you can create a new instance of this class. Optional attribute.
strict	true or false (default)	If true, you can only call methods you annotate or declare in the vso.xml file. Optional attribute.
is-deprecated	true or false (default)	If true, the object maps a deprecated Java class. Optional attribute.
since-version	String	Version since the Java class is deprecated. Optional attribute.

Table 1-31. Element Hierarchy

Parent Element	Child Elements
<scripting-objects>	<ul style="list-style-type: none"> ■ <description> ■ <deprecated> ■ <url> ■ <constructors> ■ <attributes> ■ <methods> ■ <singleton>

constructors Element

The <constructors> element is the container for the <object><constructor> elements.

The <constructors> element is optional. The <constructors> element has no attributes.

Table 1-32. Element Hierarchy

Parent Element	Child Element
<object>	<constructor>

constructor Element

The <constructor> element defines a constructor method. The <constructor> method produces documentation in the API Explorer.

The <constructor> element is optional. A plug-in can have an unlimited number of <constructor> elements. The <constructor> element has no attributes.

Table 1-33. Element Hierarchy

Parent Element	Child Elements
<constructors>	<ul style="list-style-type: none"> ■ <description> ■ <parameters>

Constructor parameters Element

The <parameters> element is the container for the <constructor><parameter> elements.

The <parameters> element is optional. The <parameters> element has no attributes.

Table 1-34. Element Hierarchy

Parent Element	Child Element
<constructor>	<parameter>

Constructor parameter Element

The <parameter> element defines the constructor's parameters.

The <parameter> element is optional. A plug-in can have an unlimited number of <parameter> elements. The <parameter> element has the following attributes.

Type	Value	Description
name	String	Parameter name to use in API documentation. Mandatory attribute.
type	Orchestrator parameter type	Parameter type to use in API documentation. Mandatory attribute.
is-optional	true or false	If true, value can be null. Optional attribute.
since-version	String	Method version. Optional attribute.

Table 1-35. Element Hierarchy

Parent Element	Child Elements
<parameters>	None

attributes Element

The <attributes> element is the container for the <object><attribute> elements.

The <attributes> element is optional. The <attributes> element has no attributes.

Table 1-36. Element Hierarchy

Parent Element	Child Element
<object>	<attribute>

attribute Element

The `<attribute>` element maps the attributes of a Java class from the plugged-in technology to JavaScript attributes that the Orchestrator JavaScript engine makes available.

The `<attribute>` element is optional. A plug-in can have an unlimited number of `<attribute>` elements. The `<attribute>` element has the following attributes.

Type	Value	Description
java-name	Java attribute	Name of the Java attribute. Mandatory attribute.
script-name	JavaScript object	Name of the corresponding JavaScript object. Mandatory attribute.
return-type	String	The type of object this attribute returns. Appears in the API Explorer documentation. Optional attribute. Note If the JavaScript return type is <code>Properties</code> , the supported underlying Java implementations are <code>java.util.HashMap</code> and <code>java.util.Hashtable</code> .
read-only	true or false	If true, you cannot modify this attribute. Optional attribute.
is-optional	true or false	If true, this field can be null. Optional attribute.
show-in-api	true or false	If false, this attribute does not appear in API documentation. Optional attribute.
is-deprecated	true or false	If true, the object maps a deprecated attribute. Optional attribute.
since-version	Number	The version at which the attribute was deprecated. Optional attribute.

Table 1-37. Element Hierarchy

Parent Element	Child Elements
<code><attributes></code>	None

methods Element

The `<methods>` element is the container for the `<object>``<method>` elements.

The `<methods>` element is optional. The `<methods>` element has no attributes.

Table 1-38. Element Hierarchy

Parent Element	Child Element
<code><object></code>	<code><method></code>

method Element

The `<method>` element maps a Java method from the plugged-in technology to a JavaScript method that the Orchestrator JavaScript engine exposes.

The `<method>` element is optional. A plug-in can have an unlimited number of `<method>` elements. The `<method>` element has the following attributes.

Type	Value	Description
java-name	Java method	Name of the Java method signature with argument types in parentheses, for example, <code>getVms(DataStore)</code> . Mandatory attribute.
script-name	JavaScript method	Name of the corresponding JavaScript method. Mandatory attribute.
return-type	Java object type	The type this method obtains. Optional attribute. Note If the JavaScript return type is <code>Properties</code> , the supported underlying Java implementations are <code>java.util.HashMap</code> and <code>java.util.Hashtable</code> .
static	true or false	If true, this method is static. Optional attribute.
show-in-api	true or false	If false, this method does not appear in API documentation. Optional attribute.
is-deprecated	true or false	If true, the object maps a deprecated method. Optional attribute.
since-version	Number	The version at which the method was deprecated. Optional attribute.

Table 1-39. Element Hierarchy

Parent Element	Child Elements
<code><methods></code>	<ul style="list-style-type: none"> ■ <code><deprecated></code> ■ <code><description></code> ■ <code><example></code> ■ <code><parameters></code>

example Element

The `<example>` element allows you to add code examples to Javascript methods that appear in the API Explorer documentation.

The `<example>` element is optional. The `<example>` element has no attributes.

Table 1-40. Element Hierarchy

Parent Element	Child Elements
<method>	<ul style="list-style-type: none"> ■ <code> ■ <description>

code Element

The <code> element provides example code that appears in the API Explorer documentation.

You provide the code example between the <code> and </code> tags. The <code> element is optional. The <code> element has no attributes.

Table 1-41. Element Hierarchy

Parent Element	Child Elements
<example>	None

Method parameters Element

The <parameters> element is the container for the <method><parameter> elements.

The <parameters> element is optional. The <parameters> element has no attributes.

Table 1-42.

Parent Element	Child Element
<method>	<parameter>

Method parameter Element

The <parameter> element defines the method's input parameters.

The <parameter> element is optional. A plug-in can have an unlimited number of <parameter> elements. The <parameter> element has the following attributes.

Type	Value	Description
name	String	Parameter name. Mandatory attribute.
type	Orchestrator parameter type	Parameter type. Mandatory attribute.
is-optional	true or false	If true, value can be null. Optional attribute.
since-version	String	Method version. Optional attribute.

Table 1-43. Element Hierarchy

Parent Element	Child Element
<parameters>	None

singleton Element

The <singleton> element creates a JavaScript scripting object as a singleton instance.

A singleton object behaves in the same way as a static Java class. Singleton objects define generic objects for the plug-in to use, rather than defining specific instances of objects that Orchestrator accesses in the plugged-in technology. For example, you can use a singleton object to establish the connection to the plugged-in technology.

The `<singleton>` element is optional. The `<singleton>` element has the following attributes.

Type	Value	Description
script-name	JavaScript object	Name of the corresponding JavaScript object. Mandatory attribute.
datasource	Java object	The source Java object for this JavaScript object. Mandatory attribute.

Table 1-44. Element Hierarchy

Parent Element	Child Element
<code><object></code>	None

enumerations Element

The `<enumerations>` element is the container for the `<enumeration>` elements.

The `<enumerations>` element is optional. The `<enumerations>` element has no attributes.

Table 1-45. Element Hierarchy

Parent Element	Child Element
<code><module></code>	<code><enumeration></code>

enumeration Element

The `<enumeration>` element defines common values that apply to all objects of a certain type.

If all objects of a certain type require a certain attribute, and if the range of values for that attribute is limited, you can define the different values as enumeration entries. For example, if a type of object requires a `color` attribute, and if the only available colors are red, blue, and green, you can define three enumeration entries to define these three color values. You define entries as child elements of the enumeration element.

The `<enumeration>` element is optional. A plug-in can have an unlimited number of `<enumeration>` elements. The `<enumeration>` element has the following attribute.

Type	Value	Description
type	Orchestrator object type	Enumeration type. Mandatory attribute.

Table 1-46. Element Hierarchy

Parent Element	Child Elements
<enumerations>	<ul style="list-style-type: none"> ■ <url> ■ <description> ■ <entries>

entries Element

The <entries> element is the container for the <enumeration><entry> elements.

The <entries> element is optional. The <entries> element has no attributes.

Table 1-47. Element Hierarchy

Parent Element	Child Element
<enumeration>	<entry>

entry Element

The <entry> element provides a value for an enumeration attribute.

The <entry> element is optional. A plug-in can have an unlimited number of <entry> elements. The <entry> element has the following attributes.

Type	Value	Description
id	Text	The identifier that objects use to set the enumeration entry as an attribute. Mandatory attribute.
name	Text	The entry name. Mandatory attribute.

Table 1-48. Element Hierarchy

Parent Element	Child Elements
<entries>	None

Best Practices for Orchestrator Plug-In Development

You can improve certain aspects of the Orchestrator plug-ins that you develop by understanding the structure and content of plug-ins, as well as by understanding how to avoid specific problems.

■ [Approaches for Building Orchestrator Plug-Ins](#)

You can use different approaches to build your Orchestrator plug-ins. You can start building a plug-in layer by layer or you can start building all layers of the plug-in at the same time.

■ [Types of Orchestrator Plug-Ins](#)

By using plug-ins, you can integrate general-purpose libraries or utilities like XML or SSH, as well as entire systems, such as vCloud Director, with Orchestrator. Depending on the technology that you integrate with Orchestrator, plug-ins can be categorized as plug-ins for services, or general purpose plug-ins, and plug-ins for systems.

■ [Plug-In Implementation](#)

You can use certain helpful practices and techniques when you structure your plug-ins, implement the required Java classes and JavaScript objects, develop the plug-in workflows and actions, as well as provide the workflow presentation.

■ [Recommendations for Orchestrator Plug-In Development](#)

Adhering to certain certain practices when developing the different components of your Orchestrator plug-ins helps you to improve the quality of the plug-ins.

■ [Documenting Plug-In User Interface Strings and APIs](#)

When you write user interface (UI) strings for Orchestrator plug-ins and the related API documentation, follow the accepted rules of style and format.

Approaches for Building Orchestrator Plug-Ins

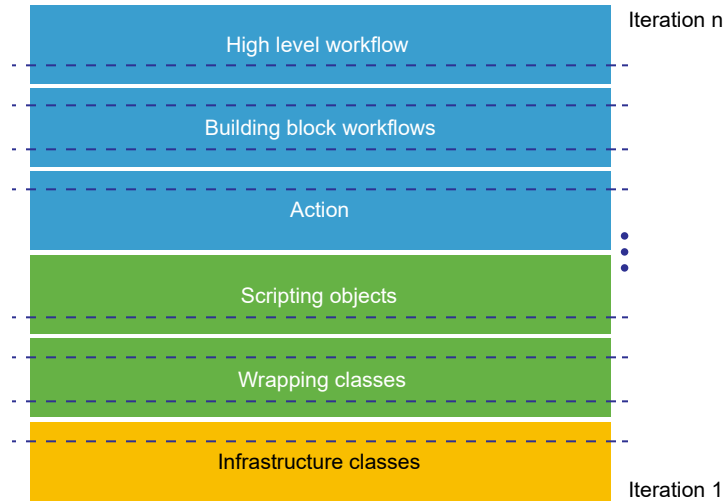
You can use different approaches to build your Orchestrator plug-ins. You can start building a plug-in layer by layer or you can start building all layers of the plug-in at the same time.

For information about plug-in layers, see [Structure of an Orchestrator Plug-In](#).

Bottom-Up Plug-In Development

A plug-in can be built layer by layer using bottom-up development approach.

Bottom-up development approach builds the plug-in layer by layer starting from the lower level layers and continuing with the higher level layers. When this approach is mixed with an interactive and iterative development approach, then part or whole layer is delivered for each iteration. At the end of the N iterations the plug-in is completely finished.

Figure 1-5. Bottom-up plug-in development

An advantage of the bottom-up plug-in development approach is that development is focused on one layer at a time.

Consider the following disadvantages of bottom-up plug-in development approach.

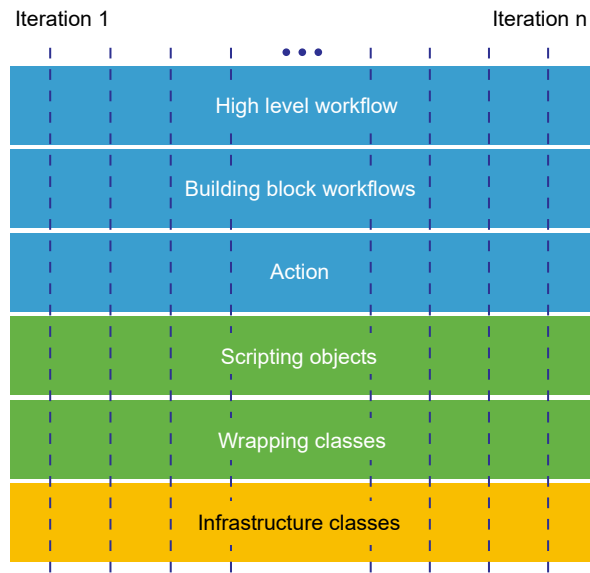
- The progress of the plug-in development is difficult to show until some insertions are completed.
- It does not fit very well in an Agile development practices.

The bottom-up development process is considered good enough for small plug-ins, with reduced or non-existent set of wrapping classes, scripting objects, actions, or workflows.

Top-Down Plug-In Development

A plug-in can be built by slicing it into top-down functionality, using top-down development approach.

When the top-down approach is mixed with an Agile development process, new functionality is delivered for each iteration. As a result, at the end of the iteration N the plug-in is completely implemented.

Figure 1-6. Top-down plug-in development

The top-down plug-in development approach has the following advantages.

- The progress of the plug-in development is easy to show from the first iteration because new functionality is completed for each iteration and the plug-in can be released and used after every iteration.
- Completing a vertical slice of functionality allows for very clearly defined success criteria and definition of what has been done, as well as better communication between developers, product management, and quality assurance (QA) engineers.
- Allows the QA engineers to start testing and automating from the beginning of the development process. Such an approach results in valuable feedback and decreases the overall project delivery time frame.

A disadvantage of the top-down plug-in development approach is that the development is in progress on different layers at the same time.

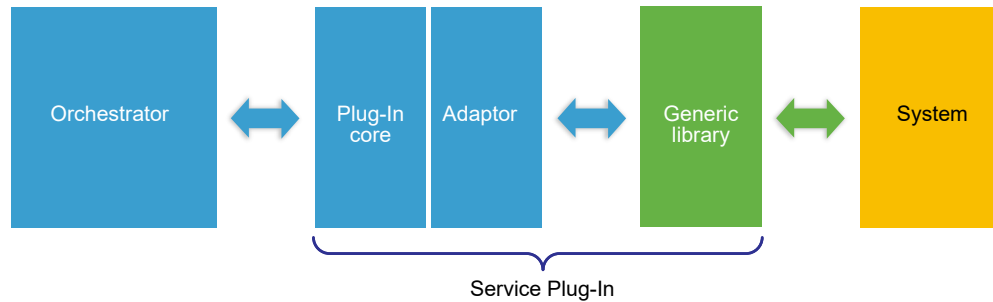
You should apply the top-down plug-in development process for most plug-ins. It is appropriate for plug-ins with dynamic requirements.

Types of Orchestrator Plug-Ins

By using plug-ins, you can integrate general-purpose libraries or utilities like XML or SSH, as well as entire systems, such as vCloud Director, with Orchestrator. Depending on the technology that you integrate with Orchestrator, plug-ins can be categorized as plug-ins for services, or general purpose plug-ins, and plug-ins for systems.

Plug-Ins for Services

Plug-ins for services or general-purpose plug-ins provide functionality that can be considered as a service inside Orchestrator.

Figure 1-7. Architecture of plug-ins for services

Plug-ins for services expose generic libraries or utilities to Orchestrator, such as XML, SSH, or SOAP. For example, the following plug-ins that are available in Orchestrator are plug-ins for services.

JDBC plug-in

Lets you use any database within a workflow.

Mail plug-in

Lets you send emails within a workflow.

SSH plug-in

Lets you open SSH connections and run commands within a workflow.

XML plug-in

Lets you manage XML documents within a workflow.

Plug-ins for services have the following characteristics.

Complexity

Plug-ins for services have low to medium levels of complexity. Plug-ins for services expose a specific library, or part of a library, inside Orchestrator so as to provide concrete functionality. For example, the XML plug-in adds an implementation of a Document Object Model (DOM) XML parser to the Orchestrator JavaScript API.

Size

Plug-ins for services are relatively small in size. They require the same basic set of classes as for all plug-ins, and other classes that offer new scripting objects to add new functionality.

Inventory

Plug-ins for services require a small inventory of objects to work, or they do not require an inventory at all. Plug-ins for services have a generic and small object model, and so, they do not need to show this model inside the Orchestrator inventory.

Plug-Ins for Systems

Plug-ins for systems connect the Orchestrator workflow engine to an external system so that you can orchestrate the external system.

Following are examples for plug-ins for systems.

vCenter Server plug-in

Lets you manage vCenter Server instances using workflows.

vCloud Director plug-in

Lets you interact with a vCloud Director installation within a workflow.

Cisco UCSM plug-in

Lets you interact with Cisco entities within a workflow.

Following are the main characteristics of plug-ins for systems.

Complexity

Plug-ins for systems have a higher level of complexity than general-purpose plug-ins, because the technologies that they expose are relatively complex. Plug-ins for systems must represent all the elements of the external system inside Orchestrator to interact with the external system and offer its functionality in Orchestrator. If the external system provides an integration mechanism, you can use it to expose the functionality of the system in Orchestrator more easily. However, besides representing the elements of the external system in Orchestrator, plug-ins for systems might also need to offer high scalability, provide a caching mechanism, deal with events and notifications, and so on.

Size

Plug-ins for system are medium to big in size. Plug-ins for systems require many classes apart from the basic set of classes because usually they offer a large number of scripting objects. Plug-ins for systems might require some other helper and auxiliary classes that will interact with them.

Inventory

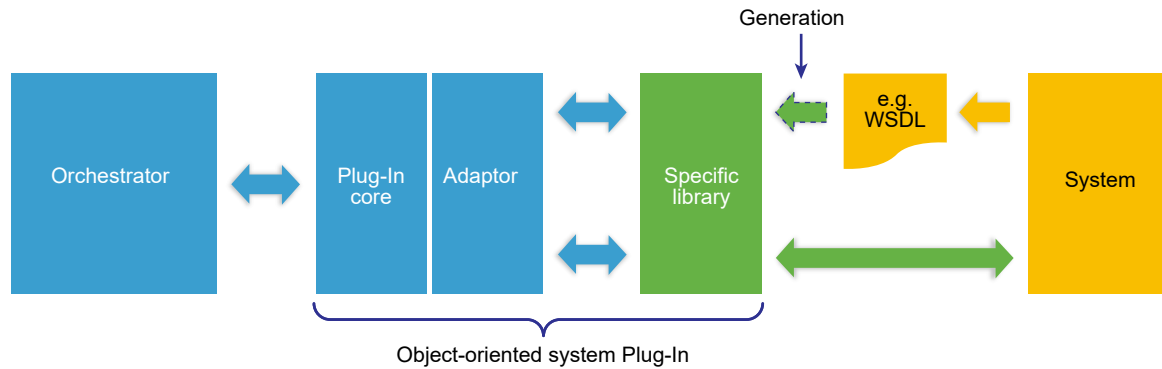
Usually, plug-ins for systems have a large number of objects, and you must expose these objects properly in the inventory so that you can locate them and work with them easily in Orchestrator. Because of the large number of objects that plug-ins for systems need to expose, you should build auxiliary tool or a process to auto-generate as much code as possible for the plug-in. For example, the vCenter Server plug-in provides such a tool.

Plug-Ins for Object-Oriented Systems

Object-oriented systems offer an interaction mechanism that is based on objects and RPC.

The most widely used model for an object-oriented system is the Web service model that uses SOAP. The objects inside this model have a set of attributes that are related to the state of the objects and offer a set of remote methods that are invoked on the target system side.

Figure 1-8. Plug-Ins for Object-Oriented Systems



You can consider the following when you implement plug-ins for object-oriented systems.

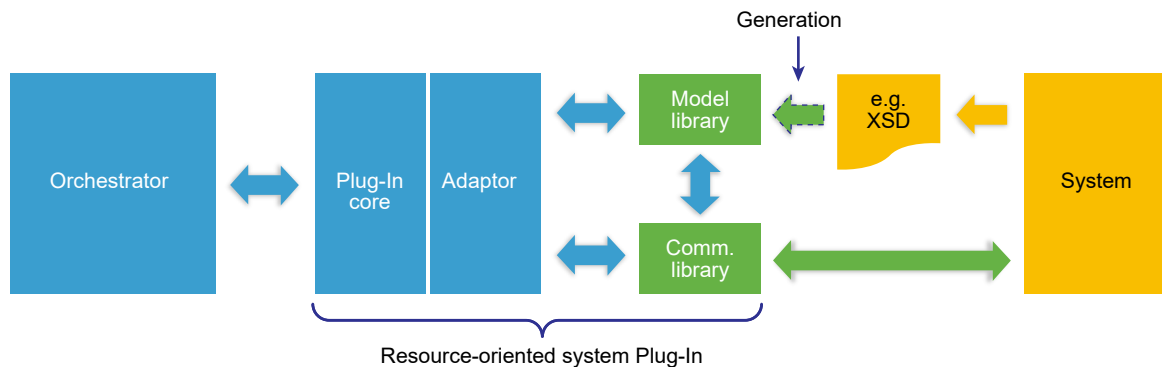
- If you use SOAP, you can use the WSDL file to generate a set of classes that combine the object model and the communication mechanism.
- This object model is almost everything that you have to expose inside Orchestrator.

Plug-Ins for Resource-Oriented Systems

Resource-oriented systems provide an interaction mechanism that is based on resources and simple operations that use HTTP methods.

The most representative model for a resource-oriented system is the REST model, combined for example with XML. The objects inside this model have a set of attributes that are related to their state. To invoke methods on the target system (communication mechanism), you must use the standard HTTP methods such as GET, POST, PUT, and so on, and follow some conventions.

Figure 1-9. Plug-ins for resource-oriented systems



You can consider the following when you develop plug-ins for resource-oriented systems.

- If you use REST or only HTTP with XML, you get one or more XML schema files to be able to read and write messages. From these schemas, you can generate a set of classes that define the object model. This set of classes only defines the state of the objects because the operations are defined implicitly with the HTTP methods, for example, as defined in the vCloud Director plug-in, or explicitly with some specific XML messages, such as the Cisco UCSM plug-in.
- You need to implement the communication mechanism in another set of classes. This set of classes defines a new object model that interacts with the original object model. The object model for the communication mechanism consists of objects and methods only.
- You can expose both the original object model and the object model for the communication mechanism inside Orchestrator. This might add some complexity depending on how both object models are exposed, and on whether you are merging related objects from both sides (to simulate an object-oriented system) or keeping them separate.

Plug-In Implementation

You can use certain helpful practices and techniques when you structure your plug-ins, implement the required Java classes and JavaScript objects, develop the plug-in workflows and actions, as well as provide the workflow presentation.

- [Project Structure](#)

You can apply a standard structure for the projects of your Orchestrator plug-ins.

- [Project Internals](#)

You can apply certain approaches when implementing your plug-in, for example, cache objects, bring objects in background, clone objects, and so on. By following such approaches, you can improve the performance of your plug-ins, avoid concurrency problems, and improve the responsiveness of the Orchestrator client.

- [Workflow Internals](#)

You can implement a workflow to monitor long-time operations that your Orchestrator plug-in performs.

- [Workflows and Actions](#)

To ease the workflow development and usage, you can use certain good practices.

- [Workflow Presentation](#)

When you create the presentation of a workflow, you should apply certain structure and rules.

Project Structure

You can apply a standard structure for the projects of your Orchestrator plug-ins.

You can use a standard Maven structure with modules for your plug-in projects to bring clarity in where every piece of functionality resides.

Table 1-49. Structure of a Plug-In Project

Module	Description
/myAwesomePlugin-plugin	The root of the plug-in project.
/o11nplugin-myAwesomePlugin	The module that composes the final plug-in DAR file.
/o11nplugin-myAwesomePlugin-config	The module that contains the plug-in configuration Web application. It generates a standard WAR file.
/o11nplugin-myAwesomePlugin-core	The module that contains all the classes that implement any of the standard Orchestrator plug-in interfaces and other auxiliary classes that they use. It generates a standard JAR file.
/o11nplugin-myAwesomePlugin-model	The module that contains all the classes that help you integrate the third-party technology with Orchestrator through the plug-in. The classes should not contain any direct reference to the standard Orchestrator plug-in APIs.
/o11nplugin-myAwesomePlugin-package	The module that imports an external Orchestrator package file with actions and workflows to include it inside the final plug-in DAR file. The module is optional.

Project Internals

You can apply certain approaches when implementing your plug-in, for example, cache objects, bring objects in background, clone objects, and so on. By following such approaches, you can improve the performance of your plug-ins, avoid concurrency problems, and improve the responsiveness of the Orchestrator client.

Cache Objects

Your plug-in can interact with a remote service, and this interaction is provided by local objects that represent remote objects on the service side. To achieve good performance of the plug-in as well as good responsiveness of the Orchestrator UI, you can cache the local objects instead of getting them every time from the remote service. You can consider the scope of the cache, for example, one cache for all the plug-in clients, one cache per user of the plug-in, and one cache per user of the third-party service. When implemented, your caching mechanism is integrated with the plug-in interface for finding and invalidating objects.

Bring Objects in Background

If you have to show large lists of objects in the plug-in inventory and do not have a fast way to retrieve those objects, you can bring objects in background. You can bring object in background, for example, by having objects with two states, fake and loaded. Assume that the fake objects are very easy to create and provide the minimal information that you have to show in the inventory, such as name and ID. Then it would be possible to always return fake objects, and when all the information (the real object) is really needed, the using entity or the plug-in can invoke a method load automatically to get the real object. You can even configure the process of loading objects to start automatically after the fake objects are returned, to anticipate the actions of the using entity.

Clone Objects to Avoid Concurrency Problems

If you use a cache for your plug-in, you have to clone objects. Use of a cache that always returns the same instance of an object to every entity that requests it can have unwanted effects. For example, entity A requests object O, and the entity views the object in the inventory with all its attributes. At the same time, entity B requests object O as well, and entity A runs a workflow that starts changing the attributes of object O. At the end of its run, the workflow invokes the object's update method to update the object on the server side. If entity A and entity B get the same instance of object O, entity A views in the inventory all the changes that entity B performs, even before the changes are committed on the server side. If the run goes fine, it should not be a problem, but if the run fails, the attributes of object O for entity A are not reverted. In such a case, if the cache (the find operations of the plug-in) returns a clone of the object instead of the same instance all the time, each using entity views and modifies its own copy, avoiding concurrency issues, at least within Orchestrator.

Notify Changes to Others

Problems might occur when you use a cache and clone objects simultaneously. The biggest one is that the object that is using entity views might not be the latest version that is available for the object. For example, if an entity displays the inventory, the objects are loaded once, but at the same time, if another entity is changing some of the objects, the first entity does not view the changes. To avoid this problem, you can use the `PluginWatcher` and `IPluginPublisher` methods from the Orchestrator plug-in API to notify that something has changed to allow other instances of Orchestrator clients to see the changes. This also applies to a unique instance of the Orchestrator client when changes from one object from the inventory affect other objects of the inventory, and they need to be notified too. The operations that are prone to use notifications are adding, updating, and deleting objects when these objects, or some properties of these objects, are shown in the inventory.

Enable Finding Any Object at Any Time

You must implement the `find` method of the `IPluginFactory` interface to find objects just by type and ID. The `find` method can be invoked directly after restarting Orchestrator and resuming a workflow.

Simulate a Query Service if You Do Not Have One

The Orchestrator client can require querying for some objects in specific cases or showing them not as a tree but as a list or a table, for example. This means that your plug-in must be able to query for some set of objects at any moment. If the third-party technology offers a query service, you need to adapt and use this service. Otherwise, you should be able to simulate a query service, despite of the higher complexity or the lower performance of the solution.

Find Methods Should Not Return Runtime Exceptions

The methods from the `IPluginFactory` interface that implement the searches inside the plug-in should not throw controlled or uncontrolled runtime exceptions. This might be the cause of strange *validation error* failures when a workflow is running. For example, between two nodes of a workflow, the `find` method is invoked if an output from the first node is an input of the second node. At that moment, if the object is not found because of any runtime exception, you might get no more information than a *validation error* in the Orchestrator client. After that, it depends on how the plug-in logs the exceptions in to get more or less information inside the log files.

Workflow Internals

You can implement a workflow to monitor long-time operations that your Orchestrator plug-in performs.

You can implement a workflow for monitoring long-time running operations such as task monitoring. This workflow can be based on Orchestrator triggers and waiting events. You must consider that a workflow that is blocked waiting for a task can be resumed as soon as the Orchestrator server starts. The plug-in must be able to get all the required information to resume the monitoring process properly.

The monitoring workflow or the task that it can use internally should provide a mechanism to specify the polling rate and a possible timeout.

The process of debugging a piece of scripting code inside a workflow is not easy, especially if the code does not invoke any Java code. Because of this, sometimes the only option is to use the logging methods offered by the default Orchestrator scripting objects.

Workflows and Actions

To ease the workflow development and usage, you can use certain good practices.

Start Developing Workflows as Building Blocks

A building block can be a simple workflow that requires a few input parameters and returns a simple output. If you have a rich set of building blocks, you can create higher-level workflows easily, and you can offer a better set of tools for composing complex workflows.

Create Higher-Level Workflows Based on Smaller Components

If you have to develop a complex workflow with several inputs and internal steps, you can split it into smaller and simpler building block workflows and actions.

Create Actions Whenever Possible

You can create actions to achieve additional flexibility when you develop workflows.

- To create complex objects or parameters for scripting methods easily
- To avoid repeating common pieces of code all the time
- To perform UI validations

Workflows Should Invoke Actions Whenever Possible

Actions can be invoked directly as nodes inside the workflow schema. This can keep the workflow schema simpler, because you do not need to add scripting code blocks to invoke a single action.

Fill In the Expected Information

Provide information for every element of a workflow or an action.

- Provide a description of the workflow or action.
- Provide a description of the input parameters.
- Provide a description of the outputs.
- Provide a description of the attributes for the workflows.

Keep the Version Information Updated

When you version plug-ins, add meaningful comments with information such as major updates to the plug-in, important implementation details, and so on.

Workflow Presentation

When you create the presentation of a workflow, you should apply certain structure and rules.

Use the following properties for the workflow inputs in the workflow presentation.

Table 1-50. Properties for Workflow Inputs

Properties	Usage
Show in Inventory	Use this property to help the user to run a workflow from the inventory view.
Specify a root object to be shown in the chooser	Use this property to help the user to select inputs. If the root object can be refreshed in the presentation, is an attribute, or is retrieved by an object method, you need to create or set an appropriate action to refresh the object in the presentation.
Maximum string length	Use this property for long strings such as names, descriptions, file paths, and so on.
Minimum string length	Use this property to avoid empty strings from the testing tools.
Custom validation	Implement non-simple validations with actions.

Organize the inputs with steps and display group. Such organization helps the user identify and distinguish all the input parameters of a workflow.

Recommendations for Orchestrator Plug-In Development

Adhering to certain certain practices when developing the different components of your Orchestrator plug-ins helps you to improve the quality of the plug-ins.

Table 1-51. Useful Practices in Plug-In Implementation

Component	Item	Description
General	Access to third-party API	Plug-ins should provide simplified methods for accessing the third-party API wherever possible.
	Interface	Plug-ins should provide a coherent and standard interface for users, even when the API does not.
Action	Scripting objects	You should create actions for every creation, modification, deletion, and all other methods available for a scripting object.
	Description	The description of an action should describe what the action does instead of how it works.
	Scripting	When you use scripting to get the properties or methods of an object, you can check whether the object value is different from null or undefined.
	Deprecation	If an action is deprecated, the <code>comment</code> or the <code>throw</code> statement should indicate the replacement action, or the action should call a new replacement action so that solutions that are built on the deprecated version of the action do not fail.
Workflow	User interface operations in the orchestrated technology	You should create a workflow for every operation that is available in the user interface of the orchestrated technology.
	Description	The description of a workflow should describe what the workflow does instead of how it works.
	Presentation property <code>mandatory input</code>	You must set the <code>mandatory input</code> property for all mandatory workflow inputs.
	Presentation property <code>default value</code>	If you develop a workflow that configures an entity, the workflow presentation should load the default configuration values for this entity. For example, if you develop a workflow that is named Host Configuration, the presentation of the workflow must load the default values of the host configuration.
	Presentation property <code>Show in inventory</code>	You must set the <code>Show in inventory</code> property so that you have contextual workflows on inventory objects.
	Presentation property <code>specify a root parameter</code>	You should use this property in workflows when it is not necessary to browse the inventory from the tree root .
	Workflow validation	You must validate workflows and fix all errors.
	Object creation	All workflows that create a new object should return the new object as an output parameter.
	Deprecation	If a workflow is deprecated, the <code>comment</code> or the <code>throw</code> statement should indicate the replacement workflow, or the deprecated workflow should call a new replacement workflow to ensure that solutions that are built on previous versions of the workflow do not fail.

Table 1-51. Useful Practices in Plug-In Implementation (continued)

Component	Item	Description
Inventory	Host disconnection	If your inventory contains a connection to a host and this host becomes unavailable, you should indicate that the host is disconnected. You can do this either by renaming the root object by appending – disconnected or by removing the tree of objects underneath this object, in the same manner as the vCloud Director plug-in does.
	Select value as list property	An inventory object must be selectable as treeview or a list.
	Host manager	If the plug-in implements a host object for the target system, then a parent <code>hostmanager</code> root object should exist with properties for adding, removing, or editing host properties.
	Getting or updating objects	If a query service is running on the orchestrated technology, you should use it for getting multiple objects.
	Child discovery	If you need to retrieve child objects separately, the retrieval process must be multithreaded and non-blocking on a single error.
	Orchestrator object change	All workflows that can change the state of an element in the inventory must update the inventory to avoid having objects out of synchronization.
	External object change	You can use a notification mechanism to notify about changes in the orchestrated technology that occur as a result of operations that are performed outside of Orchestrator. In case such operations lead to removal of objects from the orchestrated technology, you must refresh the inventory accordingly to avoid failures or loss of data. For example, if a virtual machine is deleted from vCenter Server, the vCenter Server plug-in updates the inventory to remove the object of the removed virtual machine.
Scripting object	Finder object	Finder objects should have properties that can be used to differentiate objects. These are typically the properties that are present in the user interface.
	Implementation	The <code>equals</code> method must be implemented to insure that <code>==</code> operation works on the same object as in some cases the object might have two instances.
	Plug-in object properties	Objects that have parent objects should implement a parent property.
	Plug-in object properties	Objects that have child objects should implement GET methods that return arrays of child objects.
	Inventory objects	Inventory objects should be searchable with <code>Server.find</code> . All inventory objects should be serializable so they can be used as input or output attributes in a workflow.
	Constructor and methods	In most cases, scriptable objects should have either a constructor, or should be returned by other object attributes or methods.

Table 1-51. Useful Practices in Plug-In Implementation (continued)

Component	Item	Description
	Object ID	Objects that have an ID that is issued from an external system should use an internal ID to ensure that no ID duplication occurs when you are orchestrating more than one server.
	Searching for objects	search or find methods should implement a filter so that the specified name or ID can be found instead of just all objects. For example, the Orchestrator server has a <code>Server.FindById</code> method that allows finding a plug-in object by its ID. To do this, the method must be implemented for each findable object in the plug-in.
	Trigger	If possible, triggers should be available for objects that change so that Orchestrator can have policies triggered on various events. For example, to determine when a new virtual machine is added, powered on, powered off, and so on, Orchestrator can monitor a trigger or an event in the vCenter plug-in on the Datacenter object.
	Object properties	Objects that reside in other plug-ins should have properties for being easily converted from one plug-in object to another. For example, virtual machine objects need to have a <code>moref</code> (managed object reference ID).
	Session manager	If you are connecting to a remote server that can have a different session, the plug-in should implement a shared session and a session per user.
Trigger	Trigger	All long operations and blocking methods should be able to start asynchronously with a task returned, and generate a trigger event on completion.
Enumerations	Enums	Enumerations for a given type should have an inventory object that allows selecting from the different values in the enumeration.
Logging	Logs	Methods should implement different log levels.
Versioning	Plug-in version	The plug-in version should follow standards and be updated along with the plug-in update.
API documentation	Methods	Methods that are described in the API documentation should never throw the exception <code>no xyz method / property</code> on an object. Instead, methods should return <code>null</code> when no properties are available and be documented with details when these properties are not available.
	<code>vso.xml</code>	All objects, methods, and properties must be documented in <code>vso.xml</code> .

Documenting Plug-In User Interface Strings and APIs

When you write user interface (UI) strings for Orchestrator plug-ins and the related API documentation, follow the accepted rules of style and format.

General Recommendations

- Use the official names for VMware products involved in the plug-in. For example, use the official names for the following products and VMware terminology.

Correct Term	Do Not Use
vCenter Server	VC or vCenter
vCloud Director	vCloud

- End all workflow descriptions with a period. For example, `Creates a new Organization.` is a workflow description.
- Use a text editor with a spell checker to write the descriptions and then move them to the plug-in.
- Ensure that the name of the plug-in exactly matches the approved third-party product name that it is associated with.

Workflows and Actions

- Write informative descriptions. One or two sentences are enough for most of the actions and workflows.
- Higher-level workflows might include more extensive descriptions and comments.
- Start descriptions with a verb, for example, `Creates....` Do not use self-referential language like `This workflow creates.`
- Put a period at the end of descriptions that are complete sentences.
- Describe what a workflow or action does instead of how it is implemented.
- Workflows and actions usually are included in folders and packages. Include a small description for these folders and packages as well. For example, a workflow folder can have a description similar to `Set of workflows related to vApp Template management.`

Parameters of Workflows and Actions

- Start workflow and action descriptions with a descriptive noun phrase, for example, `Name of.` Do not use a phrase like `It's the name of.`
- Do not put a period at the end of parameter and action descriptions. They are not complete sentences.
- Input parameters of workflows must specify a label with appropriate names in the presentation view. In many cases, you can combine related inputs in a display group. For example, instead of having two inputs with the labels `Name of the Organization` and `Full name of the Organization`, you can create a display group with the label `Organization` and place the inputs `Name` and `Full name` in the `Organization` group.
- For steps and display groups, add descriptions or comments that appear in the workflow presentation as well.

Plug-In API

- The documentation of the API refers to all of the documentation in the `vso.xml` file and the Java source files.
- For the `vso.xml` file, use the same rules for the descriptions of finder objects and scripting objects with their methods that you use for workflows and actions. Descriptions of object attributes and method parameters use the same rules as the workflow and action parameters.
- Avoid special characters in the `vso.xml` file and include the descriptions inside a `<![CDATA[insert your description here!]]>` tag.
- Use the standard Javadoc style for the Java source files.

Obtaining Input Parameters from Users When a Workflow Starts

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

The way you organize parameters in the **Presentation** tab translates into the input parameters dialog box when the workflow runs.

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

■ [Creating the Input Parameters Dialog Box In the Presentation Tab](#)

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

■ [Setting Parameter Properties](#)

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

Creating the Input Parameters Dialog Box In the Presentation Tab

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

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

Presentation Descriptions

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

Defining Presentation Input Steps

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

Defining Presentation Display Groups

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

Create the Presentation of the Input Parameters Dialog Box

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

Prerequisites

- Open a workflow for editing in the workflow editor.
- Verify that the workflow has a defined list of input parameters.

Procedure

- 1 In the workflow editor, click the **Presentation** tab.

By default, all of the workflow's parameters appear under the main **Presentation** node in the order in which you create them.

- 2 Right-click the **Presentation** node and select **Create new step**.

A **New Step** node appears under the **Presentation** node.

- 3 Provide an appropriate name for the step and press Enter.

This name appears as a section header in the input parameters dialog box when the workflow runs.

- 4 Click the input step and add a description in the **General** tab in the bottom half of the **Presentation** tab.

This description appears in the input parameters dialog box to provide information to the users to help them provide the correct input parameters. You can enhance the layout of the description text by using HTML formatting.

- 5 Right-click the input step you created and select **Create display group**.

A **New Group** node appears under the input step node.

- 6 Provide an appropriate name for the display group and press Enter.

This name appears as a subsection header in the input parameters dialog box when the workflow runs.

- 7 Click the display group and add a description in the **General** tab in the bottom half of the **Presentation** tab.

This description appears in the input parameters dialog box. You can enhance the layout of the description text by using HTML formatting. You can add a parameter value to a group description by using an OGNL statement, such as `${#param}`.

- 8 Repeat the preceding steps until you have created all the input steps and display groups to appear in the input parameters dialog box when the workflow runs.
- 9 Drag parameters from under the **Presentation** node to the steps and groups of your choice.

Results

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

What to do next

You must set the parameter properties.

Setting Parameter Properties

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

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

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

Static and Dynamic Parameter Property Values

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

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

Set Parameter Properties

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

Prerequisites

- Open a workflow for editing in the workflow editor.
- Verify that the workflow has a defined list of input parameters.

Procedure

- 1 In the workflow editor, click the **Presentation** tab.

- 2 Click a parameter in the **Presentation** tab.

The parameter's **General** and **Properties** tabs appear at the bottom of the **Presentation** tab.

- 3 Click the parameter's **Properties** tab.



- 4 Right-click in the **Properties** tab and select **Add property**.

A dialog box opens, presenting a list of the possible properties for a parameter of the type selected.

- 5 Select a property from the list presented in the dialog box and click **OK**.

The property appears in the **Properties** tab.



- 6 Under **Value**, make the property value either static or dynamic by selecting the corresponding symbol from the drop-down menu.

Option	Description
	Static property
	Dynamic property

- 7 If you set the property value to static, you select a property value according to the type of parameter for which you are setting the properties.

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

The workflow editor provides help writing the OGNL expression.

- a Click the  icon to obtain a list of all the attributes and parameters defined by the workflow that this expression can call upon.
- b Click the  icon to obtain a list of all the actions in the Orchestrator API that return an output parameter of the type for which you are defining the properties.

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

- 9 Click **Save** at the bottom of the workflow editor.

Results

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

What to do next

Validate and debug the workflow.

Workflow Input Parameter Properties

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

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

Parameter Property	Parameter Type	Description
Maximum string length	String	Sets a maximum length for the parameter.
Minimum string length	String	Sets a minimum length for this parameter.
Matching regular expression	String	Validates the input using a regular expression.
Maximum number value	Number	Sets a maximum value for the parameter.
Minimum number value	Number	Sets a minimum value for the parameter.
Number format	Number	Formats the input for the parameter.
Mandatory Input	All simple types	Makes the parameter mandatory.
Predefined answers	All simple types	Predefines a list of possible values for the property as an array of simple types. You either define the array manually or the property calls an action that returns an array of objects of the appropriate type.

Parameter Property	Parameter Type	Description
Predefined list of elements	Any simple or complex types	Predefines a list of possible values for the property as an array of simple or complex types. Calls an action that returns an array of objects of the appropriate type.
Show parameter input	Any simple or complex types	Shows or hides a parameter text box in the presentation dialog box, depending on the value of a preceding Boolean parameter.
Hide parameter input	Any simple or complex types	Similar to Show parameter input , but takes the negative value of a previous Boolean parameter.
Matching expression	Any parameter type obtained from a plug-in	The input parameter matches a given expression.
Show in inventory	Any parameter type obtained from a plug-in	If set, you can run the present workflow on any object of this type by right-clicking it in the inventory view and selecting Run workflow .
Specify a root object to be shown in the chooser. Root object is provided from a parameter or attribute.	Any parameter type obtained from a plug-in	Specifies the root object if the selector for this parameter is a hierarchical list selector.
Select as	Any parameter type obtained from a plug-in	Use a list or hierarchical list selector to select the parameter.
Default value	Any simple or complex types	Default value for this parameter.
Custom validation	OGNL scriptable validation	If the OGNL expression returns a string, the validation shows this string as the text of the error result.
Data binding	Any simple or complex types	Binds to a property that you have already defined in another parameter.
Authorized only	Any parameter type obtained from a plug-in	Only authorized users can access this parameter.
Multi-lines text input	Any simple or complex types	Allows users to enter multiple lines of text in the input parameters dialog box.

Predefined Constant Values for OGNL Expressions

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

Orchestrator defines the following constants for use in OGNL expressions.

Table 1-52. Predefined OGNL Constant Values

Constant Value	Description
<code>\${#__current}</code>	Current value of the custom validation property or matching expression property
<code>\${#__username}</code>	User name of the user who started the workflow
<code>\${#__userdisplayname}</code>	Display name of the user who started the workflow
<code>\${#__serverurl}</code>	URL containing the IP address of the server from which the user starts the workflow. The URL consists of the server IP address and a lookup port: <code>{ServerIP}:{lookupPort}</code>
<code>\${#__datetime}</code>	Current date and time
<code>\${#__date}</code>	Current date, with time set to 00:00:00
<code>\${#__timezone}</code>	Current timezone

Requesting User Interactions While a Workflow Runs

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

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

The default attributes for user interactions are `security.group` and `timeout.date`. When you set the `security.group` attribute to a given LDAP user group, you limit the permission to respond to the user interaction request to members of that user group.

When you set the `timeout.date` attribute, you set a time and date until which the workflow waits for the information from the user. You can set an absolute date, or you can create a scripted workflow element to calculate a time relative to the current time.

Procedure

1 (Optional) Add a User Interaction to a Workflow

You request input parameters from users during a workflow run by adding a **User interaction** schema element to the workflow. When a workflow encounters a **User interaction** element, it suspends its run and waits for the user to provide the data that it requires.

2 (Optional) Set the User Interaction security.group Attribute

The `security.group` attribute of a user interaction element sets which users or groups of users have permission to respond to the user interaction.

3 (Optional) Set the timeout.date Attribute to an Absolute Date

You set the `timeout.date` attribute for a user interaction to set how long the workflow waits for a user to respond to a user interaction.

4 (Optional) Calculate a Relative Timeout for User Interactions

You can calculate in a `Date` object a relative time and date at which a user interaction times out.

5 (Optional) Set the timeout.date Attribute to a Relative Date

You can set the `timeout.date` attribute of a **User Interaction** element to a relative time and date by binding it to a `Date` object. You define the object in a scripted function.

6 (Optional) Define the External Inputs for a User Interaction

You specify the information that users must provide during a workflow run as the input parameters of a user interaction.

7 (Optional) Define User Interaction Exception Behavior

If a user does not provide the input parameters within the timeout period, the user interaction returns an exception. You can define the exception behavior in a scripted function.

8 (Optional) Create the Input Parameters Dialog Box for the User Interaction

Users provide input parameters during a workflow run in an input parameters dialog box, in the same way that they provide input parameters when a workflow first starts.

9 (Optional) Respond to a Request for a User Interaction

Workflows that require interactions from users during their run suspend their run either until the user provides the required information or until the workflow times out.

Add a User Interaction to a Workflow

You request input parameters from users during a workflow run by adding a **User interaction** schema element to the workflow. When a workflow encounters a **User interaction** element, it suspends its run and waits for the user to provide the data that it requires.

Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.
- Add some elements to the workflow schema.

Procedure

- 1 Drag a **User interaction** element to the appropriate position in the workflow schema.

- 2 Click the **Edit** icon (✎) of the **User interaction** element.
- 3 Provide a name and a description for the user interaction in the **Info** tab and click **Close**.
- 4 Click **Save**.

Results

You added a user interaction element to a workflow. When the workflow reaches this element, it waits for information from the user before continuing its run.

What to do next

Set the `security.group` attribute of the user interaction to limit permission to respond to the user interaction to a user or user group. See [Set the User Interaction security.group Attribute](#).

Set the User Interaction security.group Attribute

The `security.group` attribute of a user interaction element sets which users or groups of users have permission to respond to the user interaction.

Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.
- Add some elements and a user interaction to the workflow schema.
- Identify an LDAP user group to respond to the user interaction request.

Procedure

- 1 Click the **Edit** icon (✎) of the **User Interaction** element in the workflow schema.
- 2 Click the **Attributes** tab for the user interaction.
- 3 Click **Not set** for the `security.group` source parameter to set which users can respond to the user interaction.
- 4 (Optional) Select **NULL** to allow all users to respond to the request for user interaction.
- 5 To limit the permission to respond to a specific user or user group, click **Create parameter/attribute in workflow**.

The **Parameter information** dialog box opens.

- 6 Name the parameter.
- 7 Select **Create workflow ATTRIBUTE with the same name** to create the `LdapGroup` attribute in the workflow.
- 8 Click **Not set** for the parameter value to open the **LdapGroup** selection box.
- 9 Type the name of the LDAP user group in the **Filter** text box.

- 10 Select the LDAP user group from the list and click **Select**.

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

You limited the permission to respond to the user interaction request.

- 11 Click **OK** to close the **Parameter information** dialog box.

Results

You set the `security.group` attribute for the user interaction.

What to do next

Set the `timer.date` attribute to set the timeout period for the user interaction.

- To set the timeout to an absolute date and time, see [Set the timeout.date Attribute to an Absolute Date](#).
- To create a function to calculate a timeout that is relative to the current date and time, see [Calculate a Relative Timeout for User Interactions](#).

Set the timeout.date Attribute to an Absolute Date

You set the `timeout.date` attribute for a user interaction to set how long the workflow waits for a user to respond to a user interaction.

You set an absolute time and date in the `Date` object. When the time on the given date arrives, the workflow that is waiting for a user interaction times out and ends in the `Failed` state. For example, you can set the user interaction to timeout at midday on February 12th. To calculate a timeout that is relative to the current time and date, see [Calculate a Relative Timeout for User Interactions](#).

Prerequisites

- Open a workflow for editing in the workflow editor.
- Add a user interaction element to the workflow schema.
- Set the `security.group` attribute for the user interaction.

Procedure

- 1 Click the **Edit** icon (✎) of the **User Interaction** element in the workflow schema.
- 2 Click the **Attributes** tab for the user interaction.
- 3 Click **Not set** for the `timeout.date` source parameter to set the timeout parameter value.
- 4 (Optional) Select **NULL** to allow the user interaction to set the workflow to wait indefinitely for the user to respond to the user interaction.

- 5 Click **Create parameter/attribute in workflow** to set the workflow to fail after a timeout period.
The **Parameter information** dialog box opens.
- 6 Name the parameter.
- 7 Select **Create workflow ATTRIBUTE with the same name** to create a Date attribute in the workflow.
- 8 Click **Not set** for the parameter **Value**.
- 9 Use the calendar to select an absolute date and time until which the workflow waits for the user to respond.
- 10 Click **OK** to close the calendar.
- 11 Click **OK** to close the **Parameter information** dialog box.

Results

You set the `timeout.date` attribute to an absolute date. The workflow times out if the user does not respond to the user interaction before this time and date.

What to do next

Define the external input parameters that the user interaction requires from the user. See [Define the External Inputs for a User Interaction](#).

Calculate a Relative Timeout for User Interactions

You can calculate in a Date object a relative time and date at which a user interaction times out.


You can set an absolute time and date in a Date object. When the time on the given date arrives, the request for a user interaction times out. Alternatively, you can create a workflow element that calculates and generates a relative Date object according to a function that you define. For example, you can create a relative Date object that adds 24 hours to the current time.

Prerequisites

- Open a workflow for editing in the workflow editor.
- Add a user interaction element to the workflow schema.
- Set the `security.group` attribute for the user interaction.

Procedure

- 1 Drag a **Scriptable task** element from the **Generic** menu to the schema of a workflow, before the element that requires the relative Date object for its `timeout.date` attribute.
- 2 Click the **Edit** icon (✎) of the **Scriptable task** element in the workflow schema.
- 3 Provide a name and description for the scripted workflow element in the **Info** properties tab.

- 4 Click the **OUT** properties tab, and click the **Bind to workflow parameter/attribute** icon ().
- 5 Click **Create parameter/attribute in workflow** to create a workflow attribute.
 - a Name the attribute `timerDate`.
 - b Select Date from the list of attribute types.
 - c Select **Create workflow ATTRIBUTE with the same name**.
 - d Leave the attribute value set to **Not set**, because a scripted function will provide this value.
 - e Click **OK**.
- 6 Click the **Scripting** tab for the scripted workflow element.
- 7 Define a function to calculate and generate a Date object named `timerDate` in the scripting pad in the **Scripting** tab.

For example, you can create a Date object by implementing the following JavaScript function, in which the timeout period is a relative delay in milliseconds.

```
timerDate = new Date();
System.log( "Current date : '" + timerDate + "'" );
timerDate.setTime( timerDate.getTime() + (86400 * 1000) );
System.log( "Timer will expire at '" + timerDate + "'" );
```

The preceding example JavaScript function defines a Date object that obtains the current date and time by using the `getTime` method and adds 86,400,000 milliseconds, or 24 hours. The **Scriptable Task** element generates this value as its output parameter.

- 8 Click **Close**.
- 9 Click **Save**.

Results

You created a function that calculates a time and date relative to the current time and date and generates a Date object. A **User Interaction** element can receive this Date object as an input parameter to set the timeout period until which it waits for input from the user. When the workflow arrives at the **User Interaction** element, it suspends its run and waits either until the user provides the required information, or for 24 hours before it times out.

What to do next

You must bind the Date object to the **User Interaction** element's `timeout.date` parameter. See [Set the timeout.date Attribute to a Relative Date](#).

Set the timeout.date Attribute to a Relative Date

You can set the `timeout.date` attribute of a **User Interaction** element to a relative time and date by binding it to a Date object. You define the object in a scripted function.

If you create a relative Date object in a scripted function, you can bind the `timeout.date` attribute of a user interaction to this Date object. For example, if you bind the `timeout.date` attribute to a Date object that adds 24 hours to the current time, the user interaction times out after waiting for 24 hours.

Prerequisites

- Add a user interaction element to the workflow schema.
- Set the `security.group` attribute for the user interaction.
- Create a scripted function that calculates a relative time and date and encapsulates it in a Date object in the workflow. See [Calculate a Relative Timeout for User Interactions](#).

Procedure

- 1 Click the **Edit** icon (✎) of the **User Interaction** element in the workflow schema.
- 2 Click the **Attributes** tab for the user interaction.
- 3 Click **Not set** for the `timeout.date` source parameter to set the timeout parameter value.
- 4 Select the Date object that encapsulates a relative time and date that you defined in a scripted function and click **Select**.

Results

You set the `timeout.date` attribute to a relative date and time that a scripted function calculates.

What to do next

Define the external input parameters that the user interaction requires from the user. See [Define the External Inputs for a User Interaction](#).

Define the External Inputs for a User Interaction

You specify the information that users must provide during a workflow run as the input parameters of a user interaction.


When a workflow reaches a user interaction element, it waits until a user provides the information that the user interaction requires as its input parameters.

Prerequisites

- Add a user interaction element to the workflow schema.
- Set the `security.group` attribute for the user interaction.
- Set the `timer.date` attribute for the user interaction

Procedure

- 1 Click the **Edit** icon (✎) of the **User Interaction** element in the workflow schema.
- 2 Click the **External inputs** tab.

- 3 Click the **Bind to workflow parameter/attribute** icon () to define the parameters that the user must provide in the user interaction.
- 4 (Optional) If you already defined the input parameters in the workflow, select the parameters from the proposed list.
- 5 Click **Create parameter/attribute in workflow** to create a workflow attribute to bind to the input parameter that the user provides.
- 6 Give the parameter an appropriate name.
- 7 Select the input parameter type from the list of types by searching for an object type in the **Filter** box.

For example, if the user interaction requires the user to provide a virtual machine as an input parameter, select VC:VirtualMachine.
- 8 Select **Create workflow ATTRIBUTE with the same name** to bind the input parameter that the user provides to a new attribute in the workflow.
- 9 Leave the input parameter value set to **Not set**.

The user provides this value when they respond to the user interaction during the workflow run.
- 10 Click **OK** to close the **Parameter information** dialog box.

Results

You defined the input parameters that the user provides during a user interaction.

What to do next

Define the exception behavior if the user interaction encounters an error. See [Define User Interaction Exception Behavior](#).

Define User Interaction Exception Behavior

If a user does not provide the input parameters within the timeout period, the user interaction returns an exception. You can define the exception behavior in a scripted function.

If you do not define the action for the workflow to take if the user interaction times out, the workflow ends in the **Failed** state. Defining the exception behavior is a good workflow development practice.

Prerequisites

- Add a user interaction element to the workflow schema.
- Set the `security.group` and `timer.date` attributes for the user interaction.
- Define the external input parameters of the user interaction.

Procedure

- 1 Click the **Edit** icon (✎) of the **User Interaction** element in the workflow schema.
- 2 Click the **Exception** tab.
- 3 Click **Not set** for the output exception binding.
- 4 Click **Create parameter/attribute in workflow** to create an exception attribute to which to bind the user interaction.

The **Parameter information** dialog box opens.

- 5 Create an `errorCode` attribute.

An `errorCode` attribute has the following default properties:

- Name: **errorCode**
- Type: string
- Create: **Create workflow ATTRIBUTE with the same name**
- Value: Type an appropriate error message.

- 6 Click **OK** to close the **Parameter information** dialog box.
- 7 Drag a scriptable task element over the user interaction element in the workflow schema.
A red dashed arrow, which represents the exception link, appears between the two elements. The scriptable task element binds automatically to the `errorCode` attribute from the user interaction.

- 8 Double-click the scriptable task element and provide an appropriate name.

For example, **Log timeout**.

- 9 In the **Scripting** tab of the scriptable task element, write a JavaScript function to handle the exception.

For example, to record the timeout in the Orchestrator log, write the following function:

```
System.log("No response from user. Timed out.");
```

- 10 Link and bind the scriptable task element that handles exceptions to the element that follows it in the workflow.

For example, link and bind the scriptable task element to a **Throw exception** element to end the workflow with an error.

Results

You defined the exception behavior if the user interaction times out.

What to do next

Create the dialog box in which users provide input parameters. See [Create the Input Parameters Dialog Box for the User Interaction](#).

Create the Input Parameters Dialog Box for the User Interaction

Users provide input parameters during a workflow run in an input parameters dialog box, in the same way that they provide input parameters when a workflow first starts.

You create the layout of the dialog box in the **Presentation** tab of the user interaction element, not in the **Presentation** tab for the whole workflow. The **Presentation** tab of the whole workflow creates the layout of the input parameters dialog box that appears when you start a workflow. The **Presentation** tab of the user interaction element creates the layout of the input parameters dialog box that opens when a workflow arrives at a user interaction element during its run.

Prerequisites

- Add a user interaction element to the workflow schema.
- Set the `security.group` and `timer.date` attributes for the user interaction.
- Define the external input parameters of the user interaction.
- Define the exception behavior.

Procedure

- 1 Click the **Edit** icon (✎) of the **User Interaction** element in the workflow schema.
- 2 Click the **Presentation** tab of the user interaction element.

The **Presentation** tab shows the external input parameters that you created for the user interaction.

- 3 (Optional) Right-click the **Presentation** node in the **Presentation** tab and select **Create new step**.

Steps allow you to create sections in the dialog box, with descriptions and headings under which you can organize the input parameters.

- 4 (Optional) Right-click the **Presentation** node in the **Presentation** tab and select **Create display group**.

Display groups allow you to sort the order in which input parameters appear in the steps, and allow you to add sub-headers and instructions to the dialog box.

- 5 Click an input parameter in the list and add a description of the input parameter in the **General** tab for that parameter.

The description text that you type appears as a label in the input parameters dialog box to inform the user of the information they must provide when they respond to the user interaction.

6 Define input parameter properties.

Input parameter properties allow you to qualify the input parameter values that users can provide, and to determine parameter values dynamically by using OGNL expressions.

7 Click **Save and close** to close the workflow editor.

Results

You created the input parameters dialog box in which users provide input parameters to respond to a user interaction during a workflow run.

What to do next

For information about creating the presentation steps and groups and setting input parameter properties, see [Creating the Input Parameters Dialog Box In the Presentation Tab](#).

Respond to a Request for a User Interaction

Workflows that require interactions from users during their run suspend their run either until the user provides the required information or until the workflow times out.

Workflows that require user interactions define which users can provide the required information and direct the requests for interaction.

Prerequisites

Verify that at least one workflow is in the Waiting for User Interaction state.

Procedure

1 From the drop-down menu in the Orchestrator Legacy Client, select **Run**.

2 Click the **My Orchestrator** view in the Orchestrator Legacy Client.

3 Click the **Waiting for Input** tab.

The **Waiting for Input** tab lists the workflows that are waiting for user inputs.

4 Double-click a workflow that is waiting for input.

The workflow token that is waiting for input appears in the **Workflows** hierarchical list with the following symbol: .

5 Right-click the workflow token and select **Answer**.

6 Follow the instructions in the input parameters dialog box and provide the information that the workflow requires.

Results

You provided information to a workflow that was waiting for user input during its run.

Calling Workflows Within Workflows

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

- **Workflow Elements that Call Workflows**

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

- **Call a Workflow Synchronously**

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

- **Call a Workflow Asynchronously**

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

- **Schedule a Workflow**

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

- **Prerequisites for Calling a Remote Workflow from Within Another Workflow**

If the workflow that you develop calls another workflow that resides on a remote Orchestrator server, certain prerequisites must be fulfilled so that the remote workflow can run successfully.

- **Call Several Workflows Simultaneously**

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

Workflow Elements that Call Workflows

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

Synchronous Workflows

A workflow can start another workflow synchronously. The called workflow runs as an integral part of the calling workflow's run, and runs in the same memory space as the calling workflow. The calling workflow starts another workflow, then waits until the end of the called workflow's run before it starts running the next element in its schema. Usually, you call a workflow synchronously because the calling workflow requires the output of the called workflow as an input parameter for a subsequent schema element. For example, a workflow

can call the Start virtual machine and wait workflow to start a virtual machine, and then obtain the IP address of this virtual machine to pass to another element or to a user by email.

Asynchronous Workflows

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

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

Scheduled Workflows

A workflow can call a workflow but defer starting that workflow until a later time and date. The calling workflow then continues its run until it ends. Calling a scheduled workflow creates a task to start that workflow at the given time and date. When the calling workflow has run, you can view the scheduled workflow in the **Scheduler** and **My Orchestrator** views in the Orchestrator client.

Scheduled workflows only run once. You can schedule a workflow to run recurrently by calling the `Workflow.scheduleRecurrently` method in a scriptable task element in a synchronous workflow.

Nested Workflows

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

Propagate Workflow Changes to other Workflows

If you call a workflow from another workflow, Orchestrator imports the input parameters of the child workflow in the parent workflow at the moment you add the workflow element to the schema.

If you modify the child workflow after you have added it to another workflow, the parent workflow calls on the new version of the child workflow, but does not import any new input parameters. To prevent changes to workflows affecting the behavior of other workflows that call them, Orchestrator does not propagate the new input parameters automatically to the calling workflows.

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

Prerequisites

Verify that you have a workflow that another workflow or workflows call.

Procedure

- 1 Modify and save a workflow that other workflows call.
- 2 Close the workflow editor.
- 3 Navigate to the workflow you changed in the hierarchical list in the **Workflows** view in the Orchestrator client.
- 4 Right-click the workflow, and select **References > Find Elements that Use this Element**.
A list of workflows that call this workflow appears.
- 5 Double-click a workflow in the list to highlight it in the **Workflows** view in the Orchestrator client.
- 6 Right-click the workflow, and select **Edit**.
The workflow editor opens.
- 7 Click the **Schema** tab in the workflow editor.
- 8 Right-click the workflow element for the changed workflow from the workflow schema and select **Synchronize > Synchronize Parameters**.
- 9 Select **Continue** in the confirmation dialog box.
- 10 Save and close the workflow editor.
- 11 Repeat [Step 5](#) to [Step 10](#) for all the workflows that use the modified workflow.

Results

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

Propagate the Input Parameters and Presentation of a Child Workflow to the Parent Workflow

If you develop a workflow that calls other workflows, you can propagate the input parameters and the presentation of the child workflows to the parent workflow.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Run**.
- 2 Right-click the workflow that you want to modify and select **Edit**.
The workflow editor opens.
- 3 Select the **Schema** tab.
- 4 Right-click the element of the child workflow whose input parameters and presentation you want to propagate to the parent workflow and select **Synchronize > Synchronize Presentation**.
- 5 In the confirmation dialog, select **OK**.
- 6 (Optional) Repeat [Step 4](#) and [Step 5](#) for all child workflows whose input parameters and presentation you want to propagate to the parent workflow.

Results

The input parameters of the child workflows are added to the input parameters of the parent workflow. The presentation of the parent workflow is extended with the presentations of the child workflows.

Call a Workflow Synchronously


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

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

Prerequisites

- Open a workflow for editing in the workflow editor.
- Add some elements to the workflow schema.

Procedure

- 1 Drag a **Workflow** element from the **Generic** menu to the appropriate position in the workflow schema.
The **Choose workflow** selection dialog box appears.
- 2 Search for and select the workflow you want and click **OK**.
If the search returns a partial result, narrow your search criterion or increase the number of search results from the **Tools > User preferences** menu in the client.
- 3 Click the **Workflow** element to show its properties tabs in the bottom half of the **Schema** tab.
- 4 Click the **Edit** icon () of the **Workflow** element in the workflow schema.

- 5 Bind the required input parameters to the workflow in the **IN** tab of the workflow schema element.
- 6 Bind the required output parameters to the workflow in the **OUT** tab of the workflow schema element's.
- 7 Define the exception behavior of the workflow in the **Exceptions** tab.
- 8 Click **Close**.
- 9 Click **Save** at the bottom of the workflow editor.

Results

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

What to do next

You can call a workflow asynchronously from a workflow.

Call a Workflow Asynchronously

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

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

Prerequisites

- Open a workflow for editing in the workflow editor.
- Add some elements to the workflow schema.

Procedure

- 1 Drag an **Asynchronous Workflow** element from the **Generic** menu to the appropriate position in the workflow schema.

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

- 2 Search for and select the desired workflow from the list and click **OK**.
- 3 Click the **Edit** icon (✎) of the **Asynchronous Workflow** element in the workflow schema.
- 4 Bind the required input parameters to the workflow in **IN** tab of the asynchronous workflow element.
- 5 Bind the required output parameter in the **OUT** tab of the asynchronous workflow element.

You can bind the output parameter either to the called workflow, or to that workflow's result.

- Bind to the called workflow to return that workflow as an output parameter

- Bind to the workflow token of the called workflow to return the result of running the called workflow.
- 6 Define the exception behavior of the asynchronous workflow element in the **Exceptions** tab.
 - 7 Click **Close**.
 - 8 Click **Save** at the bottom of the workflow editor.

Results

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

What to do next

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

Schedule a Workflow

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

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

Prerequisites

- Open a workflow for editing in the workflow editor.
- Add some elements to the workflow schema.

Procedure

- 1 Drag a **Schedule Workflow** element from the **Generic** menu to the appropriate position in the workflow schema.
- 2 Search for the workflow to call by typing part of its name in the text box.
- 3 Select the workflow from the list and click **OK**.
- 4 Click the **Edit** icon (✎) of the **Schedule Workflow** element in the workflow schema.
- 5 Click the **IN** property tab.
A parameter named workflowScheduleDate appears in the list of properties to define, together with the input parameters of the calling workflow.
- 6 Click **Not set** for the workflowScheduleDate parameter to set the parameter.
- 7 Click **Create parameter/attribute in workflow** to create the parameter and set the parameter value.
- 8 Click **Not set** for **Value** to set the parameter value.
- 9 Use the calendar that appears to set the date and time to start the scheduled workflow and click **OK**.

- 10 Bind the remaining input parameters to the scheduled workflow in the **IN** tab of the scheduled workflow element.
- 11 Bind the required output parameters to the Task object in the **OUT** tab of the scheduled workflow element.
- 12 Define the exception behavior of the scheduled workflow element in the **Exceptions** tab.
- 13 Click **Close**.
- 14 Click **Save** at the bottom of the workflow editor.

Results

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

What to do next

You can call multiple workflows simultaneously from a workflow.

Prerequisites for Calling a Remote Workflow from Within Another Workflow

If the workflow that you develop calls another workflow that resides on a remote Orchestrator server, certain prerequisites must be fulfilled so that the remote workflow can run successfully.

- All input parameters of the remote workflow must be resolvable on the remote Orchestrator server.
- All output parameters of the remote workflow must be resolvable on the local Orchestrator server.

To ensure that the parameters of the remote workflow are resolvable, the inventory objects that the workflow uses must be available both in the remote and the local Orchestrator servers. In case the remote workflow uses objects from a plug-in, the same plug-in must be available on both Orchestrator servers. The inventories of the remote plug-in and the local plug-in must be identical. In case the remote workflow uses system objects in Orchestrator, like workflows and actions, the same workflows and actions must exist in the inventories of the remote and the local Orchestrator servers.

For example, suppose that you insert the Rename virtual machine workflow in a Nested Workflow element in the Test workflow that you develop. You want to run the Rename virtual machine workflow in a remote Orchestrator server. When you run the Test workflow, the Rename virtual machine workflow is called within the run of the Test workflow. You specify a virtual machine to rename from the inventory of the local Orchestrator server. Because the Rename virtual machine workflow runs on the remote Orchestrator server, the same virtual machine must be available in the inventory of that server. Otherwise, the Rename virtual machine workflow cannot resolve its `vm` input parameter. Therefore, the vCenter Server plug-in on the local and the remote Orchestrator servers must be connected to the same vCenter Server instance.

Call Several Workflows Simultaneously

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

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

Prerequisites

- Open a workflow for editing in the workflow editor.
- Add some elements to the workflow schema.

Procedure

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

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

- 2 Search for and select a workflow to start and click **OK**.

- 3 Click the **Edit** icon (✎) of the **Nested Workflows** element in the workflow schema.

- 4 Click the **Workflows** tab.

The workflow you selected in [Step 2](#) appears in the tab.

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

- 6 Click the **Connection Info** tab in the right panel of the **Workflows** schema element properties tab.

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

- 7 To access workflows on a remote server, select **Remote** and click **Not set** to provide a host name or IP address for the remote server.

Note You can use the vRealize Orchestrator Multi-Node plug-in to call workflows on a remote server.

- 8 Define the credentials with which to access the remote server.

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

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

10 Repeat [Step 2](#) to [Step 8](#) to define the settings for each of the workflows you add.

11 Click the nested workflow element in the workflow schema.

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

Results

You called several workflows simultaneously from a workflow.

What to do next

You can define long-running workflows.

Running a Workflow on a Selection of Objects

You can automate repetitive tasks by running a workflow on a selection of objects. For example, you can create a workflow that takes a snapshot of all the virtual machines in a virtual machine folder, or you can create a workflow that powers off all the virtual machines on a given host.

You can use one of the following methods to run a workflow on a selection of objects.

- Run the **Library > vCenter > Batch > Run a workflow on a selection of objects** workflow.
- Create a workflow that calls the **Library > Orchestrator > Start workflows in a series** or **Start workflows in parallel** workflows.
- Create a workflow that obtains an array of objects and runs a workflow on each object in the array in a loop of workflow elements.
- Run a workflow from JavaScript by calling the `Workflow.execute()` method in a For loop in a scripted element in a workflow.

Which method you choose to run a workflow on a selection of objects depends on the workflow to run and can affect the performance of the workflow. For example, running the Run a workflow on a selection of objects workflow is the simplest way to run a workflow on multiple objects and requires no workflow development, but it can only run workflows that take a single input parameter.

Creating a workflow that calls the Start workflows in a series or Start workflows in parallel workflows allows you to run on multiple objects workflows that take more than one input parameter. The calling workflow must create a properties array to pass the input parameters to the Start workflows in a series or Start workflows in parallel workflow. These workflows are only for use in other workflows. Do not run them directly.

Running a workflow in a For loop in a scripted element is faster than running a workflow in a loop of workflow elements, but it is less flexible and limits the potential for reuse. Most importantly, running a workflow in a scripted loop loses the checkpointing that Orchestrator performs when it starts each element in a workflow run. As a consequence, if the Orchestrator server stops while the scripted loop is running, when the server restarts, the workflow will resume at the beginning of the scripted element, repeating the whole loop. If the Orchestrator server stops while running a workflow with a loop of workflow elements, the workflow will resume at the specific element in the loop that was running when the server stopped.

For more information about the Batch workflows, see *Using VMware vRealize Orchestrator Plug-Ins*.

How to create a workflow that runs a workflow on an array of objects in a loop of workflow elements is demonstrated in [Develop a Complex Workflow](#).

How to run a workflow in a scripted For loop is demonstrated in [Workflow Scripting Examples](#).

Implement the Start Workflows in a Series and Start Workflows in Parallel Workflows

You can use the Start workflows in a series and Start workflows in parallel workflows to run a workflow on a selection of objects.

You cannot run the Start workflows in a series and Start workflows in parallel workflows directly. You must include them in another workflow that you create. To use the Start workflows in a series and Start workflows in parallel workflows to run a workflow on a selection of objects, you must obtain the objects on which to run the workflow. You pass these objects and any other input parameters that the workflow requires to the workflow as an array of properties. The Start workflows in a series and Start workflows in parallel workflows emit the results of running the workflow on the selection of objects as an array of WorkflowToken objects.

You implement the Start workflows in a series and Start workflows in parallel workflows in the same way. The Start workflows in a series workflow runs the workflow on each object sequentially. The Start workflows in parallel workflow runs the workflow on all the objects simultaneously.

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

- 1 In the workflow schema, add a scriptable task element or an action to obtain a list of objects on which to run the workflow.

For example, to run a workflow on all the virtual machines in a virtual machine folder, you can add the `getAllVirtualMachinesByFolder` action to the workflow.

- 2 Link the scripted element or action and bind the input and output of the scripted element or action to workflow inputs or attributes.

For example, you can bind the `vmFolder` input of the `getAllVirtualMachinesByFolder` action to a workflow input parameter and the `actionResult` output to a workflow attribute in the calling workflow.

- 3 Add a scriptable task element to cast the list of objects into a properties array.

For example, if the objects on which to run the workflow are an array of virtual machines, `allVMs`, returned by the `actionResult` output of the `getAllVirtualMachinesByFolder` action, you can write the following script to cast the objects into a properties array.

```
propsArray = new Array();

for each (var vm in allVMs) {
    var prop = new Properties();
    prop.put("vm", vm);
    propsArray.push(prop);
}
```

- 4 Bind the inputs and outputs of the scriptable task element to workflow attributes.

In the example scriptable task element in [Step 3](#), you bind the input to the `allVMs` array of virtual machines and you create the `propsArray` output attribute as an array of `Properties` objects.

- 5 Add a workflow element to the workflow schema.
- 6 Select either of the Start workflows in a series or Start workflows in parallel workflows and link the workflow element to the other elements.
- 7 Bind the `wf` input of the Start workflows in a series or Start workflows in parallel workflow to the workflow to run on the objects.

For example, to remove any snapshots of all the virtual machines returned by the `getAllVirtualMachinesByFolder` action, select the Remove all snapshots workflow.

- 8 Bind the `parameters` input of the Start workflows in a series or Start workflows in parallel workflow to the array of `Properties` objects that contains the objects on which to run the workflow.

For example, bind the `parameters` input to the `propsArray` attribute defined in [Step 4](#).

- 9 (Optional) Bind the `workflowTokens` output of the Start workflows in a series or Start workflows in parallel workflow to an attribute in the workflow.
- 10 (Optional) Continue adding more elements that use the results of running the Start workflows in a series or Start workflows in parallel workflow.

Results

You created a workflow that uses either of the Start workflows in a series or Start workflows in parallel workflows to run a workflow on a selection of objects.

Developing Long-Running Workflows

A workflow in a waiting state consumes system resources because it constantly polls the object from which it requires a response. If you know that a workflow will potentially wait for a long time before it receives the response it requires, you can add long-running workflow elements to the workflow.

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

You set the duration of the wait in one of the following ways:

- Set a timer, encapsulated in a `Date` object, that suspends the workflow until a certain time and date. You implement long-running workflow elements that are based on a timer by including a **Waiting Timer** element in the schema.
- Define a trigger event, encapsulated in a `Trigger` object, that restarts the workflow after the trigger event occurs. You implement long-running workflow elements that are based on a trigger by adding a **Waiting Event** element or a **User Interaction** element in the schema.

Set a Relative Time and Date for Timer-Based Workflows

You can set the `timer.date` attribute of a **Waiting Timer** element to a relative time and date by binding it to a `Date` object. You define the `Date` object in a scripted function.


When the time on the given date arrives, the long-running workflow that is based on a timer reactivates and continues its run. For example, you can set the workflow to reactivate at midday on February 12. Alternatively, you can create a workflow element that calculates and generates a relative `Date` object according to a function that you define. For example, you can create a relative `Date` object that adds 24 hours to the current time.

Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.
- Add some elements to the workflow schema.

Procedure

- 1 Drag a **Scriptable task** element from the **Generic** menu to the schema of a workflow, before the element that requires the relative `Date` object for its `timeout.date` attribute.
- 2 Click the **Edit** icon (✎) of the **Scriptable task** element in the workflow schema.
- 3 Provide a name and description for the scripted workflow element in the **Info** properties tab.

- 4 Click the **OUT** properties tab, and click the **Bind to workflow parameter/attribute** icon (.
- 5 Click **Create parameter/attribute in workflow** to create a workflow attribute.
 - a Name the attribute `timerDate`.
 - b Select Date from the list of attribute types.
 - c Select **Create workflow ATTRIBUTE with the same name**.
 - d Leave the attribute value set to **Not set**, because a scripted function will provide this value.
 - e Click **OK**.
- 6 Click the **Scripting** tab for the scripted workflow element.
- 7 Define a function to calculate and generate a Date object named `timerDate` in the scripting pad in the **Scripting** tab.

For example, you can create a Date object by implementing the following JavaScript function, in which the timeout period is a relative delay in milliseconds.

```
timerDate = new Date();
System.log( "Current date : '" + timerDate + "'" );
timerDate.setTime( timerDate.getTime() + (86400 * 1000) );
System.log( "Timer will expire at '" + timerDate + "'" );
```

The preceding example JavaScript function defines a Date object that obtains the current date and time by using the `getTime` method and adds 86,400,000 milliseconds, or 24 hours. The **Scriptable Task** element generates this value as its output parameter.

- 8 Click **Close**.
- 9 Click **Save**.

Results

You created a function that calculates and generates a Date object. A **Waiting Timer** element can receive this Date object as an input parameter, to suspend a long-running workflow until the date encapsulated in this object. When the workflow arrives at the **Waiting Timer** element, it suspends its run and waits for 24 hours before continuing.

What to do next

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

Create a Timer-Based Long-Running Workflow

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

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


Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.
- Add some elements to the workflow schema.

Procedure

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

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

- 2 Click the **Edit** icon () of the **Waiting Timer** element in the workflow schema.
- 3 Provide a description of the reason for implementing the timer in the **Info** properties tab.
- 4 Click the **Attributes** properties tab.

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

- 5 Click the `timer.date` parameter's **Not set** button to bind the parameter to an appropriate Date object.

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

- Select a predefined Date object from the proposed list, for example one defined by a **Scriptable Task** element elsewhere in the workflow.
 - Alternatively, create a Date object that sets a specific date and time for the workflow to await.
- 6 (Optional) Create a Date object that sets a specific date and time that the workflow awaits.
 - a Click **Create parameter/attribute in workflow** in the **Waiting Timer** selection dialog box.
The **Parameter information** dialog box appears.
 - b Give the parameter an appropriate name.
 - c Leave the type set to Date.
 - d Click **Create workflow ATTRIBUTE with the same name**.
 - e Click the **Value** property's **Not set** button to set the parameter value.
A calendar appears.
 - f Use the calendar to set a date and time at which to restart workflow.
 - g Click **OK**.

- 7 Click **Close**.

8 Click **Save** at the bottom of the workflow editor.

Results

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

What to do next

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

Create a Trigger Object

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

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

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

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

Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.
- Add some elements to the workflow schema.
- In the workflow, declare a VC:Task object as an attribute or input parameter, such as a VC:Task object from a workflow or workflow element that starts or clones a virtual machine.

Procedure

- 1 Drag a **Scriptable Task** element from the **Generic** menu into the schema of a workflow.

One of the elements that precedes the **Scriptable Task** must generate a VC:Task object as its output parameter.

- 2 Click the **Edit** icon (✎) of the **Scriptable task** element in the workflow schema.
- 3 Provide a name and description for the trigger in **Info** properties tab.
- 4 Click the **IN** properties tab.

- 5 Click the **Bind to workflow parameter/attribute** icon ()

The input parameter selection dialog box opens.

- 6 Select or create an input parameter of the type **VC:Task**.

This VC:Task object represents the time-consuming event that another workflow or element launches.

- 7 (Optional) Select or create an input parameter of the Number type to define a timeout period in seconds.

- 8 Click the **OUT** properties tab.

- 9 Click the **Bind to workflow parameter/attribute** icon ()

The output parameter selection dialog box opens.

- 10 Create an output parameter with the following properties.

- a Create the Name property with the value `trigger`.
- b Create the Type property with the value `Trigger`.
- c Click **Create ATTRIBUTE with same name** to create the attribute.
- d Leave the value as **Not set**.

- 11 Define any exception behavior in the **Exceptions** properties tab.

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

For example, you could create a Trigger object by implementing the following JavaScript function.

```
trigger = task.createEndOfTaskTrigger(timeout);
```

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

- 13 Click **Close**.

- 14 Click **Save** at the bottom of the workflow editor.

Results

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

What to do next

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

Create a Trigger-Based Long-Running Workflow

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

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

Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.
- Add some elements to the workflow schema.
- Define a trigger event that is encapsulated in a Trigger object.

Procedure

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

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

- 2 Click the **Edit** icon (✎) of the **Waiting Event** element in the workflow schema.
- 3 Provide a description of the reason for the wait in the **Info** properties tab.
- 4 Click the **Attributes** properties tab.

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

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

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

- 6 Select a predefined Trigger object from the proposed list.

This Trigger object represents a trigger event that another workflow or workflow element defines.

- 7 Define any exception behavior in the **Exceptions** properties tab.
- 8 Click **Close**.

- 9 Click **Save** at the bottom of the workflow editor.

Results

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

What to do next

You can run a workflow.

Configuration Elements

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

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

If you create a package containing a workflow, action or policy that uses an attribute from a configuration element, Orchestrator automatically includes the configuration element in the package. If you import a package containing a configuration element into another Orchestrator server, you can import the configuration element attribute values as well. For example, if you create a workflow that requires attribute values that depend on the Orchestrator server on which it runs, setting those attributes in a configuration element lets you to export that workflow so that another Orchestrator server can use it. Configuration elements therefore allow you to exchange workflows, actions and policies between servers more easily.

Note You cannot import values of a configuration element attribute from a configuration element exported from Orchestrator 5.1 or earlier.

Create a Configuration Element

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

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Configurations** view.
- 3 Right-click a folder in the hierarchical list of folders and select **New folder** to create a folder.
- 4 Enter a name for the folder and click **Ok**.
- 5 Right-click the folder you created and select **New element**.

- 6 Enter a name for the configuration element and click **Ok**.

The configuration element editor opens.

- 7 Increment the version number by clicking the version digits in the **General** tab and providing a version comment.
- 8 Provide a description of the configuration element in the **Description** text box in the **General** tab.
- 9 Click the **Attributes** tab.
- 10 Click the **Add attribute** icon (A+) to create an attribute.
- 11 Click the attribute values under **Name**, **Type**, **Value**, and **Description** to set the attribute name, type, value, and description.
- 12 To save your new configuration element and close the configuration editor, click **Save and close**.

Results

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

What to do next

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

Validating Workflows

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

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

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

Workflow validation checks the data bindings and connections between elements. Workflow validation does not check the data processing that each element in the workflow performs. Consequently, a valid workflow can run incorrectly and produce erroneous results if a function in a schema element is incorrect.

By default, Orchestrator always performs workflow validation when you run a workflow. You can change the default validation behavior in the Orchestrator client. See [Testing Workflows During Development](#). For example, sometimes during workflow development you might want to run a workflow that you know to be invalid, for testing purposes.

Validate a Workflow and Fix Validation Errors

You must validate a workflow before you can run it. You can validate workflows in either the Orchestrator client or in the workflow editor. However, you can only fix validation errors if you have opened the workflow for editing in the workflow editor.

Prerequisites

Verify that you have a complete workflow to validate, with schema elements linked and bindings defined.

Procedure

1 Click the **Workflows** view.

2 Navigate to a workflow in the **Workflows** hierarchical list.

3 (Optional) Right-click the workflow and select **Validate workflow**.

If the workflow is valid, a confirmation message appears. If the workflow is invalid, a list of errors appears.

4 (Optional) Close the Workflow Validation dialog box.

5 Right-click the workflow and select **Edit** to open the workflow editor.

6 Click the **Schema** tab.

7 Click the **Validate** button in the **Schema** tab toolbar.

If the workflow is valid, a confirmation message appears. If the workflow is invalid, a list of errors appears.

8 For an invalid workflow, click an error message.

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

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

Important Always check that the fix that Orchestrator proposes is appropriate.

For example, the proposed action might be to delete an unused attribute, when in fact that attribute might not be bound correctly.

9 Repeat the preceding steps until you have eliminated all validation errors.

Results

You validated a workflow and fixed the validation errors.

What to do next

You can run the workflow.

Debugging Workflows

Orchestrator provides a workflow debugging tool. You can debug a workflow to inspect the input and output parameters and attributes at the start of any activity, replace parameter or attribute values during a workflow run in edit mode, and resume a workflow from the last failed activity.

You can debug workflows from the standard workflow library and custom workflows. You can debug custom workflows while developing them in the workflow editor.

Debug a Workflow

You can debug elements of a workflow by adding breakpoints to the elements in the workflow schema.

When a breakpoint is reached, you have several options to continue the debugging process. When you debug an element from the workflow schema, you can view general information about the workflow run, modify the workflow variables, and view log messages.

Prerequisites

Log in to the Orchestrator client as a user who can run workflows.

Procedure



- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Workflows** view.
- 3 Select a workflow from the workflow library and click the **Schema** tab.
- 4 To add breakpoints to the schema elements that you want to debug, right-click a workflow element and select **Toggle breakpoint**.



You can enable or disable the toggled breakpoints.

- 5 Click the **Debug workflow** icon ()

If the workflow requires input parameters, you must provide them.

- 6 When the workflow run is paused after it reaches a breakpoint, select one of the available options.

Option	Description
 Resume	Resumes the workflow run until another breakpoint is reached.
 Step into	Lets you step into a workflow element.
Note You cannot step into a nested workflow element when you debug a workflow in the workflow editor.	

Option	Description
 Step over	Steps over the current element in the schema and pauses the workflow run on the next element.
 Step return	Exits the workflow element that you have stepped into.

- 7 (Optional) From the **Breakpoints** tab, modify the breakpoints.

You can enable, disable, or remove existing breakpoints.

- 8 (Optional) From the **Variables** tab, review the variables.

You can modify the values of some of the variables during the debugging process.

Example Workflow Debugging



You can debug a workflow from the standard workflow library.

For example, if you provide an incorrect recipient address, you can correct the value when you debug the Example interaction with email workflow.

Prerequisites

Log in to the Orchestrator client as a user who can run Mail workflows.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Workflows** view.
- 3 In the workflows hierarchical list, open **Library > Mail**.
- 4 Select the Example interaction with email workflow and click the **Schema** tab.
- 5 Right-click the **Email Send (Interaction)** workflow element and select **Toggle breakpoint**.
- 6 Click the **Debug workflow** icon (.
- 7 Provide the required information.
 - a In the **Destination address** text box, type an incomplete recipient address.
For example, *name@company.c*.
 - b Select an LDAP group of users who are authorized to answer the query.
 - c Click **Submit**.
- 8 When the breakpoint is reached, click the **Step into** icon (.
- 9 On the **Variables** tab, verify the values.
- 10 In the **toAddress** text box, type the correct recipient address value.
For example, *name@company.com*.

11 Click the **Resume** icon () to continue the workflow run.

Results

The workflow uses the value that you provided during the debugging process and continues the workflow run.

Running Workflows

An Orchestrator workflow runs according to a logical flow of events.

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

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

This sequence repeats for each schema element until the end of the workflow.

Workflow Token Check Points

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

End of Workflow

The workflow ends if the current active schema element is an end element. After the workflow reaches an end element, other workflows or applications can use the workflow's output parameters.

Run a Workflow in the Workflow Editor

You can run a workflow while you are developing it.

Running a workflow in the workflow editor lets you verify that the workflow runs correctly without interrupting the development process. You can view log messages that provide information about the workflow run. If the workflow run returns unexpected results, you can modify the workflow and run it again without closing the workflow editor.

Prerequisites

- Create a workflow.
- Open the workflow for editing in the workflow editor.
- Validate the workflow.

Procedure

- 1 Click the **Schema** tab.
- 2 Click **Run**.
- 3 (Optional) Review the messages in the **Logs** tab.

Run a Workflow

You can perform automated operations in vCenter Server by running workflows from the standard library or workflows that you create.

For example, you can create a virtual machine by running the Create simple virtual machine workflow.

Prerequisites

Verify that you have configured the vCenter Server plug-in. For details, see *Installing and Configuring vRealize Orchestrator*.

Procedure

- 1 From the drop-down menu in the Orchestrator Legacy Client, select **Run**.
- 2 Click the **Workflows** view.
- 3 In the workflows hierarchical list, open **Library > vCenter > Virtual machine management > Basic** to navigate to the Create simple virtual machine workflow.
- 4 Right-click the Create simple virtual machine workflow and select **Start workflow**.
- 5 Provide the general parameters and click **Next**.

Option	Action
Virtual machine name	Name the virtual machine orchestrator-test .
Virtual machine folder	<ol style="list-style-type: none"> a Click Not set for the Virtual machine folder value. b Select a virtual machine folder from the inventory. <p>The Select button is inactive until you select an object of the correct type, in this case, VC:VmFolder.</p>
Size of the new disk in GB	Enter an appropriate numeric value.
Memory size in MB	Enter an appropriate numeric value.
Number of virtual CPUs	Select an appropriate number of CPUs from the Number of virtual CPUs drop-down menu.

Option	Action
Virtual machine guest OS	Click the Not set link and select a guest operating system from the list.
Make the disk thin provisioned	Select whether to make the disk thin or thick provisioned.

6 Provide the infrastructure parameters.

Option	Description
Host on which to create the virtual machine	Click Not set for the Host on which to create the virtual machine value and navigate through the vCenter Server infrastructure hierarchy to a host machine.
Resource pool	Click Not set for the Resource pool value and navigate through the vCenter Server infrastructure hierarchy to a resource pool.
The network to connect to	Click Not set for the The network to connect to value and select a network. To see all available networks, press Enter in the Filter text box.
Datastore in which to store the virtual machine files	Click Not set for the Datastore in which to store the virtual machine files value and navigate through the vCenter Server infrastructure hierarchy to a datastore.

7 To run the workflow, click **Submit**.

A workflow token appears under the Create simple virtual machine workflow, showing the workflow running icon.

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

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

10 Click the **Inventory** view.

11 Navigate through the vCenter Server infrastructure hierarchy to the resource pool you defined.

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

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

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

Results

The Create simple virtual machine workflow ran successfully.

What to do next

You can log in vSphere Client and manage the new virtual machine.

Resuming a Failed Workflow Run

If a workflow fails, Orchestrator provides an option to resume the workflow run from the last failed activity.

You can change the parameters of the workflow and attempt to resume it, or retain the parameters and make changes to external components that affect the workflow run. For example, if a workflow run fails due to a problem in a third-party system, you can make changes to the system and resume the workflow run from the failed activity, without changing the workflow parameters and without repeating the successful activities.

Set the Behavior for Resuming a Failed Workflow Run

You can set the behavior for resuming a failed run for each custom workflow. The default workflows in the library use the default system setting for resuming a failed workflow run.

You can change the default system behavior by modifying a configuration file. See [Set Custom Properties for Resuming Failed Workflow Runs](#).

Prerequisites

Verify that you have permissions to edit the workflow.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Workflows** view.
- 3 Expand the workflows hierarchical list to navigate to the workflow for which you want to set the behavior.
- 4 Right-click the workflow and select **Edit**.

The workflow editor opens.

- 5 On the **General** tab, select an option from the **Resume from failed behavior** drop-down menu.

Option	Description
System default	Follows the default behavior.
Enabled	If a workflow run fails, a pop-up window displays an option to resume the workflow run.
Disabled	If a workflow run fails, it cannot be resumed.

- 6 Click **Save and close**.

Set Custom Properties for Resuming Failed Workflow Runs

By default, Orchestrator is not set up to resume failed workflow runs. You can enable Orchestrator to resume failed workflow runs and set a custom timeout period after which failed workflow runs cannot be resumed.

Procedure

- 1 On the Orchestrator server system, navigate to `/etc/vco/app-server/`.
- 2 Open the `vmo.properties` configuration file in a text editor.
- 3 Set Orchestrator to resume failed workflow runs by editing the following line in the `vmo.properties` file.

```
com.vmware.vco.engine.execute.resume-from-failed=true
```

- 4 Set a custom timeout period for resuming failed workflow runs by editing the following line in the `vmo.properties` file.

```
com.vmware.vco.engine.execute.resume-from-failed.timeout-sec=<seconds>
```

The value you set overrides the default timeout setting of 86400 seconds.

- 5 Save the `vmo.properties` file.
- 6 Restart the Orchestrator server.

Resume a Failed Workflow Run

You can resume a workflow run from the last failed activity, if resuming a failed run is enabled for the workflow.

When the option for resuming a failed workflow run is enabled, you can change the parameters of the workflow and try to resume it by using the options in the pop-up window that appears after the workflow fails. You can also retain the parameters and make changes to external components that affect the workflow run. If you do not select an option, the workflow run times out and cannot be resumed. For modifying the timeout period, see [Set Custom Properties for Resuming Failed Workflow Runs](#).

Procedure

- 1 From the drop-down menu in the pop-up window, select **Resume** and click **Next**.
If you select **Cancel**, the workflow run cannot be resumed later.
- 2 (Optional) Modify the workflow parameters.
- 3 Click **Submit**.

Generate Workflow Documentation

You can export documentation in PDF format about a workflow or a workflow folder that you select at any time.

The exported document contains detailed information about the selected workflow or the workflows in the folder. The information about each workflow includes name, version history of the workflow, attributes, parameter presentation, workflow schema, and workflow actions. In addition, the documentation also provides the source code for the used actions.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Run** or **Design**.
- 2 Click the **Workflows** view.
- 3 Navigate to the workflow or workflow folder for which you want to generate documentation and right-click it.
- 4 Select **Generate documentation**.
- 5 Browse to locate the folder in which to save the PDF file, provide a file name, and click **Save**.

Results

The PDF file containing the information about the selected workflow, or the workflows in the folder, is saved on your system.

Use Workflow Version History

You can use version history to revert a workflow to a previously saved state. You can revert the workflow state to an earlier or a later workflow version. You can also compare the differences between the current state of the workflow and a saved version of the workflow.

Orchestrator creates a version history item for each workflow when you increase and save the workflow version. Subsequent changes to the workflow do not change the current saved version. For example, when you create a workflow version 1.0.0 and save it, the state of the workflow is stored in the version history. If you make any changes to the workflow, you can save the workflow state in the Orchestrator client, but you cannot apply the changes to workflow version 1.0.0. To store the changes in the version history, you must create a subsequent workflow version and save it. The version history is kept in the database with the workflow itself.

Prerequisites

Open a workflow for editing in the workflow editor.

Procedure

- 1 Click the **General** tab in the workflow editor and click **Show version history**.

- 2 Select a workflow version and click **Diff Against Current** to compare the differences.

A window displays the differences between the current workflow version and the selected workflow version.

- 3 Select a workflow version and click **Revert** to restore the state of the workflow.

Caution If you have not saved the current workflow version, it is deleted from the version history and you cannot revert to the current version.

The workflow state is reverted to the state of the selected version.

Develop a Simple Example Workflow

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

The example workflow that you are about to create starts an existing virtual machine in vCenter Server and sends an email to the administrator to confirm that the virtual machine has started.

The example workflow performs the following tasks:

- 1 Prompts the user to select a virtual machine to start.
- 2 Prompts the user for an email address to which it can send notifications.
- 3 Checks whether the selected virtual machine is already powered on.
- 4 Sends a request to the vCenter Server instance to start the virtual machine.
- 5 Waits for vCenter Server to start the virtual machine, and returns an error if the virtual machine fails to start or if starting the virtual machine takes too long.
- 6 Waits for vCenter Server to start VMware Tools on the virtual machine, and returns an error if the virtual machine fails to start or if starting VMware Tools takes too long.
- 7 Verifies that the virtual machine is running.
- 8 Sends a notification to the provided email address, informing that the machine has started or that an error occurred.

The ZIP file of Orchestrator examples available for download from the landing page of the Orchestrator documentation contains a complete version of the Start VM and Send Email workflow.

The process for developing the example workflow consists of several tasks.

Prerequisites

Before you attempt to develop the simple example workflow, read [Key Concepts of Workflows](#).

Procedure

1 [Create the Simple Workflow Example](#)

You must begin the workflow development process by creating the workflow in the Orchestrator client.

2 [Create the Schema of the Simple Workflow Example](#)

You can create a workflow's schema in the workflow editor. The workflow schema contains the elements that the workflow runs and determines the logical flow of the workflow.

3 [\(Optional\) Create the Simple Workflow Example Zones](#)

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

4 [Define the Parameters of the Simple Workflow Example](#)

In this phase of workflow development, you define the input parameters that the workflow requires to run. For the example workflow, you need an input parameter for the virtual machine to power on, and a parameter for the email address of the person to inform about the result of the operation. When users run the workflow, they will be required to specify the virtual machine to power on and an email address.

5 [Define the Simple Workflow Example Decision Bindings](#)

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

6 [Bind the Action Elements of the Simple Workflow Example](#)

You can bind a workflow's elements together in the workflow editor. Bindings define how the action elements process input parameters and generate output parameters.

7 [Bind the Simple Workflow Example Scripted Task Elements](#)

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

8 [Define the Simple Workflow Example Exception Bindings](#)

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

9 [Set the Read-Write Properties for Attributes of the Simple Workflow Example](#)

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

10 [Set the Simple Workflow Example Parameter Properties](#)

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

11 [Set the Layout of the Simple Workflow Example Input Parameters Dialog Box](#)

You create the layout or presentation of the input parameters dialog box in the workflow editor. The input parameters dialog box opens when users run a workflow that needs input parameters to run.

12 [Validate and Run the Simple Workflow Example](#)

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

Create the Simple Workflow Example

You must begin the workflow development process by creating the workflow in the Orchestrator client.

Prerequisites

Verify that the following components are installed and configured on the system.

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

For information about how to install and configure vCenter Server, see the *vSphere Installation and Setup* documentation. For information about how to configure Orchestrator to use an SMTP server, see *Installing and Configuring VMware vRealize Orchestrator*.

Important This example workflow is created with the workflow editor of the Orchestrator Legacy Client. Avoid editing legacy workflows in the new HTML5 client, if you plan to continue using these workflows in your legacy environment.

Procedure

- 1** From the drop-down menu in the Orchestrator client, select **Design**.
- 2** Click the **Workflows** view.
- 3** Right-click the root of the workflows list and select **Add folder**.
- 4** Name the new folder **Workflow Examples** and click **OK**.

- 5 Right-click the **Workflow Examples** folder and select **New workflow**.
- 6 Name the new workflow **Start VM and Send Email** and click **OK**.
The workflow editor opens.
- 7 In the **General** tab, click the version number digits to increment the version number.
Because this is the initial creation of the workflow, set the version to **0.0.1**.
- 8 Click the **Server restart behavior** value in the **General** tab to set whether the workflow resumes after a server restart.
- 9 Type a description of what the workflow does in the **Description** text box in the **General** tab.
For example, you can add the following description.
This workflow starts a virtual machine and sends a confirmation email to the Orchestrator administrator.
- 10 Click **Save** at the bottom of the **General** tab.

Results

You created a workflow called Start VM and Send Email, but you did not define its functions.

What to do next

Create the workflow's schema.

Create the Schema of the Simple Workflow Example

You can create a workflow's schema in the workflow editor. The workflow schema contains the elements that the workflow runs and determines the logical flow of the workflow.

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example](#).
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Click the **Schema** tab in the workflow editor.
- 2 From the **Generic** menu, drag a decision element to the arrow that links the Start element and the End element in the schema.
- 3 Double-click the decision element and change its name to **VM powered on?**.

The decision element corresponds to a boolean function that checks whether the virtual machine is already powered on.

- 4 From the **Generic** menu, drag an action element to the red arrow that links the decision element and an End element.

The dialog box for action selection appears.

- 5 Type **start** in the **Filter** text box, select the **startVM** action from the filtered list of actions, and click **Select**.
- 6 Drag the following action elements, one after the other, to the blue arrow that links the startVM action element to an End element.

vim3WaitTaskEnd

Suspends the workflow run and pings an ongoing vCenter Server task at regular intervals, until that task is finished. The startVM action starts a virtual machine and the vim3WaitTaskEnd action makes the workflow wait while the virtual machine starts up. After the virtual machine starts, the vim3WaitTaskEnd lets the workflow resume.

vim3WaitToolsStarted

Suspends the workflow run and waits until VMware Tools starts on the target virtual machine.

- 7 From the **Generic** menu, drag a scriptable task element to the blue arrow that links the vim3WaitToolsStarted action element to an End element.
- 8 Double-click the scriptable task element and rename it to **OK**.
- 9 Drag another scriptable task element to the green arrow that links the VM powered on? decision element to an End element , and name this scriptable task element **Already started**.
- 10 Modify the linking of the Already started scriptable task element.
 - a Drag the Already started scriptable task element to the left of the startVM action element.
 - b Delete the blue arrow that connects the Already started scriptable task element to an End element.
 - c Link the Already started scriptable task element to the vim3WaitToolsStarted action element with a blue arrow.
- 11 From the **Generic** menu, drag the following scriptable task elements into the schema.
 - Drag a scriptable task element to the startVM action element and name the scriptable task element **Start VM Failed**.
 - Drag a scriptable task element to the vim3WaitTaskEnd action element and name the scriptable task element **Timeout 1**.
 - Drag a scriptable task element to the vim3WaitToolsStarted action element and name the scriptable task element **Timeout 2**.
 - Drag a scriptable task element to the blue arrow that links the OK scriptable task element to an End element, name the new scriptable task element **Send Email**, and drag it to the right of the OK scriptable task element.

- Link the Start VM Failed, Timeout 1, and Timeout 2 scriptable task elements to the Send Email scriptable task element with blue arrows.
- Drag a scriptable task element to the Send Email scriptable task element, name the new scriptable task element **Send Email Failed**, drag it to the right of the Timeout 2 scriptable task element, and link it to the End element with a blue arrow.

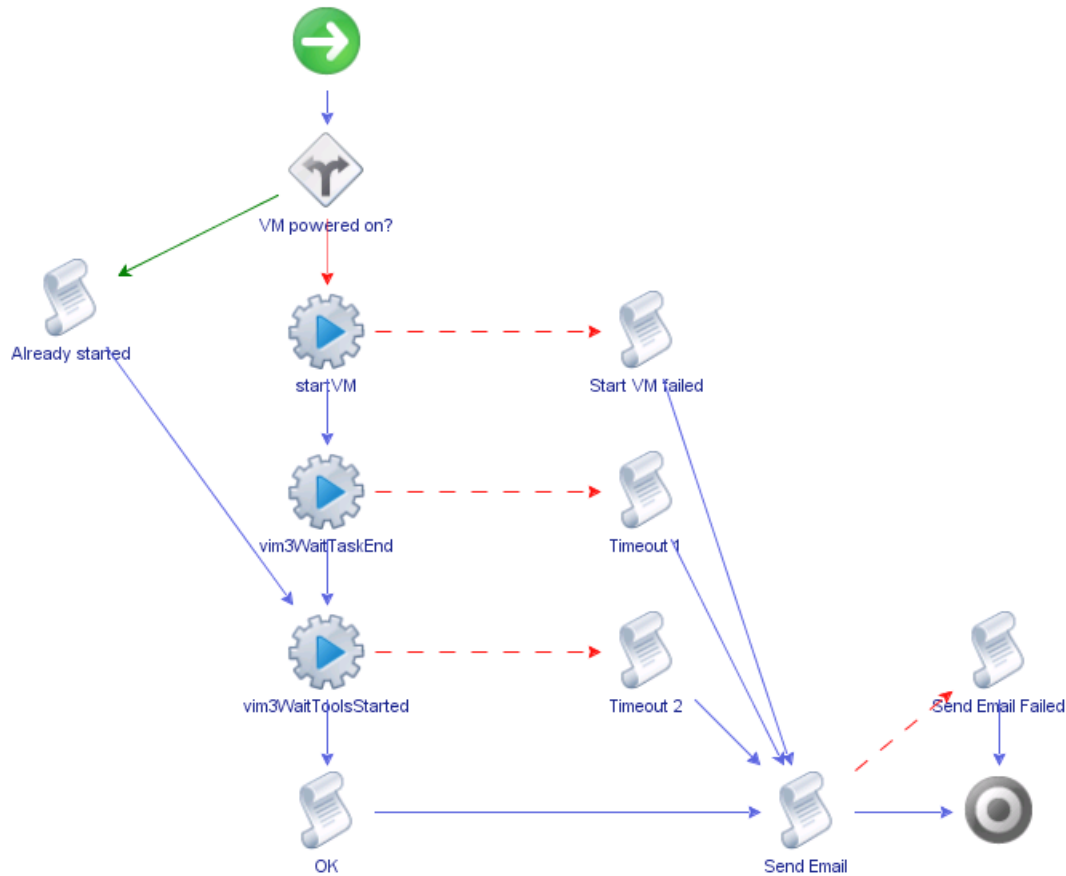
12 Drag the End element to the right of the Send Email scriptable task element.

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

Results

The following figure shows the layout of the Start VM and Send Email workflow schema elements.

Figure 1-10. Linking the Elements of the Start VM and Send Email Example Workflow



What to do next

You can highlight different zones in the workflow.

Create the Simple Workflow Example Zones

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

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example.](#)
- [Create the Schema of the Simple Workflow Example.](#)
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Drag a workflow note element from the **Generic** menu into the workflow editor.
- 2 Position the workflow note over the `Already started` scriptable task element.
- 3 Drag the edges of the workflow note to resize it so that it surrounds the `Already started` scriptable task element.
- 4 Double-click the text and add a description.

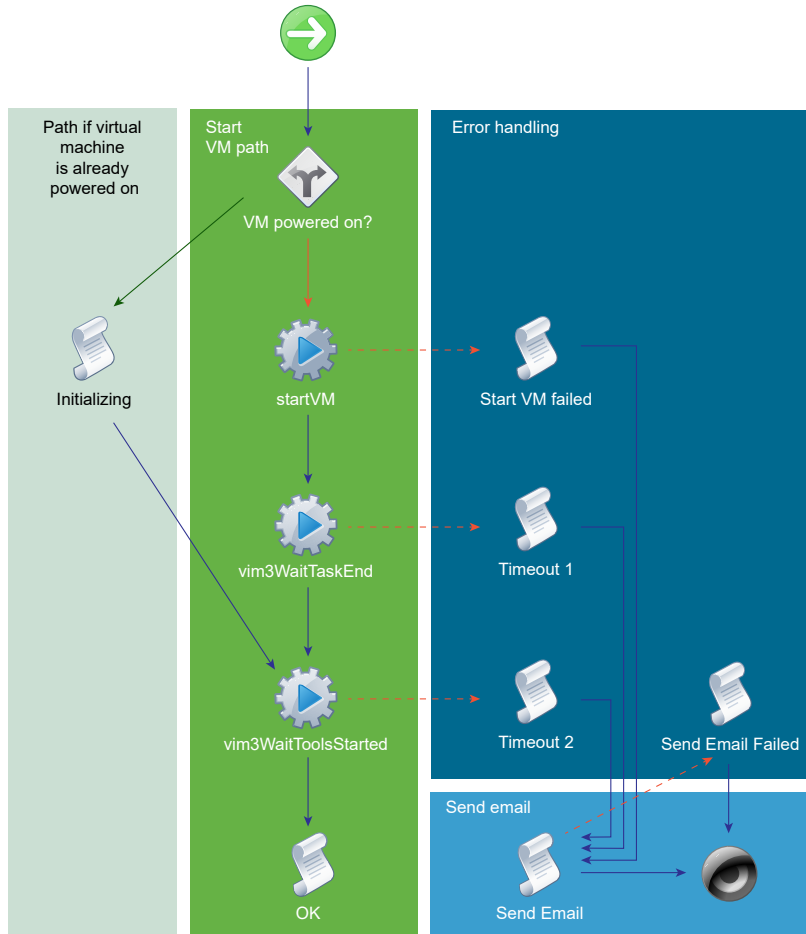
For example, **Path if virtual machine is already powered on.**

- 5 Press Ctrl+E to select the background color.
- 6 Repeat the preceding steps to highlight other zones in the workflow.
 - Place a note over the vertical sequence of elements from the `VM powered on?` decision element to the `OK` element. Add the description **Start VM path.**
 - Place a note over the `startVM failed`, both `Timeout` scriptable task elements and the `Send Email Failed` scriptable task element. Add the description **Error handling.**
 - Place a note over the `Send Email` scriptable task element. Add the description **Send email.**

Results

The following figure shows what the example workflow zones should look like.

Figure 1-11. Start VM and Send Email Example Workflow Zones



What to do next

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

Define the Parameters of the Simple Workflow Example

In this phase of workflow development, you define the input parameters that the workflow requires to run. For the example workflow, you need an input parameter for the virtual machine to power on, and a parameter for the email address of the person to inform about the result of the operation. When users run the workflow, they will be required to specify the virtual machine to power on and an email address.

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example.](#)
- Open the workflow for editing in the workflow editor.

Procedure

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

Results

You defined the workflow's input parameters.

What to do next

Define the bindings between the element parameters.

Define the Simple Workflow Example Decision Bindings

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

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example](#).
- [Create the Schema of the Simple Workflow Example](#).
- [Define the Parameters of the Simple Workflow Example](#).
- Open the workflow for editing in the workflow editor.

Procedure

- 1 On the **Schema** tab, click the **Edit** icon (✎) of the **VM Powered On?** decision element.
- 2 On the **Decision** tab, click the **Not set (NULL)** button and select **vm** as the decision element's input parameter from the list of proposed parameters.
- 3 Select the **Power State equals** statement from the list of decision statements proposed in the drop-down menu.
A **Not set** button appears in the value text box, which presents you with a limited choice of possible values.
- 4 Select **poweredOn**.
- 5 Click **Save** at the bottom of the workflow editor's **Schema** tab.

Results

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

What to do next

You must define the bindings for the other elements in the workflow.

Bind the Action Elements of the Simple Workflow Example

You can bind a workflow's elements together in the workflow editor. Bindings define how the action elements process input parameters and generate output parameters.

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example.](#)
- [Create the Schema of the Simple Workflow Example.](#)
- [Define the Parameters of the Simple Workflow Example.](#)
- [Define the Simple Workflow Example Decision Bindings.](#)
- Open the workflow for editing in the workflow editor.

Procedure

- 1 On the **Schema** tab, click the **Edit** icon (✎) of the **startVM** action element.

- 2 Set the following general information on the **Info** tab.

Option	Action
Interaction	Select No External interaction .
Business Status	Select the check box and add the text Sending start VM .
Description	Leave the text Start / Resume a VM. Return the start task.

- 3 Click the **IN** tab.

The **IN** tab displays the two possible input parameters available to the startVM action, vm and host.

Orchestrator automatically binds the vm parameter to vm[in-parameter] because the startVM action can only take a VC:VirtualMachine as an input parameter. Orchestrator detects the vm parameter you defined when you set the workflow input parameters and so binds it to the action automatically.

- 4 Set host to **NULL**.

This is an optional parameter, so you can set it to null. However, if you leave it set to **Not set**, the workflow cannot validate.

- 5 Click the **OUT** tab.

The default output parameter that all actions generate, actionResult, appears.

- 6 For the actionResult parameter, click **Not set**.

- 7 Click **Create parameter/attribute in workflow**.

The Parameter information dialog box displays the values that you can set for this output parameter. The output parameter type for the startVM action is a VC:Task object.

- 8 Name the parameter **powerOnTask** and provide a description.

For example, **Contains the result of powering on a VM**.

- 9 Click **Create workflow ATTRIBUTE with the same name** and click **OK** to exit the Parameter information dialog box.

- 10 Repeat the preceding steps to bind the input and output parameters to the vim3WaitTaskEnd and vim3WaitToolsStarted action elements.

[Simple Workflow Example Action Element Bindings](#) lists the bindings for the vim3WaitTaskEnd and vim3WaitToolsStarted action elements.

- 11 Click **Save** at the bottom of the workflow editor's **Schema** tab.

Results

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

What to do next

Bind the scriptable task elements and define their functions.

Simple Workflow Example Action Element Bindings

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

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

vim3WaitTaskEnd Action

The vim3WaitTaskEnd action element declares constants to track the progress of a task and a polling rate. The following table shows the input and output parameter bindings that the vim3WaitTaskEnd action requires.

Table 1-53. Binding Values of the vim3WaitTaskEnd Action

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
task	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: powerOnTask ■ Source parameter: task[attribute] ■ Type: VC:Task ■ Description: Contains the result of powering on a VM.
progress	IN	Create	<ul style="list-style-type: none"> ■ Local Parameter: progress ■ Source parameter: progress[attribute] ■ Type: Boolean ■ Value: No (false) ■ Description: Log progress while waiting for the vCenter Server task to complete.

Table 1-53. Binding Values of the vim3WaitTaskEnd Action (continued)

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
pollRate	IN	Create	<ul style="list-style-type: none"> ■ Local Parameter: pollRate ■ Source parameter: pollRate[attribute] ■ Type: number ■ Value: 2 ■ Description: Polling rate in seconds at which vim3WaitTaskEnd checks the advancement of the vCenter Server task.
actionResult	OUT	Create	<ul style="list-style-type: none"> ■ Local Parameter: actionResult[attribute] ■ Source parameter: returnedManagedObject[attribute] ■ Type: Any ■ Description: The returned managed object from the waitTaskEnd action.

vim3WaitToolsStarted Action

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

The vim3WaitToolsStarted action element has no output, so requires no output binding.

Table 1-54. Binding Values of the vim3WaitToolsStarted Action

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Automatic binding	<ul style="list-style-type: none"> Local Parameter: vm Source parameter: vm[in-parameter] Type: VC:VirtualMachine Value: Not editable, variable is not a workflow attribute. Description: The virtual machine to start.
pollingRate	IN	Bind	<ul style="list-style-type: none"> Local Parameter: pollRate Source parameter: pollRate[attribute] Type: number Description: The polling rate in seconds at which vim3WaitTaskEnd checks the advancement of the vCenter server task.
timeout	IN	Create	<ul style="list-style-type: none"> Local Parameter: timeout Source parameter: timeout[attribute] Type: number Value: 10 Description: The timeout limit that vim3WaitToolsStarted waits before throwing an exception.

Bind the Simple Workflow Example Scripted Task Elements


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

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example.](#)
- [Create the Schema of the Simple Workflow Example.](#)
- [Define the Parameters of the Simple Workflow Example.](#)
- [Define the Simple Workflow Example Decision Bindings.](#)
- Open the workflow for editing in the workflow editor.


Procedure

- 1 On the **Schema** tab, click the **Edit** icon () of the Already Started scriptable task element.
- 2 Set the following general information in the **Info** tab.

Option	Action
Interaction	Select No External interaction .
Business Status	Select the check box and add the text VM already powered on .
Description	Leave the text The VM is already powered on, bypassing startVM and waitTaskEnd, checking if the VM tools are up and running..

- 3 Click the **IN** tab.

Because this is a custom scriptable task element, no properties are predefined for you.

- 4 Click the **Bind to workflow parameter/attribute** icon ()
- 5 Select `vm` from the proposed list of parameters.
- 6 Leave the **OUT** and **Exception** tabs blank.

This element does not generate an output parameter or exception.

- 7 Click the **Scripting** tab.
- 8 Add the following JavaScript function.

```
//Writes the following event in the Orchestrator database
Server.log("VM '"+ vm.name +"' already started");
```

- 9 Repeat the preceding steps to bind the remaining input parameters to the other scriptable task elements.

[Simple Workflow Example Scriptable Task Element Bindings](#) lists the bindings for the Start VM failed, both Timeout or Error, Send Email Failed, and the OK scriptable task elements.

- 10 Click **Save** at the bottom of the workflow editor's **Schema** tab.

Results

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

What to do next

You must define the exception handling.

Simple Workflow Example Scriptable Task Element Bindings

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

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

Start VM Failed Scriptable Task

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

The following table shows the input and output parameter bindings that the Start VM Failed scriptable task element requires.

Table 1-55. Bindings of the Start VM Failed Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: vm ■ Source parameter: vm[in-parameter] ■ Type: VC:VirtualMachine ■ Description: The virtual machine to power on.
errorCode	IN	Create	<ul style="list-style-type: none"> ■ Local Parameter: errorCode ■ Source parameter: errorCode[attribute] ■ Type: string ■ Description: Catch any exceptions while powering on a VM.
body	OUT	Create	<ul style="list-style-type: none"> ■ Local Parameter: body ■ Source parameter: body[attribute] ■ Type: string ■ Description: The email body

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

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

Timeout 1 Scriptable Task Element

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

The following table shows the input and output parameter bindings that the Timeout 1 scriptable task element requires.

Table 1-56. Bindings of the Timeout 1 Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: vm ■ Source parameter: vm[in-parameter] ■ Type: VC:VirtualMachine ■ Description: The virtual machine to start.
errorCode	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: errorCode ■ Source parameter: errorCode[attribute] ■ Type: string ■ Description: Catch any exceptions while powering on a VM.
body	OUT	Bind	<ul style="list-style-type: none"> ■ Local Parameter: body ■ Source parameter: body[attribute] ■ Type: string ■ Description: The email body

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

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

Timeout 2 Scriptable Task Element

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

The following table shows the input and output parameter bindings that the Timeout 2 scriptable task element requires.

Table 1-57. Bindings of the Timeout 2 Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: vm ■ Source parameter: vm[in-parameter] ■ Type: VC:VirtualMachine ■ Description: The virtual machine to power on.
errorCode	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: errorCode ■ Source parameter: errorCode[attribute] ■ Type: string ■ Description: Catch any exceptions while powering on a VM.
body	OUT	Bind	<ul style="list-style-type: none"> ■ Local Parameter: body ■ Source parameter: body[attribute] ■ Type: string ■ Description: The email body

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

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

OK Scriptable Task Element

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

The following table shows the input and output parameter bindings that the OK scriptable task element requires.

Table 1-58. Bindings of the OK Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: vm ■ Source parameter: vm[in-parameter] ■ Type: VC:VirtualMachine ■ Description: The virtual machine to power on.
body	OUT	Bind	<ul style="list-style-type: none"> ■ Local Parameter: body ■ Source parameter: body[attribute] ■ Type: string ■ Description: The email body

The OK scriptable task element requires the following scripted function.

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

Send Email Failed Scriptable Task Element

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

The following table shows the input parameter bindings that the Send Email Failed scriptable task element requires.

Table 1-59. Bindings of the Send Email Failed Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: vm ■ Source parameter: vm[in-parameter] ■ Type: VC:VirtualMachine ■ Description: The virtual machine to power on.
toAddress	IN	Bind	<ul style="list-style-type: none"> ■ Local Parameter: toAddress ■ Source parameter: toAddress[in-parameter] ■ Type: string ■ Description: The email address of the person to inform of the result of this workflow
emailErrorCode	IN	Create	<ul style="list-style-type: none"> ■ Local Parameter: emailErrorCode ■ Source parameter: emailErrorCode[attribute] ■ Type: string ■ Description: Catch any exceptions while sending an email

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

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

Send Email Scriptable Task Element

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

The following table shows the input and output parameter bindings that the Send Email scriptable task element requires.

Table 1-60. Bindings of the Send Email Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> Local Parameter: vm Source parameter: vm[in-parameter] Type: VC:VirtualMachine Description: The virtual machine to power on.
toAddress	IN	Bind	<ul style="list-style-type: none"> Local Parameter: toAddress Source parameter: toAddress[in-parameter] Type: string Description: The email address of the person to inform of the result of this workflow
body	IN	Bind	<ul style="list-style-type: none"> Local Parameter: body Source parameter: body[attribute] Type: string Description: The email body
smtpHost	IN	Create	<ul style="list-style-type: none"> Local Parameter: smtpHost Source parameter: smtpHost[attribute] Type: string Description: The email SMTP server
fromAddress	IN	Create	<ul style="list-style-type: none"> Local Parameter: fromAddress Source parameter: fromAddress[attribute] Type: string Description: The email address of the sender
subject	IN	Create	<ul style="list-style-type: none"> Local Parameter: subject Source parameter: subject[attribute] Type: string Description: The email subject

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

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

//Apply methods on this instance that populate the email message
```



```

myEmailMessage.smtpHost = smtpHost;
myEmailMessage.fromAddress = fromAddress;
myEmailMessage.toAddress = toAddress;
myEmailMessage.subject = subject;
myEmailMessage.addMimePart(body , "text/html");

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

```

Define the Simple Workflow Example Exception Bindings

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

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

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example.](#)
- [Create the Schema of the Simple Workflow Example.](#)
- [Define the Parameters of the Simple Workflow Example.](#)
- [Define the Simple Workflow Example Decision Bindings.](#)
- [Bind the Action Elements of the Simple Workflow Example.](#)
- [Bind the Simple Workflow Example Scripted Task Elements.](#)
- Open the workflow for editing in the workflow editor.

Procedure

- 1 On the **Schema** tab, click the **Edit** icon (✎) of the **startVM** action element.
- 2 Click the **Exception** tab.
- 3 Click the **Not set** button.
- 4 Select **errorCode** from the proposed list.
- 5 Repeat the preceding steps to set the exception binding to **errorCode** for both `vim3WaitTaskEnd` and `vim3WaitToolsStarted`.
- 6 Click the **Edit** icon (✎) of the **Send Email** scriptable task element.
- 7 Click the **Exception** tab.
- 8 Click the **Not set** button.
- 9 Select **emailErrorCode** from the proposed list.

10 Click **Save** at the bottom of the workflow editor's **Schema** tab.

Results

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

What to do next

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

Set the Read-Write Properties for Attributes of the Simple Workflow Example

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

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

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example.](#)
- [Create the Schema of the Simple Workflow Example.](#)
- [Define the Parameters of the Simple Workflow Example.](#)
- [Define the Simple Workflow Example Decision Bindings.](#)
- [Bind the Action Elements of the Simple Workflow Example.](#)
- [Bind the Simple Workflow Example Scripted Task Elements.](#)
- [Define the Simple Workflow Example Exception Bindings.](#)
- Open the workflow for editing in the workflow editor.

Procedure

1 Click the **General** tab at the top of the workflow editor.

Under **Attributes** is a list of all the defined attributes, with check boxes next to each attribute. When you select these check boxes, you set attributes as read-only.

2 Select the check boxes to make the following attributes read-only constants:

- progress
- pollRate
- timeout
- smtpHost
- fromAddress

- subject

Results

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

What to do next

Set the parameter properties and place constraints on the possible values for that parameter.

Set the Simple Workflow Example Parameter Properties


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

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example.](#)
- [Create the Schema of the Simple Workflow Example.](#)
- [Define the Parameters of the Simple Workflow Example.](#)
- [Define the Simple Workflow Example Decision Bindings.](#)
- [Bind the Action Elements of the Simple Workflow Example.](#)
- [Bind the Simple Workflow Example Scripted Task Elements.](#)
- [Define the Simple Workflow Example Exception Bindings.](#)
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Click the **Presentation** tab in the workflow editor.
The two input parameters you defined for this workflow are listed.
- 2 Click the **(VC:VirtualMachine)vm** parameter.
- 3 Add a description in the **General** tab in the bottom half of the screen.
For example, type **The virtual machine to start.**
- 4 Click the **Properties** tab in the bottom half of the screen.
On this tab, you can set the properties for the (VC:VirtualMachine)vm parameter.
- 5 Click the **Add property** icon (+).

- 6 From the list of proposed properties, select the **Mandatory input** property, click **Ok**, and set its value to **Yes**.



When you enable this property, users cannot run the Start VM and Send Email workflow without providing a virtual machine to start.

- 7 Click the **Add property** icon (+).
- 8 From the list of proposed properties, select **Select value as**, click **Ok**, and select **list** from the list of possible values.

When you set this property, you set how the user selects the value of the (VC:VirtualMachine)vm input parameter.

- 9 Click the **(string)toAddress** parameter in the top half of the **Presentation** tab.
- 10 Add a description in the **Description** tab in the bottom half of the screen.

For example, type **The email address of the person to notify**.

- 11 Click the **Properties** tab for (string)toAddress and click the **Add property** icon (+).
- 12 From the list of proposed properties, select the **Mandatory input** property, click **Ok**, and set its value to **Yes**.
- 13 Click the **Add property** icon (+).

- 14 From the list of proposed properties, select **Matching regular expression** and click **Ok**.
This property allows you to set constraints on what users can provide as input .

- 15 Click the **Value** text box for **Matching regular expression** and set the constraints to `[a-zA-Z0-9_%-+.] + @ [a-zA-Z0-9-.\+ \. [a-zA-Z] {2,4}`.

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

Results

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

What to do next

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

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

You create the layout or presentation of the input parameters dialog box in the workflow editor. The input parameters dialog box opens when users run a workflow that needs input parameters to run.

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example.](#)
- [Create the Schema of the Simple Workflow Example.](#)
- [Define the Parameters of the Simple Workflow Example.](#)
- [Define the Simple Workflow Example Decision Bindings.](#)
- [Bind the Action Elements of the Simple Workflow Example.](#)
- [Bind the Simple Workflow Example Scripted Task Elements.](#)
- [Define the Simple Workflow Example Exception Bindings.](#)
- [Set the Read-Write Properties for Attributes of the Simple Workflow Example.](#)
- [Set the Simple Workflow Example Parameter Properties.](#)
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Click the **Presentation** tab in the workflow editor.
- 2 Right-click the **Presentation** node in the presentation hierarchical list and select **Create display group**.

A **New step** node and a **New group** sub-node appear under the **Presentation** node.

- 3 Right-click **New step** and select **Delete**.

Because this workflow has only two parameters, you do not need multiple layers of display sections in the input parameters dialog box.

- 4 Double-click **New group** to edit the group name and press Enter.

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

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

- 5 In the **Description** text box of the **General** tab at the bottom of the **Presentation** tab, provide a description for the new display group.

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

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

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

In the input parameters dialog box, a text box in which the user types the virtual machine name will appear under a Virtual Machine heading.

- 7 Repeat the preceding steps to create a display group for the `toAddress` parameter, setting the following properties:
 - a Create a display group and name it **Recipient's Email Address**.
 - b Add a description for the display group, for example, **Enter the email address of the person to notify when this virtual machine is powered-on**.
 - c Drag the `toAddress` parameter under the **Recipient's Email Address** display group.

Results

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

What to do next

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

Validate and Run the Simple Workflow Example

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

Prerequisites

Complete the following tasks.

- [Create the Simple Workflow Example](#).
- [Create the Schema of the Simple Workflow Example](#).
- [Define the Parameters of the Simple Workflow Example](#).
- [Define the Simple Workflow Example Decision Bindings](#).
- [Bind the Action Elements of the Simple Workflow Example](#).
- [Bind the Simple Workflow Example Scripted Task Elements](#).
- [Define the Simple Workflow Example Exception Bindings](#).
- [Set the Read-Write Properties for Attributes of the Simple Workflow Example](#).
- [Set the Simple Workflow Example Parameter Properties](#).
- [Set the Layout of the Simple Workflow Example Input Parameters Dialog Box](#).
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Click **Validate** in the **Schema** tab of the workflow editor.

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

- 2 After you have eliminated any errors, click **Save and Close** at the bottom of the workflow editor.

You return to the Orchestrator client.

- 3 Click the **Workflows** view.
- 4 Select **Workflow Examples > Start VM and Send Email** in the workflow hierarchical list.
- 5 Right-click the **Start VM and Send Email** workflow and select **Start workflow**.

The input parameters dialog box opens and prompts you for a virtual machine to start and an email address to send notifications to.

- 6 Select a virtual machine to start from the vCenter Server inventory.
- 7 Type an email address to which to send email notifications.
- 8 Click **Submit** to start the workflow.

A workflow token appears under the Start VM and Send Email workflow.

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

Results

If the workflow runs successfully, the virtual machine you selected is in the powered-on state, and the email recipient you defined receives a confirmation email.

What to do next

You can generate a document in which to review information about the workflow. See [Generate Workflow Documentation](#).

Develop a Complex Workflow

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

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

- 1 Prompts the user for a resource pool that contains the virtual machines of which to take snapshots.
- 2 Determines whether the resource pool contains running virtual machines.
- 3 Determines how many running virtual machines the resource contains.
- 4 Verifies whether an individual virtual machine running in the pool meets specific criteria for a snapshot to be taken.
- 5 Takes the snapshot of the virtual machine.

- 6 Determines whether more virtual machines exist in the pool of which to take snapshots.
- 7 Repeats the verification and snapshot process until the workflow has taken snapshots of all eligible virtual machines in the resource pool.

The ZIP file of Orchestrator examples that you can download from the landing page of the Orchestrator documentation contains a completed version of the Take a Snapshot of All Virtual Machines in a Resource Pool workflow.

Prerequisites

Before you attempt to develop this complex workflow, follow the exercises in [Develop a Simple Example Workflow](#). The procedures to develop a complex workflow provide the broad steps of the development process, but are not as detailed as the simple workflow exercises.

Procedure

1 [Create the Complex Workflow Example](#)

You must begin the workflow development process by creating the workflow in the Orchestrator client.

2 [Create a Custom Action for the Complex Workflow Example](#)

The Check VM scriptable element calls on an action that does not exist in the Orchestrator API. You must create the getVMDiskModes action.

3 [Create the Schema of the Complex Workflow Example](#)

You can create a workflow's schema in the workflow editor. The workflow schema contains the elements that the workflow runs, and determines the logical flow of the workflow.

4 [\(Optional\) Create the Complex Workflow Example Zones](#)

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

5 [Define the Parameters of the Complex Workflow Example](#)

You define workflow parameters in the workflow editor. The input parameters provide data for the workflow to process. The output parameters are the data the workflow returns when it completes its run.

6 [Define the Bindings for the Complex Workflow Example](#)

You can bind a workflow's elements together in the workflow editor. Bindings define the data flow of the workflow. You also bind the scriptable task elements to their JavaScript functions.

7 [Set the Complex Workflow Example Attribute Properties](#)

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

8 Create the Layout of the Complex Workflow Example Input Parameters

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

9 Validate and Run the Complex Workflow Example

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

Create the Complex Workflow Example

You must begin the workflow development process by creating the workflow in the Orchestrator client.

For information about how to install and configure vCenter Server, see the *vSphere Installation and Setup* documentation. For information about how to configure Orchestrator, see *Installing and Configuring VMware vRealize Orchestrator*.

Prerequisites

Verify that the following components are installed and configured on the system.

- vCenter Server, controlling a resource pool that contains some virtual machines
- The **Workflow Examples** folder in the workflows hierarchical list, that you created in [Create the Simple Workflow Example](#).

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Select **Workflows > Workflow Examples**.
- 3 Right-click the **Workflow Examples** folder and select **New workflow**.
- 4 Name the new workflow **Take a Snapshot of All Virtual Machines in a Resource Pool** and click **OK**.

The workflow editor opens.

- 5 On the **General** tab of the workflow editor, click the version number digits to increment the version number.

For the initial creation of the workflow, set the version to **0.0.1**.

- 6 Click the **Server restart behavior** value to set whether the workflow resumes after a server restart.
- 7 In the **Description** text box, type a description of what the workflow does.
- 8 Click **Save** at the bottom of the **General** tab.

Results

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

What to do next

You must create a custom action.

Create a Custom Action for the Complex Workflow Example

The Check VM scriptable element calls on an action that does not exist in the Orchestrator API. You must create the getVMDiskModes action.

For more detail about creating actions, see [Chapter 3 Developing Actions](#).

Prerequisites

Create the Take a Snapshot of All Virtual Machines in a Resource Pool workflow. See [Create the Complex Workflow Example](#).

Procedure

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

```
This action returns an array containing the disk modes of all disks on a VM.
The elements in the array each have one of the following string values:
- persistent
- independent-persistent
- nonpersistent
- independent-nonpersistent
Legacy values:
- undoable
- append
```

- 9 Click the **Scripting** tab.
- 10 Right-click in the top pane of the **Scripting** tab and select **Add Parameter** to create the following input parameter.
 - Name: vm
 - Type: VC:VirtualMachine

- Description: **The virtual machine for which to return the Disk Modes**

11 Add the following scripting in the bottom of the **Scripting** tab.

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

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

12 Click **Save and Close** to exit the **Actions** palette.

Results

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

What to do next

Create the workflow's schema.

Create the Schema of the Complex Workflow Example

You can create a workflow's schema in the workflow editor. The workflow schema contains the elements that the workflow runs, and determines the logical flow of the workflow.

Prerequisites

Complete the following tasks.

- [Create the Complex Workflow Example.](#)
- [Create a Custom Action for the Complex Workflow Example.](#)
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Click the **Schema** tab in the workflow editor.
- 2 Add the following schema elements to the workflow schema.

Element Type	Element Name	Position in Schema
Scriptable task	Initializing	Below the Start element
Decision	VMs to Process?	Below the Initializing scriptable task element
Scriptable task	Pool Has No VMs	Below the VMs to Process? custom decision element, linked with a red arrow

Element Type	Element Name	Position in Schema
Custom decision	Remaining VMs?	Right of the VMs to Process? custom decision element, linked with a green arrow
Action	getVMDiskModes	Right of the Remaining VMs? custom decision element, linked with a green arrow
Custom decision	Create Snapshot?	Right of the getVMDiskModes action element, linked with a blue arrow
Workflow	Create a snapshot	Above the Create Snapshot? custom decision element, linked with a green arrow
Scriptable task	VM Snapshots	Left of the Create a snapshot workflow, linked with a blue arrow
Scriptable task	Increment	Left of the VM Snapshots scriptable task element, linked with a blue arrow
Scriptable task	Set Output	Right of the Pool Has No VMs scriptable task element, linked with a blue arrow

- 3 Add a Log Exception scriptable task element.
 - a Create an exception handling link from the Create a snapshot workflow to an End element.
 - b Drag a scriptable task element to the red dashed arrow that links the Create a snapshot workflow to an End element.
 - c Double-click the scriptable task element and rename it to **Log Exception**.
 - d Move the Log Exception scriptable task element to above the VM Snapshots scriptable task element.
- 4 Unlink all End elements except the End element that is at the right of the Set Output scriptable task element.
- 5 Link the remaining elements as described in the following table.

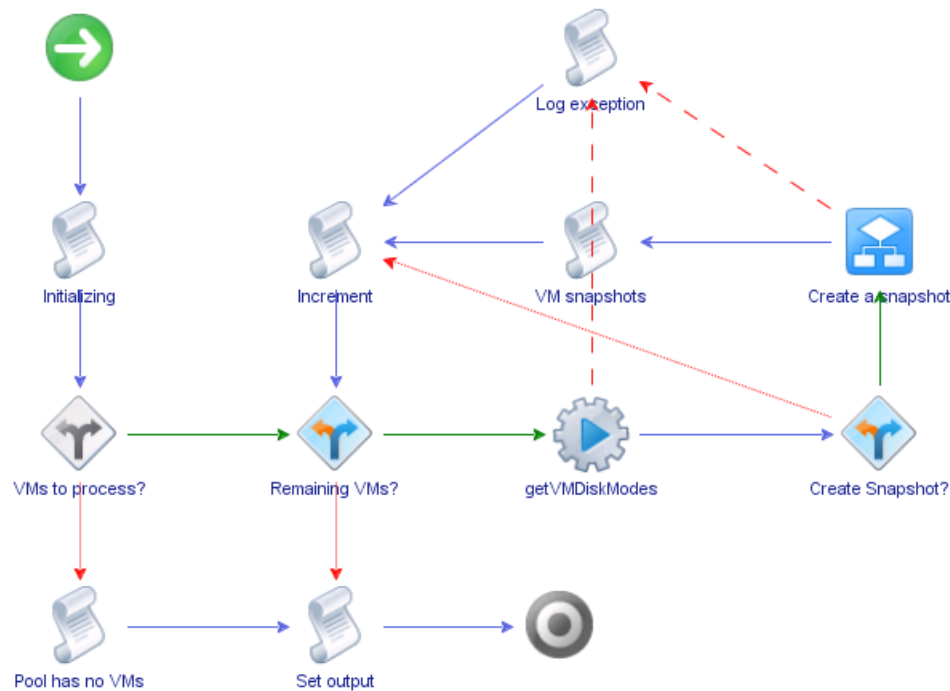
Element	Link to	Type of Arrow	Description
getVMDiskModes action element	Log Exception scriptable task element	Red dashed	Exception handling
Create Snapshot? custom decision element	Increment scriptable task element	Red	False result
Log Exception scriptable task element	Increment scriptable task element	Blue	Normal workflow progression
Increment scriptable task element	Remaining VMs? custom decision element	Blue	Normal workflow progression
Remaining VMs? custom decision element	Set Output scriptable task element	Red	False result

- 6 Click **Save** at the bottom of the **Schema** tab.

Results

The following figure shows what the linked elements of the Take a Snapshot of All Virtual Machines in a Resource Pool workflow should look like.

Figure 1-12. Linking of the Take a Snapshot of All Virtual Machines in a Resource Pool Example Workflow



What to do next

You can optionally define workflow zones by using workflow notes.

Create the Complex Workflow Example Zones

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

Prerequisites

Complete the following tasks.

- [Create the Complex Workflow Example.](#)
- [Create the Schema of the Complex Workflow Example.](#)
- Open the workflow for editing in the workflow editor.

Procedure

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

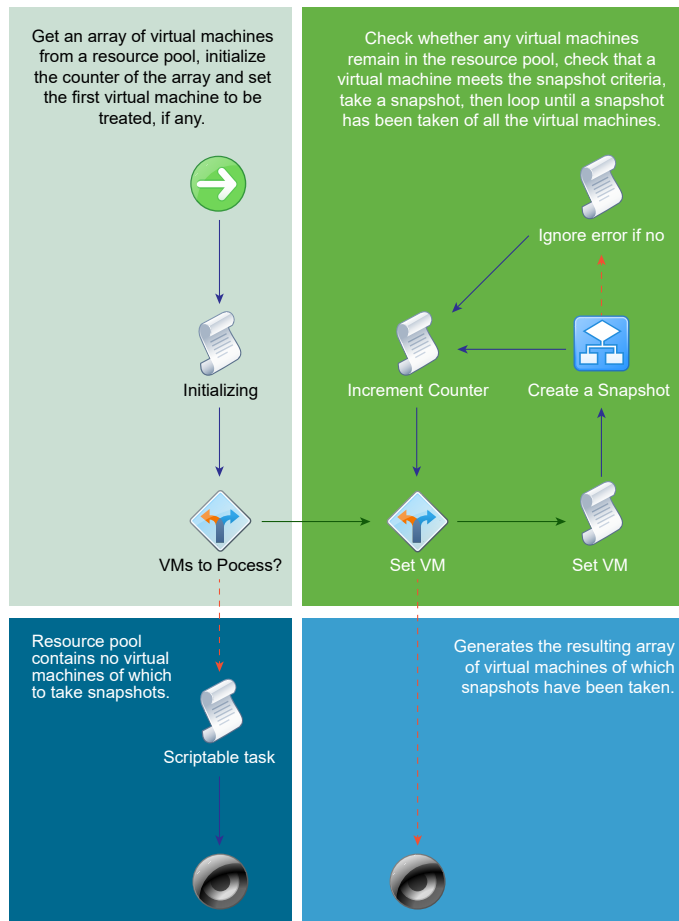
Elements in Zone	Description
Start element; Initialize scriptable task; VMs to Process? custom decision	Get an array of virtual machines from a resource pool, initialize the counter of the Array and set the first virtual machine to be treated, if any.
Pool has no VMs scriptable task.	Resource pool contains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

- 2 Select a workflow note and press Ctrl+E to select the background color.
- 3 Click **Save** at the bottom of the workflow editor **Schema** tab.

Results

Your workflow zones should look like the following diagram.

Figure 1-13. Schema Diagram for Take Snapshot of all Virtual Machines in a Resource Pool
Example Workflow



What to do next

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

Define the Parameters of the Complex Workflow Example

You define workflow parameters in the workflow editor. The input parameters provide data for the workflow to process. The output parameters are the data the workflow returns when it completes its run.

Prerequisites

Complete the following tasks.

- [Create the Complex Workflow Example.](#)
- [Create the Schema of the Complex Workflow Example.](#)
- Open the workflow for editing in the workflow editor.

Procedure

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

Results

You have defined the workflow's input and output parameters.

What to do next

You must define the bindings between the element parameters.

Define the Bindings for the Complex Workflow Example

You can bind a workflow's elements together in the workflow editor. Bindings define the data flow of the workflow. You also bind the scriptable task elements to their JavaScript functions.

Prerequisites

Complete the following tasks.

- [Create the Complex Workflow Example.](#)
- [Create the Schema of the Complex Workflow Example](#)
- [Define the Parameters of the Complex Workflow Example](#)
- Review the bindings that you must define. See [Complex Workflow Example Bindings](#).
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Click the **Schema** tab in the workflow editor.
- 2 Define the bindings.
- 3 Click **Save** at the bottom of the **Schema** tab.

Results

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

What to do next

Set the attribute properties.

Complex Workflow Example Bindings

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

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

In cases in which you bind to existing parameters, the binding inherits the type and description values from the original parameter.

Initializing Scriptable Task

The Initializing scriptable task element initializes the attributes of the workflow. The following table shows the input and output parameter bindings that the Initializing scriptable task element requires.

Table 1-61. Bindings of the Initializing Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
resourcePool	IN	Bind	<ul style="list-style-type: none"> Local parameter: resourcePool Source parameter: resourcePool[in-parameter] Type: VC:ResourcePool Description: The resource pool containing the virtual machines of which to take snapshots
allVMs	OUT	Create	<ul style="list-style-type: none"> Local parameter: allVMs Source parameter: allVMs[attribute] Type: Array/VC:VirtualMachine Description: The virtual machines in the resource pool.

Table 1-61. Bindings of the Initializing Scriptable Task Element (continued)

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
numberOfVms	OUT	Create	<ul style="list-style-type: none"> Local parameter: numberOfVms Source parameter: numberOfVms[attribute] Type: number Description: The number of virtual machines found in the resourcePool
vmCounter	OUT	Create	<ul style="list-style-type: none"> Local parameter: vmCounter Source parameter: vmCounter[attribute] Type: number Description: The counter of the virtual machines inside the array
vm	OUT	Create	<ul style="list-style-type: none"> Local parameter: vm Source parameter: vm[attribute] Type: VC:VirtualMachine Description: The current virtual machine having a snapshot taken
snapshotVmArray	OUT	Create	<ul style="list-style-type: none"> Local parameter: snapshotVmArray Source parameter: snapshotVmArray[attribute] Type: Array/VC:VirtualMachine Description: The Array of virtual machines of which snapshots have been taken

The Initialize scriptable task element performs the following scripted function.

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

VMs to Process? Decision Element

The VMs to Process? decision element determines whether any virtual machines of which to take snapshots exist in the resource pool. The following table shows the bindings that the VMs to Process? decision element requires.

Table 1-62. Bindings of the VMs to Process? Decision Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
numberOfVMs	Decision	Bind	<ul style="list-style-type: none"> ■ Source parameter: numberOfVMs[attribute] ■ Decision statement: Greater than ■ Value: 0.0 ■ Description: The number of virtual machines found in the resourcePool

Pool Has No VMs Scriptable Task Element

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

Table 1-63. Bindings of the Pool Has No VMs Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
resourcePool	IN	Bind	<ul style="list-style-type: none"> ■ Local parameter: resourcePool ■ Source parameter: resourcePool[in-parameter] ■ Type: VC:ResourcePool ■ Description: The resource pool containing the virtual machines of which to take snapshots.

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

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

Remaining VMs? Custom Decision Element

The Remaining VMs? custom decision element determines whether any virtual machines of which to take snapshots remain in the resource pool. The following table shows the bindings that the Remaining VMs? custom decision element requires.

Table 1-64. Bindings of the Remaining VMs? Custom Decision Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
numberOfVMs	IN	Bind	<ul style="list-style-type: none"> ■ Source parameter: numberOfVMs[attribute] ■ Decision statement: Greater than ■ Value: 0.0 ■ Description: The number of virtual machines found in the resourcePool
vmCounter	IN	Bind	<ul style="list-style-type: none"> ■ Local parameter: vmCounter ■ Source parameter: vmCounter[attribute] ■ Type: number ■ Description: The counter of the virtual machines inside the array

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

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

getVMDisksModes Action Element

The getVMDisksModes action element obtains the modes of the disks running in a virtual machine. The following table shows the bindings that the getVMDisksModes action element requires.

Table 1-65. Bindings of the getVMDisksModes Action Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> Local parameter: vm Source parameter: vm[attribute] Type: VC:VirtualMachine Description: The current virtual machine having a snapshot taken
actionResult	OUT	Create	<ul style="list-style-type: none"> Local parameter: actionResult Source parameter: vmDisksModes[attribute] Type: Array/String Description: The current Disks Modes of the virtual machine
errorCode	Exception	Create	Local parameter: errorCode

Create Snapshot? Custom Decision Element

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

Table 1-66. Bindings of the Create Snapshot? Decision Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vmDisksMode	IN	Bind	<ul style="list-style-type: none"> Local parameter: vmDisksMode Source parameter: vmDisksMode[attribute] Type: Array/String Description: The current Disks Modes of the virtual machine
vm	IN	Bind	<ul style="list-style-type: none"> Local parameter: vm Source parameter: vm[attribute] Type: VC:VirtualMachine Description: The current virtual machine having a snapshot taken

The Create Snapshot? custom decision element custom decision element performs the following scripted function.

```
//A snapshot cannot be taken if one of its disks is in independent mode
// (independent-persistent or independent-nonpersistent)
```

```

var containsIndependentDisks = false;
if (vmDisksModes!=null && vmDisksModes.length>0) {
    for (i in vmDisksModes) {
        if (vmDisksModes[i].charAt(0)=='i') {
            containsIndependentDisks = true;
        }
    }
} else {
    //if no disk found no need to try to snapshot the VM
    System.warn("Won't snapshot '"+vm.name+"', no disks found");
    return false;
}
if (containsIndependentDisks) {
    System.warn("Won't snapshot '"+vm.name+"', independent disk(s) found");
    return false;
} else {
    System.log("Snapshotting '"+vm.name+"'");
    return true;
}

```

Create a snapshot Workflow Element

The Create a snapshot workflow element takes snapshots of virtual machines. The following table shows the bindings that the Create a snapshot workflow element requires.

Table 1-67. Bindings of the Create a snapshot Workflow Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> Local parameter: vm Source parameter: vm[attribute] Type: VC:VirtualMachine Description: An active virtual machine of which to take a snapshot.
name	IN	Create	<ul style="list-style-type: none"> Local parameter: name Source parameter: snapshotName[attribute] Type: string Description: The name for this snapshot. The name does not need to be unique for this virtual machine.
description	IN	Create	<ul style="list-style-type: none"> Local parameter: description Source parameter: snapshotDescription[attribute] Type: string Description: A description for this snapshot.

Table 1-67. Bindings of the Create a snapshot Workflow Element (continued)

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
memory	IN	Create	<ul style="list-style-type: none"> Local parameter: memory Source parameter: snapshotMemory[attribute] Type: Boolean Value: no Description: If TRUE, a dump of the internal state of the virtual machine (a memory dump) is included in the snapshot.
quiesce	IN	Create	<ul style="list-style-type: none"> Local parameter: quiesce Source parameter: snapshotQuiesce[attribute] Type: Boolean Value: yes Description: If TRUE and the virtual machine is powered on when the snapshot is taken, the VMware Tools are used to quiesce the file system in the virtual machine.
snapshot	OUT	Create	<ul style="list-style-type: none"> Local parameter: snapshot Source parameter: NULL Type: VC:VirtualMachineSnapshot Description: The snapshot taken.
errorCode	Exception	Create	Local parameter: errorCode

VM Snapshots Scriptable Task Element

The VM Snapshots scriptable task element adds the snapshots to an array. The following table shows the bindings that the VM Snapshots scriptable task element requires.

Table 1-68. Bindings of the VM Snapshots Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> Local parameter: vm Source parameter: vm[attribute] Type: VC:VirtualMachine Description: An active virtual machine of which to take a snapshot.
snapshotVmArray	IN	Bind	<ul style="list-style-type: none"> Local parameter: snapshotVmArray Source parameter: snapshotVmArray[attribute] Type: Array/VC:VirtualMachine Description: The Array of virtual machines of which snapshots have been taken
snapshotVmArray	OUT	Bind	<ul style="list-style-type: none"> Local parameter: snapshotVmArray Source parameter: snapshotVmArray[attribute] Type: Array/VC:VirtualMachine Description: The Array of virtual machines of which snapshots have been taken

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

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

Increment Scriptable Task Element

The Increment scriptable task element increments the counter that counts the number of virtual machines in the array. The following table shows the bindings that the Increment scriptable task element requires.

Table 1-69. Bindings of the Increment Scriptable Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vmCounter	IN	Bind	<ul style="list-style-type: none"> Local parameter: vmCounter Source parameter: vmCounter[attribute] Type: number Description: The counter of the virtual machines inside the array
allVMs	IN	Bind	<ul style="list-style-type: none"> Local parameter: allVMs Source parameter: allVMs[attribute] Type: Array/VC:VirtualMachine Description: The virtual machines in the resource pool.
vmCounter	OUT	Bind	<ul style="list-style-type: none"> Local parameter: vmCounter Source parameter: vmCounter[attribute] Type: number Description: The counter of the virtual machines inside the array
vm	OUT	Bind	<ul style="list-style-type: none"> Local parameter: vm Source parameter: vm[attribute] Type: VC:VirtualMachine Description: The current virtual machine having a snapshot taken

The Increment scriptable task element performs the following scripted function.

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

Log Exception Scriptable Task Element

The Log Exception scriptable task element handles exceptions from the workflow and action elements. The following table shows the bindings that the Log Exception scriptable task element requires.

Table 1-70. Bindings of the Log Exception Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
vm	IN	Bind	<ul style="list-style-type: none"> Local parameter: vm Source parameter: vm[attribute] Type: VC:VirtualMachine Description: The current virtual machine having a snapshot taken
errorCode	IN	Bind	<ul style="list-style-type: none"> Local parameter: errorCode Source parameter: errorCode[attribute] Type: string Description: An exception caught while taking a snapshot of a virtual machine

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

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

Set Output Scriptable Task Element

The Set Output scriptable generates the workflow's output parameter, that contains the array of virtual machines of which snapshots have been taken. The following table shows the bindings that the Set Output scriptable task element requires.

Table 1-71. Bindings of the Set Output Task Element

Parameter Name	Binding Type	Bind to Existing or Create Parameter?	Binding Values
snapshotVmArray	IN	Bind	<ul style="list-style-type: none"> Local parameter: snapshotVmArray Source parameter: snapshotVmArray[attribute] Type: Array/VC:VirtualMachine Description: The Array of virtual machines of which snapshots have been taken
snapshotVmArrayOut	OUT	Bind	<ul style="list-style-type: none"> Local parameter: snapshotVmArrayOut Source parameter: snapshotVmArrayOut[out-parameter] Type: Array/VC:VirtualMachine Description: The Array of virtual machines of which snapshots have been

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

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

Set the Complex Workflow Example Attribute Properties

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

Prerequisites

Complete the following tasks.

- [Create the Complex Workflow Example.](#)
- [Create the Schema of the Complex Workflow Example.](#)
- [Define the Bindings for the Complex Workflow Example.](#)
- Open the workflow for editing in the workflow editor.

Procedure

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

- snapshotQuiesce

Results

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

What to do next

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

Create the Layout of the Complex Workflow Example Input Parameters

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

Prerequisites

Complete the following tasks.

- [Create the Complex Workflow Example.](#)
- [Create the Schema of the Complex Workflow Example.](#)
- [Define the Parameters of the Complex Workflow Example.](#)
- [Define the Bindings for the Complex Workflow Example.](#)
- [Set the Complex Workflow Example Attribute Properties.](#)
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Click the **Presentation** tab in the workflow editor.

The Take a Snapshot of All Virtual Machines in a Resource Pool workflow has only one input parameter, so creating the presentation is straightforward.

- 2 Right-click the **Presentation** node in the presentation hierarchical list and select **Create display group**.
- 3 Delete the **New step** element that appears above the **New group** element.
- 4 Double-click the **New group** element and change the group name to **Resource Pool**.
- 5 Provide a description of the **Resource Pool** display group in the **Description** text box on the **General** tab at the bottom of the **Presentation** tab.

For example,

Enter the name of the resource pool that contains the virtual machines of which to take a snapshot.

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

- 7 Click the **Properties** tab for (VC:ResourcePool)resourcePool.
- 8 Right-click within the **Properties** tab and select **Add Property > Mandatory input**.
- 9 Right-click within the **Properties** tab and select **Add Property > Select value as**.
When you set this property, you set how the user selects the value of the (VC:ResourcePool)resourcePool input parameter.
- 10 Drag the (VC:ResourcePool)resourcePool parameter under the **Resource Pool** display group.

Results

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

What to do next

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

Validate and Run the Complex Workflow Example

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

Prerequisites

Create a workflow, lay out its schema, define the links and bindings, define the parameter properties, and create the presentation of the input parameters dialog box.

Complete the following tasks.

- [Create the Complex Workflow Example.](#)
- [Create a Custom Action for the Complex Workflow Example.](#)
- [Create the Schema of the Complex Workflow Example.](#)
- [Define the Parameters of the Complex Workflow Example.](#)
- [Define the Bindings for the Complex Workflow Example.](#)
- [Set the Complex Workflow Example Attribute Properties.](#)
- [Create the Layout of the Complex Workflow Example Input Parameters.](#)
- Open the workflow for editing in the workflow editor.

Procedure

- 1 Click **Validation** in the **Schema** tab of the workflow editor.

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

- 2 After you have eliminated any errors, click **Save and Close** at the bottom of the workflow editor.

You return to the Orchestrator client.

- 3 Click the **Workflows** view.
- 4 In the workflow hierarchical list, select **Workflow Examples > Take a Snapshot of All Virtual Machines in a Resource Pool**.
- 5 Right-click the **Take a Snapshot of All Virtual Machines in a Resource Pool** workflow and select **Start workflow**.

The input parameters dialog box opens and prompts you for a resource pool that contains the virtual machines of which to take a snapshot.

- 6 Click **Submit** to run the workflow.

A workflow token appears under the Take a Snapshot of All Virtual Machines in a Resource Pool workflow.

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

Results

If the workflow runs successfully, the workflow takes a snapshot of all of the virtual machines in the selected resource pool.

What to do next

You can generate a document in which to review information about the workflow. See [Generate Workflow Documentation](#).

Scripting

2

Orchestrator uses JavaScript to create building blocks from which you create actions, workflow elements, and policies that access the APIs of the technologies that you plug into Orchestrator.

Orchestrator uses the Mozilla Rhino 1.7R4 JavaScript engine as its scripting engine. The scripting engine provides variable type checking, name space management, automatic completion, and exception handling.

The Orchestrator workflow engine allows you to use basic JavaScript language features, such as if, loops, arrays, and strings. You can use objects in scripting that the Orchestrator API provides, or objects from any other API that you import into Orchestrator through a plug-in and that you map to JavaScript objects. For information about Rhino, see the Mozilla Rhino Web site.

This chapter includes the following topics:

- [Orchestrator Elements that Require Scripting](#)
- [Limitations of the Mozilla Rhino Implementation in Orchestrator](#)
- [Using the Orchestrator Scripting API](#)
- [Using XPath Expressions with the vCenter Server Plug-In](#)
- [Exception Handling Guidelines](#)
- [Orchestrator JavaScript Examples](#)

Orchestrator Elements that Require Scripting

Not all Orchestrator elements require you to write scripts. To provide maximum flexibility to your applications, you can customize certain elements by adding JavaScript functions.

You can add scripts in the following Orchestrator elements.

Actions

Actions are scripted functions. You can limit the scripting you write for an action to a single operation, to maximize the potential for action reuse by other elements, such as other

workflows. Alternatively, an action can contain many operations, to limit the complexity of workflows, although this does reduce the capacity for reusing the action.

Policies

You set policies by using scripts that watch for trigger events. When the trigger events occur, policies launch orchestration operations that you define in scripts.

Workflows

The Scriptable Task workflow element allows you to write a custom scripted operation or sequence of operations that you can use in the workflows. You also define the Boolean decision statement for custom decision elements in scripts that return either `true` or `false`.

Limitations of the Mozilla Rhino Implementation in Orchestrator

Orchestrator uses the Mozilla Rhino 1.7R4 JavaScript engine. However, the implementation of Rhino in Orchestrator presents some limitations.

When writing scripts for workflows, you must consider the following limitations of the Mozilla Rhino implementation in Orchestrator.

- When a workflow runs, the objects that pass from one workflow element to another are not JavaScript objects. What is passed from one element to the next is the serialization of a Java object that has a JavaScript image. As a consequence, you cannot use the whole JavaScript language, but only the classes that are present in the API Explorer. You cannot pass function objects from one workflow element to another.
- Orchestrator runs the code in scriptable task elements in a context that is not the Rhino root context. Orchestrator transparently wraps scriptable task elements and actions into JavaScript functions, which it then runs. A scriptable task element that contains `System.log(this);` does not display the global object `this` in the same way as a standard Rhino implementation does.
- You can only call actions that return nonserializable objects from scripting, and not from workflows. To call an action that returns a nonserializable object, you must write a scriptable task element that calls the action by using the `System.getModuleModuleName.action()` method.
- Workflow validation does not check whether a workflow attribute type is different from an input type of an action or subworkflow. If you change the type of a workflow input parameter, for example from `VIM3:VirtualMachine` to `VC:VirtualMachine`, but you do not update any scriptable tasks or actions that use the original input type, the workflow validates but does not run.

Using the Orchestrator Scripting API

The Orchestrator API exposes all of the objects and functions of the technologies, that Orchestrator accesses through its plug-ins, as JavaScript objects and methods.

For example, you can access JavaScript implementations of the vCenter Server API through the Orchestrator API, to include vCenter operations in scripted elements that you create. You can also access JavaScript implementations of objects from all of the other plug-ins you install in the Orchestrator server. If you create a custom plug-in to a third-party application, you map the objects from its API to JavaScript objects that the Orchestrator API then exposes.

Procedure

1 [Access the Scripting Engine from the Workflow Editor](#)

The Orchestrator scripting engine uses the Mozilla Rhino 1.7R4 JavaScript engine to help you write scripts for scripted elements in workflows. You access the scripting engine for scripted workflow elements from the **Scripting** tab in the workflow editor.

2 [Access the Scripting Engine from the Action or Policy Editor](#)

The Orchestrator scripting engine uses the Mozilla Rhino JavaScript engine to help you write scripts for actions or policies. You access the scripting engine for actions and policies from the **Scripting** tabs in the action and policy editors.

3 [Access the Orchestrator API Explorer](#)

Orchestrator provides an API Explorer that you can use to search the Orchestrator API and see the documentation for JavaScript objects that you can use in scripted elements.

4 [Use the Orchestrator API Explorer to Find Objects](#)

The Orchestrator API exposes the API of all plugged-in technologies, including the entire vCenter Server API. The Orchestrator API Explorer helps you find the objects you need to add to scripts.

5 [Writing Scripts](#)

The Orchestrator scripting engine helps you to write scripts. Automatic insertion of functions and automatic completion of lines of scripting accelerates the scripting process and minimizes the potential for writing errors in scripts.

6 [Add Parameters to Scripts](#)

The Orchestrator scripting engine helps you to import available parameters into scripts.

7 [Accessing the Orchestrator Server File System from JavaScript and Workflows](#)

Orchestrator limits access to the Orchestrator server file system from JavaScript and Workflows to specific directories.

8 [Accessing Java Classes from JavaScript](#)

By default, Orchestrator restricts JavaScript access to a limited set of Java classes. If you require JavaScript access to a wider range of Java classes, you must set an Orchestrator system property to allow this access.

9 Accessing Operating System Commands from JavaScript

The Orchestrator API provides a scripting class, `Command`, that runs commands in the Orchestrator server host operating system. To prevent unauthorized access to the Orchestrator server host, by default, Orchestrator applications do not have permission to run the `Command` class.

Access the Scripting Engine from the Workflow Editor

The Orchestrator scripting engine uses the Mozilla Rhino 1.7R4 JavaScript engine to help you write scripts for scripted elements in workflows. You access the scripting engine for scripted workflow elements from the **Scripting** tab in the workflow editor.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Right-click a workflow in the **Workflows** view of the Orchestrator client and select **Edit**.
- 3 Click the **Schema** tab in the workflows editor.
- 4 Add a Scriptable Task element or a Custom Decision element to the workflow schema.
- 5 Click on the scriptable element's **Scripting** tab.

Results

You accessed the scripting engine to define the scripted functions of workflow elements. The **Scripting** tab allows you to navigate through the API, consult documentation about the objects, search for objects, and write JavaScript.

What to do next

Search the Orchestrator API using the API Explorer.

Access the Scripting Engine from the Action or Policy Editor

The Orchestrator scripting engine uses the Mozilla Rhino JavaScript engine to help you write scripts for actions or policies. You access the scripting engine for actions and policies from the **Scripting** tabs in the action and policy editors.

Procedure

- 1 Select an option from the drop-down menu in the Orchestrator client, depending on the type of the element whose scripting you want to edit.

Option	Description
Design	Select this option to edit the scripting of an action element.
Run	Select this option to edit the scripting of a policy.

- 2 Right-click an action or policy in the **Actions** or **Policies** views and select **Edit**.
- 3 Click the **Scripting** tab in the action or policy editor.

Results

You accessed the scripting engine to define the scripted functions of action or policy elements. The **Scripting** tab allows you to navigate through the API, consult documentation about the objects, search for objects, and write JavaScript.

What to do next

Search the Orchestrator API using the API Explorer.

Access the Orchestrator API Explorer

Orchestrator provides an API Explorer that you can use to search the Orchestrator API and see the documentation for JavaScript objects that you can use in scripted elements.

You can consult an online version of the Scripting API for the vCenter Server plug-in on the Orchestrator documentation home page.

Procedure

- 1 Log in to the Orchestrator client.
- 2 Select **Tools > API Explorer**.

Results

The API Explorer appears. You can use it to search all the objects and functions of the Orchestrator API.

What to do next

Use the API Explorer to write scripts for scriptable elements.

Use the Orchestrator API Explorer to Find Objects

The Orchestrator API exposes the API of all plugged-in technologies, including the entire vCenter Server API. The Orchestrator API Explorer helps you find the objects you need to add to scripts.

Prerequisites

Open the API Explorer.

Procedure

- 1 Enter the name or part of a name of an object in the API Explorer **Search** text box and click **Search**.

To limit your search to a particular object type, uncheck or check the **Scripting Class**, **Attributes & Methods** and **Types & Enumerations** check boxes.

- 2 Double-click the element in the proposed list.

The object is highlighted in the hierarchical list on the left. A documentation pane under the hierarchical list presents information about the object.

What to do next











Use the objects you find in scripts.

JavaScript Objects in the API Explorer

The Orchestrator API Explorer identifies and groups together the different kinds of JavaScript objects in the hierarchical tree on the left of the **Scripting** tab or API Explorer dialog box. The API Explorer uses icons to help you identify the different kinds of object.

The following table describes the objects of the Orchestrator API and shows their icon.

Table 2-1. JavaScript Objects in the Orchestrator API

Object	Icon in Hierarchical List	Description
Type		Types
Function set		Internal type that contains a set of static methods
Primitive		Primitive types
Object		Standard Orchestrator scripting objects
Attribute		JavaScript attributes
Method		JavaScript methods
Constructor		JavaScript constructors
Enumeration		JavaScript enumerations
String set		String set, default values
Module		A collection of actions
Plug-in	Image that plug-in defines	The APIs that plug-ins expose to Orchestrator

Writing Scripts

The Orchestrator scripting engine helps you to write scripts. Automatic insertion of functions and automatic completion of lines of scripting accelerates the scripting process and minimizes the potential for writing errors in scripts.

Prerequisites

Open a scripted element for editing and click its **Scripting** tab.

Procedure

- 1 Navigate through the hierarchical list of objects on the left of the **Scripting** tab, or use the API Explorer search function, to select a type, class, or method to add to the script.

2 Right-click the type, class, or method and select **Copy**.

If the scripting engine does not allow you to copy the element you selected, this object is not possible in the context of the script.

3 Right-click in the scripting pad, and paste the element you copied into the appropriate place in the script.

The scripting engine enters the element into the script, complete with its constructor and an instance name.

For example, if you copied the `Date` object, the scripting engine pastes the following code into the script.

```
var myDate = new Date();
```

4 Copy and paste a method to add to the script.

The scripting engine completes the method call, adding the required attributes.

For example, if you copied the `cloneVM()` method from the `com.vmware.library.vc.vm` module, the scripting engine pastes the following code into the script.

```
System.getModule("com.vmware.library.vc.vm").cloneVM(vm,folder,name,spec)
```

The scripting engine highlights those parameters that you already defined in the element. Any undefined parameters remain unhighlighted.

5 Place the cursor at the end of an element you pasted into the script and press Ctrl+space to select from a contextual list of possible methods and attributes that the object can call.

Note The automatic completion feature is currently experimental.

Results

You added object and functions to the script.

What to do next

Add parameters to the script.

Color Coding of Scripting Keywords

When you add scripts on the **Scripting** tab of a scripted workflow element, certain types of keywords appear in different colors to enhance the readability of the code.

All scripting appears in standard black font unless stated otherwise.

Table 2-2. Color Coding of Scripting Keywords

Keyword Type	Text Color in Scripting Tab
Standard JavaScript keywords, for example <code>if</code> , <code>else</code> , <code>for</code> , and <code>new</code>	Bold black
Variable declarations, namely <code>var</code>	Green

Table 2-2. Color Coding of Scripting Keywords (continued)

Keyword Type	Text Color in Scripting Tab
Modifiers in loops, for example <code>in</code>	Red
Null variable values	Purple
Non-null variable values	Green
Code comments	Italic gray
Orchestrator plug-in object types, for example <code>VC:VirtualMachine</code> or <code>VC:Host</code>	Green
Output text	Green
Workflow attributes	Pink
Workflow inputs	Pink
Workflow outputs	Pink

Add Parameters to Scripts

The Orchestrator scripting engine helps you to import available parameters into scripts.

If you have already defined parameters for the element you are editing, they appear as links in the **Scripting** tab toolbar.

Prerequisites

A scripted element is open for editing and its **Scripting** tab is open.

Procedure

- 1 Move the cursor to the appropriate position in a script in the scripting pad of the **Scripting** tab.
- 2 Click the parameter link in the **Scripting** tab toolbar.
Orchestrator inserts the parameter at the position of the cursor.
- 3 Insert a parameter with a null value into the script.
If you pass null values to primitive types such as integers, Booleans, and Strings, the Orchestrator scripting API automatically sets the default value for this argument.

Results

You added parameters to the script.

What to do next

Add access to Java classes in scripts.

Accessing the Orchestrator Server File System from JavaScript and Workflows

Orchestrator limits access to the Orchestrator server file system from JavaScript and Workflows to specific directories.

JavaScript functions and workflows only have read, write, and execute permission in the permanent directory `c:\orchestrator`.

The Orchestrator administrator can modify the folders to which JavaScript functions and workflows have read, write, and execute access by setting a system property. See *Installing and Configuring VMware vRealize Orchestrator* for information about setting system properties.

JavaScript functions and workflows also have read, write, and execute permission in the server system default temporary I/O folder. Writing to the default temporary I/O folder is the only portable, guaranteed, and configuration-independent means of accessing the file system with full permissions. However, files that you write to the temporary I/O folder are lost when you reboot the server.

You obtain the default temporary I/O folder by calling the `System.getTempDirectory` method in JavaScript functions.

Access the Server File System Using the `System.getTempDirectory` Method

As an alternative to writing to the folders on the Orchestrator server system in which the administrator has set the appropriate permissions, you can write to the default temporary I/O folder.

Orchestrator has full read, write, and execute rights in the default temporary I/O folder by default. You obtain the default temporary I/O folder by using the `System.getTempDirectory` method in JavaScript functions

Procedure

- ◆ Include the following code line in JavaScript functions to access the `java.io.temp-dir` folder.

```
var tempDir = System.getTempDirectory()
```

Accessing Java Classes from JavaScript

By default, Orchestrator restricts JavaScript access to a limited set of Java classes. If you require JavaScript access to a wider range of Java classes, you must set an Orchestrator system property to allow this access.

By default, the Orchestrator JavaScript engine can access only the classes in the `java.util.*` package.

The Orchestrator administrator can allow access to other Java classes from JavaScript functions by setting a system property. See *Installing and Configuring VMware vRealize Orchestrator* for information about setting system properties.

Accessing Operating System Commands from JavaScript

The Orchestrator API provides a scripting class, `Command`, that runs commands in the Orchestrator server host operating system. To prevent unauthorized access to the Orchestrator server host, by default, Orchestrator applications do not have permission to run the `Command` class.

The Orchestrator administrator can allow access to the `Command` scripting class by setting the `com.vmware.js.allow-local-process=true` system property.

For information about setting system properties, see the *Installing and Configuring VMware vCenter Orchestrator*.

For information about setting system properties, see *Installing and Configuring VMware vCenter Orchestrator*.

Using XPath Expressions with the vCenter Server Plug-In

You can use the finder methods in the vCenter Server plug-in to query for vCenter Server inventory objects. You can use XPath expressions to define search parameters.

The vCenter Server plug-in includes a set of object finder methods such as `getAllDatastores()`, `getAllResourcePools()`, `findAllForType()`. You can use these methods to access the inventories of the vCenter Server instances that are connected to your Orchestrator server and search for objects by ID, name, or other properties.

For performance reasons, the finder methods do not return any properties for the queried objects, unless you specify a set of properties in the search query.

You can consult an online version of the Scripting API for the vCenter Server plug-in on the Orchestrator documentation home page.

Important The queries based on XPath expressions might impact the Orchestrator performance because the finder method returns all objects of a given type on the vCenter Server side and the query filters are applied on the vCenter Server plug-in side.

Using XPath Expressions with the vCenter Server Plug-In

When you invoke a finder method, you can use expressions based on the XPath query language. The search returns all the inventory objects that match the XPath expressions. If you want to query for any properties, you can include them to the search script in the form of a string array.

The following JavaScript example uses the `VcPlugin` scripting object and an XPath expression to return the names of all datastore objects that are part of the vCenter Server managed objects and contain the string **ds** in their names.

```
var datastores = VcPlugin.getAllDatastores(null, "xpath:name[contains(.,'ds')]");
for each (datastore in datastores){
    System.log(datastore.name);
}
```


The same XPath expression can be invoked by using the Server scripting object and the `findAllForType` finder method.

```
var datastores = Server.findAllForType("VC:Datastore", "xpath:name[contains(.,'ds')]");
for each (datastore in datastores){
    System.log(datastore.name);
}
```

The following script example returns the names of all host system objects whose ID starts with the digit **1**.

```
var hosts = VcPlugin.getAllHostSystems(null, "xpath:id[starts-with(.,'1')]");
for each (host in hosts){
    System.log(host.name);
}
```

The following script returns the names and IDs of all data center objects that contain the string **DC**, in upper- or lower-case letters, in their names. The script also retrieves the **tag** property.

```
var datacenters = VcPlugin.getAllDatacenters(['tag'], "xpath:name[contains(translate(., 'DC', 'dc'), 'dc')]");
for each (datacenter in datacenters){
    System.log(datacenter.name + " " + datacenter.id);
}
```

Exception Handling Guidelines

The Orchestrator implementation of the Mozilla Rhino JavaScript Engine supports exception handling, to allow you to process errors. You must use the following guidelines when writing exception handlers in scripts.

- Use the following European Computer Manufacturers Association (ECMA) error types. Use `Error` as a generic exception that plug-in functions return, and the following specific error types.
 - `TypeError`
 - `RangeError`
 - `EvalError`
 - `ReferenceError`
 - `URIError`
 - `SyntaxError`

The following example shows a `URIError` definition.

```
try {
    ...
    throw new URIError("VirtualMachine with ID 'vm-0056'
                        not found on 'vcenter-test-1'");
}
```

```
...
} catch ( e if e instanceof URIError ) {

}
```

- All exceptions that scripts do not catch must be simple string objects of the form <type>:SPACE<human readable message>, as the following example shows.

```
throw "ValidationError: The input parameter 'myParam' of type 'string' is too short."
```

- Write human readable messages as clearly as possible.
- Simple string exception type checking must use the following pattern.

```
try {
    throw "VMwareNoSpaceLeftOnDatastore: Datastore 'myDatastore' has no space left" ;
} catch ( e if (typeof(e)=="string" && e.indexOf("VMwareNoSpaceLeftOnDatastore:") == 0) ) {
    System.log("No space left on device") ;
    // Do something useful here
}
```

- Simple string exception type checking, must use the following pattern in scripted elements in workflows.

```
if (typeof(errorCode)=="string"
    && errorCode.indexOf("VMwareNoSpaceLeftOnDatastore:")
    == 0) {
    // Do something useful here
}
```

Orchestrator JavaScript Examples

You can cut, paste, and adapt the Orchestrator JavaScript examples to help you write JavaScripts for common orchestration tasks.

- [Basic Scripting Examples](#)

Workflow scripted elements, actions, and policies require basic scripting of common tasks. You can cut, paste, and adapt these examples into your scripted elements.

- [Email Scripting Examples](#)

Workflow scripted elements can include scripting of common email-related tasks. You can cut, paste, and adapt these examples into your scripted elements.

- [File System Scripting Examples](#)

Workflow scripted elements, actions, and policies require scripting of common file system tasks. You can cut, paste, and adapt these examples into your scripted elements.

- [LDAP Scripting Examples](#)

Workflow scripted elements, actions, and policies require scripting of common LDAP tasks. You can cut, paste, and adapt these examples into your scripted elements.

- [Logging Scripting Examples](#)

Workflow scripted elements, actions, and policies require scripting of common logging tasks. You can cut, paste, and adapt these examples into your scripted elements.

- [Networking Scripting Examples](#)

Workflow scripted elements, actions, and policies require scripting of common networking tasks. You can cut, paste, and adapt these examples into your scripted elements.

- [Workflow Scripting Examples](#)

Workflow scripted elements, actions, and policies require scripting examples of common workflow tasks. You can cut, paste, and adapt these examples into your scripted elements.

Basic Scripting Examples

Workflow scripted elements, actions, and policies require basic scripting of common tasks. You can cut, paste, and adapt these examples into your scripted elements.

Access XML Documents

The following JavaScript example allows you to access XML documents from JavaScript by using the ECMAScript for XML (E4X) implementation in the Orchestrator JavaScript API.

Note In addition to implementing E4X in the JavaScript API, Orchestrator also provides a Document Object Model (DOM) XML implementation in the XML plug-in. For information about the XML plug-in and its sample workflows, see the *Using vRealize Orchestrator Plug-Ins*.

```
var people = <people>
    <person id="1">
        <name>Moe</name>
    </person>
    <person id="2">
        <name>Larry</name>
    </person>
</people>;

System.log("'people' = " + people);

// built-in XML type
System.log("'people' is of type : " + typeof(people));

// list-like interface System.log("which contains a list of " +
people.person.length() + " persons");
System.log("whose first element is : " + people.person[0]);

// attribute 'id' is mapped to field '@id'
people.person[0].@id='47';
// change Moe's id to 47
// also supports search by constraints
System.log("Moe's id is now : " + people.person.(name=='Moe').@id);

// suppress Moe from the list
```

```

delete people.person[0];
System.log("Moe is now removed.");

// new (sub-)document can be built from a string
people.person[1] = new XML("<person id=\"3\"><name>James</name></person>");
System.log("Added James to the list, which is now :");
for each(var person in people..person)

for each(var person in people..person){
    System.log("- " + person.name + " (id=" + person.@id + ")");
}

```

Setting and Obtaining Properties from a Hashtable

The following JavaScript example sets properties in a hashtable and obtains the properties from the hashtable. In the following example, the key is always a String and the value is an object, a number, a Boolean, or a String.

```

var table = new Properties() ;
table.put("myKey",new Date()) ;
// get the object back
var myDate= table.get("myKey") ;
System.log("Date is : "+myDate) ;

```

Replace the Contents of a String

The following JavaScript example replaces the content of a String and replaces it with new content.

```

var str1 = "'hello'" ;
var reg = new RegExp("'", "g");
var str2 = str1.replace(reg,"\\'") ;
System.log(str2) ; // result : \'hello\'

```

Compare Types

The following JavaScript example checks whether an object matches a given object type.

```

var path = 'myurl/test';
if(typeof(path, string)){
    throw("string");
} else {
    throw("other");
}

```

Run a Command in the Orchestrator Server

The following JavaScript example allows you to run a command line on the Orchestrator server. Use the same credentials as those used to start the server.

Note Access to the file system is limited by default.

```
var cmd = new Command("ls -al") ;
cmd.execute(true) ;
System.log(cmd.output) ;
```

Email Scripting Examples

Workflow scripted elements can include scripting of common email-related tasks. You can cut, paste, and adapt these examples into your scripted elements.

When you run a mail workflow, it uses the default mail server configuration that you set in the Configure mail workflow. You can override the default values by using input parameters, or by defining custom values in workflow scripted elements.

Obtain an Email Address

The following JavaScript example obtains the email address of the current owner of a running script.

```
var emailAddress = Server.getRunningUser().emailAddress ;
```

Send an Email

The following JavaScript example sends an email to the defined recipient, through an SMTP server, with the defined content.

```
var message = new EmailMessage() ;
message.smtpHost = "smtpHost" ;
message.subject= "my subject" ;
message.toAddress = "receiver@vmware.com" ;
message.fromAddress = "sender@vmware.com" ;
message.addMimePart("This is a simple message","text/html") ;
message.sendMessage() ;
```

Retrieve Email Messages

The following JavaScript example retrieves the messages of an email account, without deleting them, by using the scripting API provided by the MailClient class.

```
var myMailClient = new MailClient();

myMailClient.setProtocol(mailProtocol);
if(useSSL){
    myMailClient.enableSSL();
}
```

```

myMailClient.connect( mailServer, mailPort, mailUsername, mailPassword);
System.log("Successfully login!");

try {
    myMailClient.openFolder("Inbox");

    var messages = myMailClient.getMessages();
    System.log("Reading messages...!");
    if ( messages != null && messages.length > 0 ) {
        System.log( "You have " + messages.length + " email(s) in your inbox" );
        for (i = 0; i < messages.length; i++) {
            System.log("");
            System.log("-----MSG-----");
            System.log("Headers: ");
            var headerProp = messages[i].getHeaders();
            for each(key in headerProp.keys){
                System.log(key+": "+headerProp.get(key));
            }
            System.log("");

            System.log( "Message["+ i +"] with from: " + messages[i].from + " to: " + messages[i].to);
            System.log( "Message["+ i +"] with subject: " + messages[i].subject);
            var content = messages[i].getContent();
            System.log("Msg content as string: " + content);
        }
    } else {
        System.warn( "No messages found" );
    }
} finally {
    myMailClient.closeFolder();
    myMailClient.close();
}

```

File System Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common file system tasks. You can cut, paste, and adapt these examples into your scripted elements.

Add Content to a Simple Text File

The following JavaScript example adds content to a text file.

```

var tempDir = System.getTempDirectory() ;
var fileWriter = new FileWriter(tempDir + "/readme.txt") ;
fileWriter.open() ;
fileWriter.writeLine("File written at : "+new Date()) ;
fileWriter.writeLine("Another line") ;
fileWriter.close() ;

```

Obtain the Contents of a File

The following JavaScript example obtains the contents of a file from the Orchestrator server host machine.

```
var tempDir = System.getTempDirectory() ;
var fileReader = new FileReader(tempDir + "/readme.txt") ;
fileReader.open() ;
var fileContentAsString = fileReader.readAll();
fileReader.close() ;
```

LDAP Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common LDAP tasks. You can cut, paste, and adapt these examples into your scripted elements.

Convert LDAP Objects to Active Directory Objects

The following JavaScript example converts LDAP group elements to Active Directory user group objects, and the reverse.

```
var ldapGroup ;
// convert from ldap element to Microsoft:UserGroup object
var adGroup = ActiveDirectory.search("UserGroup",ldapGroup.commonName) ;
// convert back to LdapGroup element
var ldapElement = Server.getLdapElement(adGroup.distinguishedName) ;
```

Logging Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common logging tasks. You can cut, paste, and adapt these examples into your scripted elements.

Persistent Logging

The following JavaScript example creates persistent log entries.

```
Server.log("This is a persistant message", "enter a long description here");
Server.warn("This is a persistant warning", "enter a long description here");
Server.error("This is a persistant error", "enter a long description here");
```

Non-Persistent Logging

The following JavaScript example creates non-persistent log entries.

```
System.log("This is a non-persistant log message");
System.warn("This is a non-persistant log warning");
System.error("This is a non-persistant log error");
```

Networking Scripting Examples

Workflow scripted elements, actions, and policies require scripting of common networking tasks. You can cut, paste, and adapt these examples into your scripted elements.

Obtain Text from a URL

The following JavaScript example accesses a URL, obtains text, and converts it to a string.

```
var url = new URL("http://www.vmware.com") ;
var htmlContentAsString = url.getContent() ;
```

Workflow Scripting Examples

Workflow scripted elements, actions, and policies require scripting examples of common workflow tasks. You can cut, paste, and adapt these examples into your scripted elements.

Return All Workflows Run by the Current User

The following JavaScript example obtains all workflow runs from the server and checks whether they belong to the current user.

```
var allTokens = Server.findAllForType('WorkflowToken');
var currentUser = Server.getCredential().username;
var res = [];
for(var i = 0; i<res.length; i++){
    if(allTokens[i].runningUserName == currentUser){
        res.push(allTokens[i]);
    }
}
return res;
```

Access the Current Workflow Token

You can access the current workflow token by using the `workflow` variable. It is an object of type `WorkflowToken` that provides access to the current workflow run. The following JavaScript example gets the ID of the workflow token and its start date.

```
System.log("Current workflow run ID: " + workflow.id);
System.log("Current workflow run start date: "+workflow.startDate);
```

Schedule a Workflow

The following JavaScript example starts a workflow with a given set of properties, and then schedules it to start one hour later.

```
var workflowToLaunch = myWorkflow ;
// create parameters
var workflowParameters = new Properties() ;
workflowParameters.put("name","John Doe") ;
// change the task name
```



```

workflowParameters.put("__taskName","Workflow for John Doe") ;

// create scheduling date one hour in the future
var workflowScheduleDate = new Date() ;
var time = workflowScheduleDate.getTime() + (60*60*1000) ;
workflowScheduleDate.setTime(time) ; var scheduledTask =
workflowToLaunch.schedule(workflowParameters,workflowScheduleDate);

```

Run a Workflow on a Selection of Objects in a Loop

The following JavaScript example takes the array of virtual machines and runs a workflow on each one in a For loop. VMs and workflowToRun are workflow inputs.

```

var len=VMs.length;
for (var i=0; i < len; i++ )
{
    var VM = VMs[i];
    //var workflowToLaunch = Server.getWorkflowWithId("workflowId");
    var workflowToLaunch = workflowToRun;
    if (workflowToLaunch == null) {
        throw "Workflow not found";
    }
    var workflowParameters = new Properties();
    workflowParameters.put("vm",VM);
    var wfToken = workflowToLaunch.execute(workflowParameters);
    System.log ("Ran workflow on " +VM.name);
}

```

Developing Actions

3

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

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

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

This chapter includes the following topics:

- [Reusing Actions](#)
- [Access the Actions View](#)
- [Components of the Actions View](#)
- [Creating Actions](#)
- [Use Action Version History](#)
- [Restore Deleted Actions](#)

Reusing Actions

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

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

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

Access the Actions View

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

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

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Actions** view.
- 3 Browse the libraries of actions by expanding the nodes of the actions hierarchical list.

Results

You can use the **Actions** view to view information about the actions in the libraries and create and edit actions.

Components of the Actions View

When you click an action in the actions hierarchical list, information about that action appears in the Orchestrator client's right pane.

The **Actions** view presents three tabs.

General

Displays general information about the action, including its name, its version number, and a description.

Scripting

Shows the action's return types, input parameters, and the JavaScript code that defines the action's function.

Events

Shows all the events that this action encountered or triggered.

Creating Actions

You can define individual functions as actions that other elements, such as workflows, can use. Actions are JavaScript functions with defined input and output parameters.

■ [Create an Action](#)

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

■ Find Elements That Implement an Action

If you edit an action and change its behavior, you might inadvertently break a workflow or application that implements that action. Orchestrator provides a function to find all of the actions, workflows, or packages that implement a given element. You can check whether modifying the element affects the operation of other elements.

■ Action Coding Guidelines

To optimize the performance of workflows and to maximize the potential to reuse actions, you should follow some basic coding guidelines when creating actions.

Create an Action

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

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Actions** view.
- 3 Expand the root of the actions hierarchical list and navigate to the module in which you want to create the action.
- 4 Right-click the module and select **Add action**.
- 5 Enter a name for the action in the text box and click **OK**.
Your custom action is added to the library of actions.
- 6 Right-click the action and select **Edit**.
- 7 Click the **Scripting** tab.
- 8 To change the default return type, click the **void** link.
- 9 Add the action input parameters by clicking the arrow icon.
- 10 Write the action script.
- 11 Click **Save and close**.

Results

You created a custom action and added the action input parameters.

What to do next

You can use the new custom action in a workflow.

Find Elements That Implement an Action

If you edit an action and change its behavior, you might inadvertently break a workflow or application that implements that action. Orchestrator provides a function to find all of the actions,

workflows, or packages that implement a given element. You can check whether modifying the element affects the operation of other elements.

Important The **Find Elements that Use this Element** function checks all packages, workflows, and policies, but it does not check in scripts. Consequently, modifying an action might affect an element that calls this action in a script that the **Find Elements that Use this Element** function did not identify.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Actions** view.
- 3 Expand the nodes of the actions hierarchical list to navigate to a given action.
- 4 Right-click the action and select **Find Elements that Use this Element**.

A dialog box shows all of the elements, such as workflows or packages, that implement this action.

- 5 Double-click an element in the list of results to show that element in the Orchestrator client.

Results

You located all of the elements that implement an action.

What to do next

You can check whether modifying this element affects any other elements.

Action Coding Guidelines

To optimize the performance of workflows and to maximize the potential to reuse actions, you should follow some basic coding guidelines when creating actions.

Basic Action Guidelines

When you create an action, you must use basic guidelines.

- Every action must include a description of its role and function.
- Write short, elementary actions and combine them in a workflow.
- Avoid writing actions that perform multiple functions, because this limits the potential for reusing the action.
- Avoid actions that run for long periods of time. Instead, create a loop in the workflow and include a Waiting Event or Waiting Timer element after the action element.
- Do not write check points in actions. Workflows set a check point at the start and end of each element's run.

- Avoid writing loops in an action. Create loops in the workflow instead. If the server restarts, a running workflow resumes at its last check point, at the start of an element. If you write a loop inside an action and the server restarts while the workflow is running that action, the workflow resumes at the check point at the beginning of that action, and the loop starts again from the beginning.

Action Naming Guidelines

Use basic guidelines when you name actions.

- Write action names in English.
- Start action names with a lowercase letter. Use an uppercase letter at the beginning of each conjoined word in the name. For example, `myAction`.
- Make action names as explicit as possible, so that the function of the action is clear. For example, `backupAllVMsInPool`.
- Make module names as explicit as possible.
- Make module names unique.
- Use the inverse Internet address format for module names. For example, `com.vmware.myactions.myAction`.

Action Parameter Guidelines

Use basic guidelines when you write action parameter definitions.

- Write parameter names in English.
- Start parameter names with a lowercase letter.
- Make parameter names as explicit as possible.
- Preferably limit parameter names to a single word. If a name must contain more than one word, use an uppercase letter at the beginning of each conjoined word in the name. For example, `myParameter`.
- Use the plural form for parameters that represent an array of objects.
- Make variable names unambiguous, for example, `displayName`.
- Include a description for each parameter to describe its purpose.
- Do not use an excessive number of parameters in a single action.

Use Action Version History

You can use version history to revert an action to a previous version. You can revert the action state to an earlier or a later action version. You can also compare the differences between the current state of the action and a saved version of the action.

Orchestrator creates a new version history item for each action when you increase and save the action version. Subsequent changes to the action do not change the current version item. For example, when you create action version 1.0.0 and save it, the state of the action is stored in the database. If you make any changes to the action, you can save the action state in the Orchestrator client, but you cannot apply the changes to action version 1.0.0. To store the changes in the database, you must create a subsequent action version and save it. The version history is kept in the database along with the action itself.

When you delete an action, Orchestrator marks the element as deleted in the database without deleting the version history of the element from the database. This way, you can restore deleted actions. See [Restore Deleted Actions](#).

Prerequisites

Open an action for editing.

Procedure

- 1 Click the **General** tab in the action editor.

- 2 Click **Show version history**.

A version history window appears.

- 3 Select an action version and click **Diff Against Current** to compare the differences.

A window that displays the differences between the current action version and the selected action version appears.

- 4 Select an action version and click **Revert** to restore the state of the action.

Caution If you have not saved the current action version, it is deleted from the version history and you cannot revert back to the current version.

The action state is reverted to the state of the selected version.

Restore Deleted Actions

You can restore actions that have been deleted from the library.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Actions** view.
- 3 Navigate to the folder in which you want to restore a deleted action or actions.
- 4 Right-click the folder and select **Restore deleted actions**.
- 5 Select the action or actions that you want to restore and click **Restore**.

Results

The action or actions appear in the selected folder.

Creating Resource Elements

4

Workflows might require objects that you create independently of Orchestrator to be used as attributes. To use external objects as attributes in workflows, you import them into the Orchestrator server as resource elements.

Objects that workflows can use as resource elements include image files, scripts, XML templates, HTML files, and so on. Any workflows that run in the Orchestrator server can use any resource elements that you import into Orchestrator.

Importing an object into Orchestrator as a resource element lets you make changes to the object in a single location, and to propagate those changes automatically to all the workflows that use this resource element.

You can organize resource elements into folders. The maximum size for a resource element is 16MB.

This chapter includes the following topics:

- [View a Resource Element](#)
- [Import an External Object to Use as a Resource Element](#)
- [Edit the Resource Element Information](#)
- [Save a Resource Element to a File](#)
- [Update a Resource Element](#)
- [Add a Resource Element to a Workflow](#)

View a Resource Element

You can view existing resource elements in the Orchestrator client, to examine their contents and discover which workflows use this resource element.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Resources** view.
- 3 Expand the hierarchical tree viewer to navigate to a resource element.

- 4 Click a resource element to show information about it in the right pane.
- 5 Click the **Viewer** tab to display the contents of the resource element.
- 6 Right-click the resource element and select **Find Elements that Use this Element**.

Orchestrator lists all the workflows that use this resource element.

What to do next

Import and edit a resource element.

Import an External Object to Use as a Resource Element

Workflows can require objects that you create independently of Orchestrator to be used as attributes. To use external objects as attributes in workflows, you must import them to the Orchestrator server as resource elements.

Prerequisites

Verify that you have an image file, script, XML template, HTML file, or other type of object to import.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Resources** view.
- 3 Right-click a resource folder in the hierarchical list or the root and select **New folder** to create a folder in which to store the resource element.
- 4 Right-click the resource folder in which to import the resource element and select **Import resources**.
- 5 Select the resource to import and click **Open**.

Orchestrator adds the resource element to the folder you selected.

Results

You imported a resource element into the Orchestrator server.

What to do next

Edit the general information of the resource element.

Edit the Resource Element Information

After you import an object into the Orchestrator server as a resource element, you can edit the resource element's details.

Prerequisites

Verify that you have imported an image, script, XML, or HTML file, or any other type of object into Orchestrator as a resource element.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Resources** view.
- 3 Right-click the resource element and select **Edit**.
- 4 Click the **General** tab and set the resource element name, version, and description.
- 5 (Optional) To see a preview of imported image resource elements, click the **Viewer** tab.
- 6 Click **Save and close** to exit the editor.

Results

You edited the general information about the resource element.

What to do next

Save the resource element to a file to update it, or add the resource element to a workflow.

Save a Resource Element to a File

You can save a resource element to a file on your local system. Saving the resource element as a file allows you to edit it.

You cannot edit a resource element in the Orchestrator client. For example, if the resource element is an XML configuration file or a script, you must save it locally to modify it.

Prerequisites

Verify that the Orchestrator server contains a resource element that you can save to a file.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Resources** view.
- 3 Right-click the resource element and select **Save to file**.
- 4 Make the required modifications to the file.

Results

You saved a resource element to a file.

What to do next

Update the resource element in the Orchestrator server.

Update a Resource Element

If you want to update a resource element, you must export it to the file system, edit the exported file with an appropriate tool, and update the resource element by importing the edited file.

Prerequisites

Verify that you have imported an image, script, XML, or HTML file, or any other type of object into Orchestrator as a resource element.

Procedure

- 1 Modify the source file of the resource element in your local system.
- 2 From the drop-down menu in the Orchestrator client, select **Design**.
- 3 Click the **Resources** view.
- 4 Navigate through the hierarchical list to the resource element that you have updated.
- 5 Right-click the resource element and select **Update resource**.
- 6 (Optional) Click the **Viewer** tab to verify that Orchestrator has updated the resource element.

Results

You updated a resource element that the Orchestrator server contains.

Add a Resource Element to a Workflow

Resource elements are external objects that you can import to the Orchestrator server for workflows to use as attributes when they run. For example, a workflow can use an imported XML file that defines a map to convert one type of data to another, or a script that defines a function, when it runs.

Prerequisites

Verify that you have the following objects in your Orchestrator server:

- An image, script, XML, or HTML file, or any other type of object imported into Orchestrator as a resource element.
- A workflow that requires the resource element as an attribute.

Procedure

- 1 From the drop-down menu in the Orchestrator client, select **Design**.
- 2 Click the **Workflows** view.
- 3 Expand the hierarchical tree viewer to navigate to the workflow that requires the resource element as an attribute.
- 4 Right-click the workflow and select **Edit**.

- 5 On the **General** tab, in the Attributes pane, click the **Add attribute** icon (A+).
- 6 Click the attribute name and type a new name for the attribute.
- 7 Click **Type** to set the attribute type.
- 8 In the **Select a type** dialog box, type **resource** in the **Filter** box to search for an object type.

Option	Action
Define a single resource element as an attribute	Select ResourceElement from the list.
Define a folder that contains multiple resource elements as an attribute	Select ResourceElementCategory from the list.

- 9 Click **Value** and type the name of the resource element or category of resource elements in the **Filter** text box.
- 10 From the proposed list, select the resource element or a folder containing resource elements and click **Select**.
- 11 Click **Save and close** to exit the editor.

Results

You added a resource element or folder of resource elements as an attribute in a workflow.

Creating Packages

5

Packages are used for distributing content from one Orchestrator server to another. Packages can contain workflows, actions, policy templates, configurations, or resources.

When you add an element to a package, Orchestrator checks for dependencies and adds any dependent elements to the package. For example, if you add a workflow that uses actions or other workflows, Orchestrator adds those actions and workflows to the package.

When you import a package, the server compares the versions of the different elements of its contents to matching local elements. The comparison shows the differences in versions between the local and imported elements. The administrator can decide whether to import the package, or can select specific elements to import.

Packages use digital rights management to control how the receiving server can use the contents of the package. Orchestrator signs packages and encrypts the packages for data protection. Packages can track which users export and redistribute elements by using X509 certificates.

For more information about using packages, see *Using the VMware vRealize Orchestrator Client*.

■ [Create a Package](#)

You can export workflows, policy templates, actions, plug-in references, resources, and configuration elements in packages. All elements that an element in a package implements are added to the package automatically, to ensure compatibility between versions. If you do not want to add the referenced elements, you can delete them in the package editor.

Create a Package

You can export workflows, policy templates, actions, plug-in references, resources, and configuration elements in packages. All elements that an element in a package implements are added to the package automatically, to ensure compatibility between versions. If you do not want to add the referenced elements, you can delete them in the package editor.

Prerequisites

Verify that the Orchestrator server contains elements such as workflows, actions, and policy templates that you can add to a package.

Procedure

- 1 From the drop-down menu in the Orchestrator Legacy Client, select **Administer**.
- 2 Click the **Packages** view.
- 3 Right-click in the left pane and select **Add package**.
- 4 Type the name of the new package and click **Ok**.
The syntax for package names is *domain.your_company.folder.package_name*.
For example, `com.vmware.myfolder.mypackage`.
- 5 Right-click the package and select **Edit**.
The package editor opens.
- 6 On the **General** tab, add a description for the package.
- 7 On the **Workflows** tab, add workflows to the package.
 - To search for and select workflows in a selection dialog box, click **Insert Workflows (list search)**.
 - To browse and select folders of workflows from the hierarchical list, click **Insert Workflows (tree browsing)**.
- 8 On the **Policy Templates**, **Actions**, **Configurations**, **Resources**, and **Used Plug-Ins** tabs, add policy templates, actions, configuration elements, resource elements, and plug-ins to the package.
- 9 To exit the editor, click **Save and close**.

Results

You created a package and added elements to it.

Developing Plug-Ins

6

Orchestrator allows integration with management and administration solutions through its open plug-in architecture. You use the Orchestrator client to run and create plug-in workflows and access the plug-in API.

This chapter includes the following topics:

- [Overview of Plug-Ins](#)
- [Contents and Structure of a Plug-In](#)
- [Orchestrator Plug-In API Reference](#)
- [Elements of the vso.xml Plug-In Definition File](#)
- [Best Practices for Orchestrator Plug-In Development](#)

Overview of Plug-Ins

Orchestrator plug-ins must include a standard set of components and must adhere to a standard architecture. These practices help you to create plug-ins for the widest possible variety of external technologies.

- [Structure of an Orchestrator Plug-In](#)

Orchestrator plug-ins have a common structure that consists of various types of layers that implement specific functionality.

- [Exposing an External API to Orchestrator](#)

You expose an API from an external product to the Orchestrator platform by creating an Orchestrator plug-in. You can create a plug-in for any technology that exposes an API that you can map into JavaScript objects that Orchestrator can use.

- [Components of a Plug-In](#)

Plug-ins are composed of a standard set of components that expose the objects in the plugged-in technology to the Orchestrator platform.

- **Role of the vso.xml File**

You use the `vso.xml` file to map the objects, classes, methods, and attributes of the plugged-in technology to Orchestrator inventory objects, scripting types, scripting classes, scripting methods, and attributes. The `vso.xml` file also defines the configuration and start-up behavior of the plug-in.

- **Roles of the Plug-In Adapter**

The plug-in adapter is the entry point of the plug-in to the Orchestrator server. The plug-in adapter serves as the datastore for the plugged-in technology in the Orchestrator server, creates the plug-in factory, and manages events that occur in the plugged-in technology.

- **Roles of the Plug-In Factory**

The plug-in factory defines how Orchestrator finds objects in the plugged-in technology and performs operations on the objects.

- **Role of Finder Objects**

Finder objects identify and locate specific instances of managed object types in the plugged-in technology. Orchestrator can modify and interact with objects that it finds in the plugged-in technology by running workflows on the finder objects.

- **Role of Scripting Objects**

Scripting objects are JavaScript representations of objects from the plugged-in technology. Scripting objects from plug-ins appear in the Orchestrator Javascript API and you can use them in scripted elements in workflows and actions.

- **Role of Event Handlers**

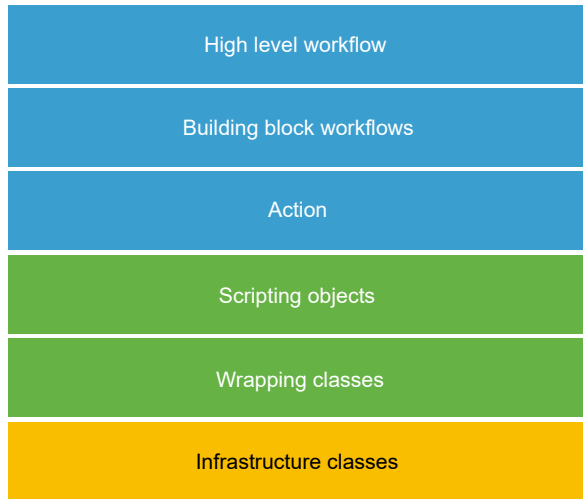
Events are changes in the states or attributes of the objects that Orchestrator finds in the plugged-in technology. Orchestrator monitors events by implementing event handlers.

Structure of an Orchestrator Plug-In

Orchestrator plug-ins have a common structure that consists of various types of layers that implement specific functionality.

The bottom three layers of a Orchestrator plug-in, which are the infrastructure classes, wrapping classes, and scripting objects, implement the connection between the plugged-in technology and Orchestrator.

The user-visible parts of a Orchestrator plug-in are the top three layers, which are actions, building blocks, and high-level workflows.

Figure 6-1. Structure of an Orchestrator Plug-In**Infrastructure classes**

A set of classes that provide the connection between the plugged-in technology and Orchestrator. The infrastructure classes include the classes to implement according to the plug-in definition, such as plug-in factory, plug-in adaptor, and so on. The infrastructure classes also include the classes that provide functionality for common tasks and objects such as helpers, caching, inventory, and so on.

Wrapping classes

A set of classes that adapt the object model of the plugged-in technology to the object model that you want to expose inside Orchestrator.

Scripting objects

JavaScript object types that provide access to the wrapping classes, methods, and attributes in the plugged-in technology. In the `vso.xml` file, you define which wrapping classes, attributes, and methods from the plugged-in technology will be exposed to Orchestrator.

Actions

A set of JavaScript functions that you can use directly in workflows and scripting tasks. Actions can take multiple input parameters and have a single return value.

Building block workflows

A set of workflows that cover all generic functionality that you want to provide with the plug-in. Typically, a building block workflow represents an operation in the user interface of the orchestrated technology. The building block workflows can be used directly or can be included inside high-level workflows.

High-level workflows

A set of workflows that cover specific functionality of the plug-in. You can provide high-level workflows to meet concrete requirements or to show complex examples of the plug-in usage.

Exposing an External API to Orchestrator

You expose an API from an external product to the Orchestrator platform by creating an Orchestrator plug-in. You can create a plug-in for any technology that exposes an API that you can map into JavaScript objects that Orchestrator can use.

Plug-ins map Java objects and methods to JavaScript objects that they add to the Orchestrator scripting API. If an external technology exposes a Java API, you can map the API directly to JavaScript for Orchestrator to use in workflows and actions.

You can create plug-ins for applications that expose an API in a language other than Java by using WSDL (Web service definition language), REST (Representational state transfer), or a messaging service to integrate the exposed API with Java objects. You then map the integrated Java objects to JavaScript for Orchestrator to use.

The plugged-in technology is independent from Orchestrator. You can create Orchestrator plug-ins for external products even if you only have access to binary code, for example in Java archives (JAR files), rather than source code.

Components of a Plug-In

Plug-ins are composed of a standard set of components that expose the objects in the plugged-in technology to the Orchestrator platform.

The main components of a plug-in are the plug-in adapter, factory, and event implementations. You map the objects and operations defined in the adapter, factory, and event implementations to Orchestrator objects in an XML definition file named `vso.xml`. The `vso.xml` file maps objects and functions from the plugged in technology to JavaScript scripting objects that appear in the Orchestrator JavaScript API. The `vso.xml` file also maps object types from the plugged-in technology to finders, that appear in the Orchestrator **Inventory** tab.

Plug-ins are composed of the following components.

Plug-In Module

The plug-in itself, as defined by a set of Java classes, a `vso.xml` file, and packages of the workflows and actions that interact with the objects that you access through the plug-in. The plug-in module is mandatory.

Plug-In Adapter

Defines the interface between the plugged-in technology and the Orchestrator server. The adapter is the entry point of the plug-in to the Orchestrator platform. The adapter creates the plug-in factory, manages the loading and unloading of the plug-in, and manages the events that occur on the objects in the plugged-in technology. The plug-in adapter is mandatory.

Plug-In Factory

Defines how Orchestrator finds objects in the plugged-in technology and performs operations on them. The adapter creates a factory for the client session that opens between Orchestrator and a plugged-in technology. The factory allows you either to share a session between all client connections or to open one session per client connection. The plug-in factory is mandatory.

Configuration

Orchestrator does not define a standard way for the plug-in to store its configuration. You can store configuration information by using Windows Registries, static configuration files, storing information in a database, or in XML files. Orchestrator plug-ins can be configured by running configuration workflows in the Orchestrator client.

Finders

Interaction rules that define how Orchestrator locates and represents the objects in the plugged-in technology. Finders retrieve objects from the set of objects that the plugged-in technology exposes to Orchestrator. You define in the `vso.xml` file the relations between objects to allow you to navigate through the network of objects. Orchestrator represents the object model of the plugged-in technology in the **Inventory** tab. Finders are mandatory if you want to expose objects in the plugged-in technology to Orchestrator.

Scripting Objects

JavaScript object types that provide access to the objects, operations, and attributes in the plugged-in technology. Scripting objects define how Orchestrator accesses the object model of the plugged-in technology through JavaScript. You map the classes and methods of the plugged-in technology to JavaScript objects in the `vso.xml` file. You can access the JavaScript objects in the Orchestrator scripting API and integrate them into Orchestrator scripted tasks, actions, and workflows. Scripting objects are mandatory if you want to add scripting types, classes, and methods to the Orchestrator JavaScript API.

Inventory

Instances of objects in the plugged-in technology that Orchestrator locates by using finders appear in the **Inventory** view in the Orchestrator client. You can perform operations on the objects in the inventory by running workflows on them. The inventory is optional. You can create a plug-in that only adds scripting types and classes to the Orchestrator JavaScript API and does not expose any instances of objects in the inventory.

Events

Changes in the state of an object in the plugged-in technology. Orchestrator can listen passively for events that occur in the plugged-in technology. Orchestrator can also actively trigger events in the plugged-in technology. Events are optional.

Role of the vso.xml File

You use the `vso.xml` file to map the objects, classes, methods, and attributes of the plugged-in technology to Orchestrator inventory objects, scripting types, scripting classes, scripting methods, and attributes. The `vso.xml` file also defines the configuration and start-up behavior of the plug-in.

The `vso.xml` file performs the following principal roles.

Start-Up and Configuration Behavior

Defines the manner in which the plug-in starts and locates any configuration implementations that the plug-in defines. Loads the plug-in adapter.

Inventory Objects

Defines the types of objects that the plug-in accesses in the plugged-in technology. The finder methods of the plug-in factory implementation locate instances of these objects and display them in the Orchestrator inventory.

Scripting Types

Adds scripting types to the Orchestrator JavaScript API to represent the different types of object in the inventory. You can use these scripting types as input parameters in workflows.

Scripting Classes

Adds classes to the Orchestrator JavaScript API that you can use in scripted elements in workflows, actions, policies, and so on.

Scripting Methods

Adds methods to the Orchestrator JavaScript API that you can use in scripted elements in workflows, actions, policies, and so on.

Scripting Attributes

Adds the attributes of the objects in the plugged-in technology to the Orchestrator JavaScript API that you can use in scripted elements in workflows, actions, policies, and so on.

Roles of the Plug-In Adapter

The plug-in adapter is the entry point of the plug-in to the Orchestrator server. The plug-in adapter serves as the datastore for the plugged-in technology in the Orchestrator server, creates the plug-in factory, and manages events that occur in the plugged-in technology.

To create a plug-in adapter, you create a Java class that implements the `IPluginAdaptor` interface.

The plug-in adapter class that you create manages the plug-in factory, events, and triggers in the plugged-in technology. The `IPluginAdaptor` interface provides methods that you use to perform these tasks.

The plug-in adapter performs the following principal roles.

Creates a factory

The most important role of the plug-in adapter is to load and unload one plug-in factory instance for every connection from Orchestrator to the plugged-in technology. The plug-in adapter class calls the `IPluginAdapter.createPluginFactory()` method to create an instance of a class that implements the `IPluginFactory` interface.

Manages events

The plug-in adapter is the interface between the Orchestrator server and the plugged-in technology. The plug-in adapter manages the events that Orchestrator performs or watches for on the objects in the plugged-in technology. The adapter manages events through event publishers. Event publishers are instances of the `IPluginEventPublisher` interface that the adapter creates by calling the `IPluginAdapter.registerEventPublisher()` method. Event publishers set triggers and gauges on objects in the plugged-in technology, to allow Orchestrator to launch defined actions if certain events occur on the object, or if the object's values pass certain thresholds. Similarly, you can define `PluginTrigger` and `PluginWatcher` instances that define events that Wait Event elements in long-running workflows await.

Sets the plug-in name

You provide a name for the plug-in in the `vso.xml` file. The plug-in adapter gets this name from the `vso.xml` file and publishes it in the Orchestrator client **Inventory** view.

Installs licenses

You can call methods to install any license files that the plugged-in technology requires in the adapter implement.

For full details of the `IPluginAdapter` interface, all of its methods, and all of the other classes of the plug-in API, see [Orchestrator Plug-In API Reference](#).

Roles of the Plug-In Factory

The plug-in factory defines how Orchestrator finds objects in the plugged-in technology and performs operations on the objects.

To create the plug-in factory, you must implement and extend the `IPluginFactory` interface from the Orchestrator plug-in API. The plug-in factory class that you create defines the finder functions that Orchestrator uses to access objects in the plugged-in technology. The factory allows the Orchestrator server to find objects by their ID, by their relation to other objects, or by searching for a query string.

The plug-in factory performs the following principal tasks.

Finds objects

You can create functions that find objects according to their name and type. You find objects by name and type by using the `IPluginFactory.find()` method.

Finds objects related to other objects

You can create functions to find objects that relate to a given object by a given relation type. You define relations in the `vso.xml` file. You can also create finders to find dependent child objects that relate to all parents by a given relation type. You implement the `IPluginFactory.findRelation()` method to find any objects that are related to a given parent object by a given relation type. You implement the `IPluginFactory.hasChildrenInRelation()` method to discover whether at least one child object exists for a parent instance.

Define queries to find objects according to your own criteria

You can create object finders that implement query rules that you define. You implement the `IPluginFactory.findAll()` method to find all objects that satisfy query rules you define when the factory calls this method. You obtain the results of the `findAll()` method in a `QueryResult` object that contains a list of all of the objects found that match the query rules you define.

For more information about the `IPluginFactory` interface, all of its methods, and all of the other classes of the plug-in API, see [Orchestrator Plug-In API Reference](#).

Role of Finder Objects

Finder objects identify and locate specific instances of managed object types in the plugged-in technology. Orchestrator can modify and interact with objects that it finds in the plugged-in technology by running workflows on the finder objects.

Every instance of a given managed object type in the plugged-in technology must have a unique identifier so that Orchestrator finder objects can find them. The plugged-in technology provides the unique identifiers for the object instances as strings. When a workflow runs, Orchestrator sets the unique identifiers of the objects that it finds as workflow attribute values. Workflows that require an object of a given type as an input parameter run on a specific instance of that type of object.

Finder objects that plug-ins add to the Orchestrator JavaScript API have the plug-in name as a prefix. For example, the `VirtualMachine` managed object type from the vCenter Server API appears in Orchestrator as the `VC:VirtualMachine` JavaScript type.

For example, Orchestrator accesses a specific `VC:VirtualMachine` instance through the vCenter Server plug-in by implementing a finder object that uses the `id` attribute of the virtual machine as its unique identifier. You can pass this object instance to workflow elements as attribute values.

An Orchestrator plug-in maps the objects from the plugged-in technology to equivalent Orchestrator finder objects in the `<finder>` elements in the `vso.xml` file. The `<finder>` elements identify the method or function from the plugged-in technology that obtains the unique identifier for a specific instance of an object. The `<finder>` elements also define relations between objects, to find objects by the manner in which they relate to other objects.

Finder objects appear in the Orchestrator **Inventory** tab under the plug-in that contains them.

Role of Scripting Objects

Scripting objects are JavaScript representations of objects from the plugged-in technology. Scripting objects from plug-ins appear in the Orchestrator JavaScript API and you can use them in scripted elements in workflows and actions.

Scripting objects from plug-ins appear in the Orchestrator JavaScript API as JavaScript modules, types, and classes. Most finder objects have a scripting object representation. The JavaScript classes can add methods and attributes to the Orchestrator JavaScript API that represent the methods and attributes from objects from the API of the plugged-in technology. The plugged-in technology provides the implementations of the objects, types, classes, attributes, and methods independently of Orchestrator. For example, the vCenter Server plug-in represents all the objects from the vCenter Server API as JavaScript objects in the Orchestrator JavaScript API, with JavaScript representations of all the classes, methods and attributes that the vCenter Server API defines. You can use the vCenter Server scripting classes and the methods and attributes they define in Orchestrator scripted functions.

For example, the `VirtualMachine` managed object type from the vCenter Server API is found by the `VC:VirtualMachine` finder and appears in the Orchestrator JavaScript API as the `VcVirtualMachine` JavaScript class. The `VcVirtualMachine` JavaScript class in the Orchestrator JavaScript API defines all of the same methods and attributes as the `VirtualMachine` managed object from the vCenter Server API.

An Orchestrator plug-in maps the objects, types, classes, attributes, and methods from the plugged-in technology to equivalent Orchestrator JavaScript objects, types, classes, attributes, and methods in the `<scripting-objects>` element in the `vso.xml` file.

Role of Event Handlers

Events are changes in the states or attributes of the objects that Orchestrator finds in the plugged-in technology. Orchestrator monitors events by implementing event handlers.

Orchestrator plug-ins allow you to monitor events in a plugged-in technology in different ways. The Orchestrator plug-in API allows you to create the following types of event handlers to monitor events in a plugged-in technology.

Listeners

Passively monitor objects in the plugged-in technology for changes in their state. The plugged-in technology or the plug-in implementation defines the events that listeners monitor. Listeners do not initiate events, but notify Orchestrator when the events occur. Listeners detect events either by polling the plugged-in technology or by receiving notifications from the plugged-in technology. When events occur, Orchestrator policies or workflows that are waiting for the event can react by starting operations in the Orchestrator server. Listener components are optional.

Policies

Monitor certain events in the plugged-in technology and start operations in the Orchestrator server if the events occur. Policies can monitor policy triggers and policy gauges. Policy triggers define an event in the plugged-in technology that, when it occurs, causes a running policy to start an operation in the Orchestrator server, for example running a workflow. Policy gauges define ranges of values for the attributes of an object in the plugged-in technology that, when exceeded, cause Orchestrator to start an operation. Policies are optional.

Workflow triggers

If a running workflow contains a Wait Event element, when it reaches that element it suspends its run and waits for an event to occur in a plugged-in technology. Workflow triggers define the events in the plugged-in technology that Waiting Event elements in workflows await. You register workflow triggers with watchers. Workflow triggers are optional.

Watchers

Watch workflow triggers for a certain event in the plugged-in technology, on behalf of a Waiting Event element in a workflow. When the event occurs, the watchers notify any workflows that are waiting for that event. Watchers are optional.

Contents and Structure of a Plug-In

Orchestrator plug-ins must contain a standard set of components and conform to a standard file structure. For a plug-in to conform to the standard file structure, it must include specific folders and files.

To create an Orchestrator plug-in, you define how Orchestrator accesses and interacts with the objects in the plugged-in technology. And, you map all of the objects and functions of the plugged-in technology to corresponding Orchestrator objects and functions in the `vso.xml` file.

The `vso.xml` file must include a reference to every type of object or operation to expose to Orchestrator. Every object that the plug-in finds in the plugged-in technology must have a unique identifier that you provide. You define the object names in the `finder` elements and in the `object` elements in the `vso.xml` file.

A plug-in can be delivered as a standard Java archive file (JAR) or a ZIP file, but in either case, the file must be renamed with a `.dar` extension.

Note You can use the Orchestrator Control Center to import a DAR file to the Orchestrator server.

■ [Defining the Application Mapping in the vso.xml File](#)

Objects that you include in the `vso.xml` file appear as scripting objects in the Orchestrator scripting API, or as finder objects in the Orchestrator **Inventory** tab.

■ [Format of the vso.xml Plug-In Definition File](#)

The `vso.xml` file defines how the Orchestrator server interacts with the plugged-in technology. You must include a reference to every type of object or operation to expose to Orchestrator in the `vso.xml` file.

■ [Naming Plug-In Objects](#)

You must provide a unique identifier for every object that the plug-in finds in the plugged-in technology. You define the object names in the `<finder>` elements and in the `<object>` elements in the `vso.xml` file.

■ [Plug-In Object Naming Conventions](#)

You must follow Java class naming conventions when you name all objects in plug-ins.

■ [File Structure of the Plug-In](#)

A plug-in must conform to a standard file structure and must include certain specific folders and files. You deliver a plug-in as a standard Java archive (JAR) or ZIP file, that you must rename with the `.dar` extension.

Defining the Application Mapping in the vso.xml File

Objects that you include in the `vso.xml` file appear as scripting objects in the Orchestrator scripting API, or as finder objects in the Orchestrator **Inventory** tab.

The `vso.xml` file provides the following information to the Orchestrator server:

- A version, name, and description for the plug-in
- References to the classes of the plugged-in technology and to the associated plug-in adapter
- Initializes the plug-in when the Orchestrator server starts
- Scripting types to represent the types of objects in the plugged-in technology
- The relationships between object types to define how the objects display in the Orchestrator Inventory
- Scripting classes that map the objects and operations in the plugged-in technology to functions and object types in the Orchestrator JavaScript API
- Enumerations to define a list of constant values that apply to all objects of a certain type
- Events that Orchestrator monitors in the plugged-in technology

The `vso.xml` file must conform to the XML schema definition of Orchestrator plug-ins. You can access the schema definition at the VMware support site.

```
http://www.vmware.com/support/orchestrator/plugin-4-1.xsd
```

For descriptions of all of the elements of the `vso.xml` file, see [Elements of the vso.xml Plug-In Definition File](#).

Format of the vso.xml Plug-In Definition File

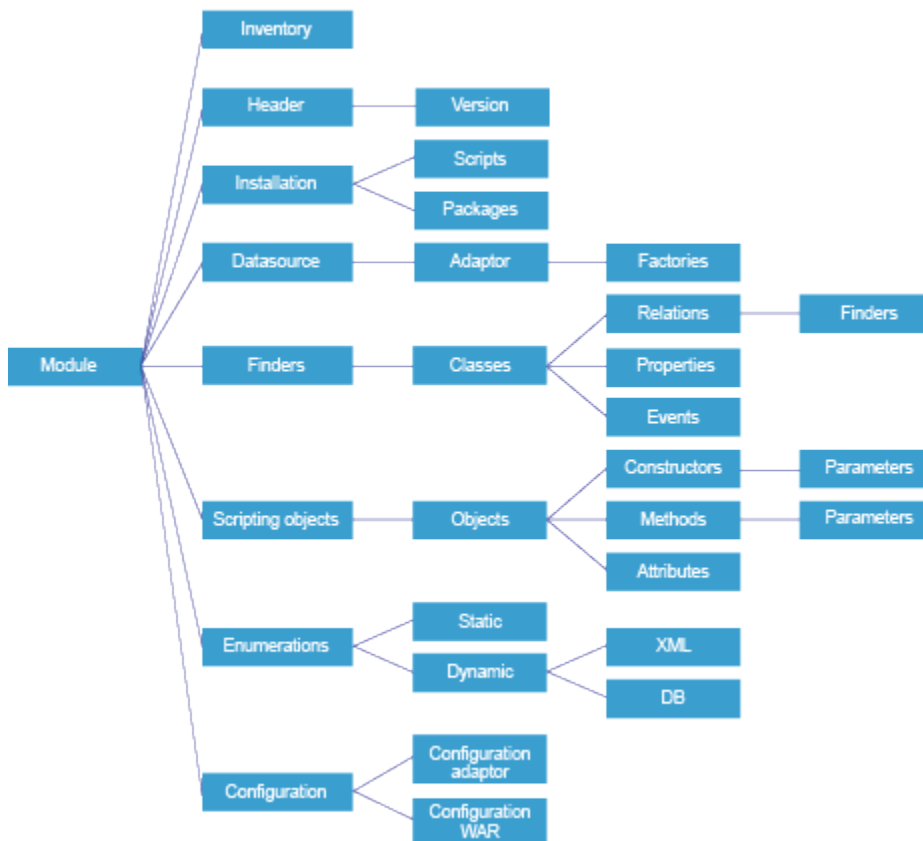
The `vso.xml` file defines how the Orchestrator server interacts with the plugged-in technology. You must include a reference to every type of object or operation to expose to Orchestrator in the `vso.xml` file.

Objects that you include in the `vso.xml` file appear as scripting objects in the Orchestrator scripting API, or as finder objects in the Orchestrator **Inventory** tab.

As part of the open architecture and standardized implementation of plug-ins, the `vso.xml` file must adhere to a standard format.

The following diagram shows the format of the `vso.xml` plug-in definition file and how the elements nest within each other.

Figure 6-2. Format of the vso.xml Plug-In Definition File



Naming Plug-In Objects

You must provide a unique identifier for every object that the plug-in finds in the plugged-in technology. You define the object names in the `<finder>` elements and in the `<object>` elements in the `vso.xml` file.

The finder operations that you define in the factory implementation find objects in the plugged-in technology. When the plug-in finds objects, you can use them in Orchestrator workflows and pass them from one workflow element to another. The unique identifiers that you provide for the objects allows them to pass between the elements in a workflow.

The Orchestrator server stores only the type and identifier of each object that it processes, and stores no information about where or how Orchestrator obtained the object. You must name objects consistently in the plug-in implementation so that you can track the objects you obtain from plug-ins.

If the Orchestrator server stops while workflows are running, when you restart the server the workflows resume at the workflow element that was running when the server stopped. The workflow uses the identifiers to retrieve objects that the element was processing when the server stopped.

Plug-In Object Naming Conventions

You must follow Java class naming conventions when you name all objects in plug-ins.

Important Because of the way in which the workflow engine performs data serialization, do not use the following string sequences in object names. Using these character sequences in object identifiers causes the workflow engine to parse workflows incorrectly, which can cause unexpected behavior when you run the workflows.

- #;#
 - #,#
 - #=#
-

Use these guidelines when you name objects in plug-ins.

- Use an initial uppercase letter for each word in the name.
- Do not use spaces to separate words.
- For letters, only use the standard characters A to Z and a to z.
- Do not use special characters, such as accents.
- Do not use a number as the first character of a name.
- Where possible, use fewer than 10 characters.

[Table 6-1. Plug-In Object Naming Rules](#) shows rules that apply to individual object types.

Table 6-1. Plug-In Object Naming Rules

Object Type	Naming Rules
Plug-In	<ul style="list-style-type: none"> ■ Defined in the <code><module></code> element in the <code>vso.xml</code> file. ■ Must adhere to Java class naming conventions. ■ Must be unique. You cannot run two plug-ins with the same name in an Orchestrator server.
Finder object	<ul style="list-style-type: none"> ■ Defined in the <code><finder></code> elements in the <code>vso.xml</code> file. ■ Must adhere to Java class naming conventions. ■ Must be unique in the plug-in. <p>Orchestrator adds the plug-in name and a colon to the finder object names in the finder object types in the Orchestrator scripting API. For example, the <code>VirtualMachine</code> object type from the vCenter Server plug-in appears in the Orchestrator scripting API as <code>VC:VirtualMachine</code>.</p>
Scripting object	<ul style="list-style-type: none"> ■ Defined in the <code><scripting-object></code> elements in the <code>vso.xml</code> file. ■ Must adhere to Java class naming conventions. ■ Must be unique in the Orchestrator server. ■ To avoid confusing scripting objects with finder objects of the same name or with scripting objects from other plug-ins, always prefix the scripting object name with the name of the plug-in, but do not add a colon. For example, the <code>VirtualMachine</code> class from the vCenter Server plug-in appears in the Orchestrator scripting API as the <code>VcVirtualMachine</code> class.

File Structure of the Plug-In

A plug-in must conform to a standard file structure and must include certain specific folders and files. You deliver a plug-in as a standard Java archive (JAR) or ZIP file, that you must rename with the `.dar` extension.

The contents of the DAR archive must use the following folder structure and naming conventions.

Table 6-2. Structure of the DAR Archive

Folders	Description
<i>plug-in_name</i> \VSO-INF\	Contains the <code>vso.xml</code> file that defines the mapping of the objects in the plugged-in technology to Orchestrator objects. The VSO-INF folder and the <code>vso.xml</code> file are mandatory.
<i>plug-in_name</i> \lib\	Contains the JAR files that contain the binaries of the plugged-in technology. Also contains JAR files that contain the implementations of the adapter, factory, notification handlers, and other interfaces in the plug-in. The lib folder and JAR files are mandatory.
<i>plug-in_name</i> \resources\	Contains resource files that the plug-in requires. The resources folder can include the following types of element: <ul style="list-style-type: none"> ■ Image files, to represent the objects of the plug-in in the Orchestrator Inventory tab. ■ Scripts, to define initialization behavior when the plug-in starts. ■ Orchestrator packages, that can contain custom workflows, actions, and other resources that interact with the objects that you access by using the plug-in. You can organize resources in subfolders. For example, <code>resources\images\</code> , <code>resources\scripts\</code> , or <code>resources\packages\</code> . The resources folder is optional.

You use the Orchestrator Control Center to import a DAR file to the Orchestrator server.

Orchestrator Plug-In API Reference

The Orchestrator plug-in API defines Java interfaces and classes to implement and extend when you develop the `IPluginAdaptor` and `IPluginFactory` implementations to create a plug-in.

All classes are contained in the `ch.dunes.vso.sdk.api` package, unless stated otherwise.

IAop Interface

The IAop interface provides methods to obtain and set properties on objects in the plugged-in technology.

```
public interface IAop
```

The IAop interface defines the following methods:

Method	Returns	Description
<code>get(java.lang.String propertyName, java.lang.Object object, java.lang.Object sdkObject)</code>	<code>java.lang.Object</code>	Obtains a property from a given object in the plug-in.
<code>set(java.lang.String propertyName, java.lang.String propertyValue, java.lang.Object object)</code>	<code>Void</code>	Sets a property on a given object in the plug-in.

IDynamicFinder Interface

The IDynamicFinder interface returns the ID and properties of a finder programmatically, instead of defining the ID and properties in the `vso.xml` file.

The IDynamicFinder Interface defines the following methods.

Method	Returns	Description
<code>getIdAccessor(java.lang.String type)</code>	<code>java.lang.String</code>	Provides an OGNL expression to obtain an object ID programmatically.
<code>getProperties(java.lang.String type)</code>	<code>java.util.List<SDKFinderProperty></code>	Provides a list of object properties programmatically.

IPluginAdaptor Interface

You implement the IPluginAdaptor interface to manage plug-in factories, events and watchers. The IPluginAdaptor interface defines an adapter between a plug-in and the Orchestrator server.

IPluginAdaptor instances are responsible for session management. The IPluginAdaptor Interface defines the following methods.

Method	Returns	Description
<code>addWatcher(PluginWatcher watcher)</code>	<code>Void</code>	Adds a watcher to monitor for a specific event
<code>createPluginFactory(java.lang.String sessionId, java.lang.String username, java.lang.String password, IPluginNotificationHandler notificationHandler)</code>	<code>IPluginFactory</code>	<p>Creates an IPluginFactory instance. The Orchestrator server uses the factory to obtain objects from the plugged-in technology by their ID, by their relation to other objects, and so on.</p> <p>The session ID allows you to identify a running session. For example, a user could log into two different Orchestrator clients and run two sessions simultaneously.</p> <p>Similarly, starting a workflow creates a session that is independent from the client in which the workflow started. A workflow continues to run even if you close the Orchestrator client.</p>

Method	Returns	Description
<code>installLicenses(PluginLicense[] licenses)</code>	Void	Installs the license information for standard plug-ins that VMware provides
<code>registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)</code>	Void	Sets triggers and gauges on an element in the inventory
<code>removeWatcher(java.lang.String watcherId)</code>	Void	Removes a watcher
<code>setPluginName(java.lang.String pluginName)</code>	Void	Gets the plug-in name from the <code>vso.xml</code> file
<code>setPluginPublisher(IPluginPublisher pluginPublisher)</code>	Void	Sets the publisher of the plug-in
<code>uninstallPluginFactory(IPluginFactory plugin)</code>	Void	Uninstalls a plug-in factory.
<code>unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)</code>	Void	Removes triggers and gauges from an element in the inventory

IPluginEventPublisher Interface

The `IPluginEventPublisher` interface publishes gauges and triggers on an event notification bus for Orchestrator policies to monitor.

You can create `IPluginEventPublisher` instances directly in the plug-in adaptor implementation or you can create them in separate event generator classes.

You can implement the `IPluginEventPublisher` interface to publish events in the plugged-in technology to the Orchestrator policy engine. You create methods to set policy triggers and gauges on objects in the plugged-in technology and event listeners to listen for events on those objects.

Policies can implement either gauges or triggers to monitor objects in the plugged-in technology. Policy gauges monitor the attributes of objects and push an event in the Orchestrator server if the values of the objects exceed certain limits. Policy triggers monitor objects and push an event in the Orchestrator server if a defined event occurs on the object. You register policy gauges and triggers with `IPluginEventPublisher` instances so that Orchestrator policies can monitor them.

The `IPluginEventPublisher` Interface defines the following methods.

Type	Returns	Description
<code>pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)</code>	Void	Publish a gauge for policies to monitor. Takes the following parameters: <ul style="list-style-type: none"> ■ <code>type</code>: Type of the object to monitor. ■ <code>id</code>: Identifier of the object to monitor. ■ <code>gaugeName</code>: Name for this gauge. ■ <code>deviceName</code>: Name for the type of attribute that the gauge monitors. ■ <code>gaugeValue</code>: Value for which the gauge monitors the object.
<code>pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)</code>	Void	Publish a trigger for policies to monitor. Takes the following parameters: <ul style="list-style-type: none"> ■ <code>type</code>: Type of the object to monitor. ■ <code>id</code>: Identifier of the object to monitor. ■ <code>triggerName</code>: Name for this trigger. ■ <code>additionalProperties</code>: Any additional properties for the trigger to monitor.

IPluginFactory Interface

The `IPluginAdaptor` returns `IPluginFactory` instances. `IPluginFactory` instances run commands in the plugged-in application, and finds objects upon which to perform Orchestrator operations.

The `IPluginFactory` interface defines the following field:

```
static final java.lang.String RELATION_CHILDREN
```

The `IPluginFactory` interface defines the following methods.

Method	Returns	Description
<code>executePluginCommand(java.lang.String cmd)</code>	Void	Use the plug-in to run a command. VMware recommends that you do not use this method.
<code>find(java.lang.String type, java.lang.String id)</code>	<code>java.lang.Object</code>	Use the plug-in to find an object. Identify the object by its ID and type.
<code>findAll(java.lang.String type, java.lang.String query)</code>	<code>QueryResult</code>	Use the plug-in to find objects of a certain type and that match a query string. You define the syntax of the query in the <code>IPluginFactory</code> implementation of the plug-in. If you do not define query syntax, <code>findAll()</code> returns all objects of the specified type.

Method	Returns	Description
<code>findRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)</code>	<code>java.util.List</code>	Determines whether an object has children.
<code>hasChildrenInRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)</code>	<code>HasChildrenResult</code>	Finds all children related to a given parent by a certain relation.
<code>invalidate(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	Invalidate objects by type and ID.
<code>void invalidateAll()</code>	<code>Void</code>	Invalidate all objects in the cache.

IPluginNotificationHandler Interface

The `IPluginNotificationHandler` defines methods to notify Orchestrator of different types of event that occur on the objects Orchestrator accesses through the plug-in.

The `IPluginNotificationHandler` Interface defines the following methods.

Method	Returns	Description
<code>getSessionID()</code>	<code>java.lang.String</code>	Returns the current session ID
<code>notifyElementDeleted(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	Notifies the system that an object with the given type and ID has been deleted
<code>notifyElementInvalidate(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	Notifies the system that an object's relations have changed. You can use the <code>notifyElementInvalidate()</code> method to notify Orchestrator of all changes in relations between objects, not only for relation changes that invalidate an object. For example, adding a child object to a parent represents a change in the relation between the two objects.
<code>notifyElementUpdated(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	Notifies the system that an object's attributes have been modified
<code>notifyMessage(ch.dunes.vso.sdk.api.ErrorLevel severity, java.lang.String type, java.lang.String id, java.lang.String message)</code>	<code>Void</code>	Publishes an error message related to the current module

IPluginPublisher Interface

The `IPluginPublisher` interface publishes a watcher event on an event notification bus for long-running workflow Wait Event elements to monitor.

When a workflow trigger starts an event in the plugged-in technology, a plug-in watcher that watches that trigger and that is registered with an `IPluginPublisher` instance notifies any waiting workflows that the event has occurred.

The `IPluginPublisher` Interface defines the following method.

Type	Value	Description
<code>pushWatcherEvent(java.lang.String id, java.util.Properties properties)</code>	<code>Void</code>	Publish a watcher event on event notification bus

WebConfigurationAdaptor Interface

The `WebConfigurationAdaptor` interface implements `IConfigurationAdaptor` and defines methods to locate and install a Web application in the configuration tab for a plug-in.

Note The `WebConfigurationAdaptor` interface is deprecated since Orchestrator 4.1. To add a Web application to the configuration, implement `IConfigurationAdaptor` and use the `configuration-war` attribute in the `vso.xml` file to identify the Web application.

The `WebConfigurationAdaptor` interface defines the following methods.

Method	Returns	Description
<code>getWebAppContext()</code>	<code>String</code>	Locates the WAR file of the Web application for the configuration tab. Provide the name and path to the WAR file from the <code>/webapps</code> directory in the DAR file as a string.
<code>setWebConfiguration(boolean webConfiguration)</code>	<code>Boolean</code>	Determine whether the contents of the configuration tab are defined by a Web application.

PluginTrigger Class

The `PluginTrigger` class creates a trigger module that obtains information about objects and events to monitor in the plugged-in technology, on behalf of a Wait Event element in a workflow.

The `PluginTrigger` class defines methods to obtain or set the type and name of the object to monitor, the nature of the event, and a timeout period.

You create implementations of the `PluginTrigger` class exclusively for use by Wait Event elements in workflows. You define policy triggers for Orchestrator policies in classes that define events and implement the `IPluginEventPublisher.pushTrigger()` method.

```
public class PluginTrigger
extends java.lang.Object
implements java.io.Serializable
```

The `PluginTrigger` class defines the following methods:

Method	Returns	Description
getModuleName()	java.lang.String	Obtains the name of the trigger module.
getProperties()	java.util.Properties	Obtains a list of properties for the trigger.
getSdkId()	java.lang.String	Obtains the ID of the object to monitor in the plugged-in technology.
getSdkType()	java.lang.String	Obtains the type of the object to monitor in the plugged-in technology.
getTimeout()	Long	Obtains the trigger timeout period.
setModuleName(java.lang.String moduleName)	Void	Sets the name of the trigger module.
setProperties(java.util.Properties properties)	Void	Sets a list of properties for the trigger.
setSdkId(java.lang.String sdkId)	Void	Sets the ID of the object to monitor in the plugged-in technology.
setSdkType(java.lang.String sdkType)	Void	Sets the type of the object to monitor in the plugged-in technology.
setTimeout(long timeout)	Void	Sets a timeout period in seconds. A negative value deactivates the timeout.

Constructors

- PluginTrigger()
- PluginTrigger(java.lang.String moduleName, long timeout, java.lang.String sdkType, java.lang.String sdkId)

PluginWatcher Class

The PluginWatcher class watches a trigger module for a defined event in the plugged-in technology on behalf of a long-running workflow Wait Event element.

The PluginWatcher class defines a constructor that you can use to create plug-in watcher instances. The PluginWatcher class defines methods to obtain or set the name of the workflow trigger to watch and a timeout period.

```
public class PluginWatcher
extends java.lang.Object
implements java.io.Serializable
```

The PluginWatcher class defines the following methods:

Method	Returns	Description
getId()	java.lang.String	Obtains the ID of the trigger
getModuleName()	java.lang.String	Obtains the trigger module name

Method	Returns	Description
getTimeoutDate()	Long	Obtains the trigger timeout date
getTrigger()	Void	Obtains a trigger
setId(java.lang.String id)	Void	Sets the ID of the trigger
setTimeoutDate()	Void	Sets the trigger timeout date

Constructor

PluginWatcher(PluginTrigger trigger)

QueryResult Class

The QueryResult class contains the results of a find query made on the objects Orchestrator accesses through the plug-in.

```
public class QueryResult
extends java.lang.Object
implements java.io.Serializable
```

The totalCount value can be greater than the number of elements the QueryResult returns, if the total number of results found exceeds the number of results the query returns. The number of results the query returns is defined in the query syntax in the vso.xml file.

The QueryResult class defines the following methods:

Method	Returns	Description
addElement(java.lang.Object element)	Void	Adds an element to the QueryResult
addElements(java.util.List elements)	Void	Adds a list of elements to the QueryResult
getElements()	java.util.List	Obtains elements from the plugged in application
getTotalCount()	Long	Obtains a count of all the elements available in the plugged in technology
isPartialResult()	Boolean	Determines whether the result obtained is complete
removeElement(java.lang.Object element)	Void	Removes an element from the plugged in technology
setElements(java.util.List elements)	Void	Sets elements in the plugged in technology
setTotalCount(long totalCount)	Void	Sets the total number of elements available in the plugged in technology

Constructors

- QueryResult()
- QueryResult(java.util.List ret)

■ QueryResult(java.util.List elements, long totalCount)

SDKFinderProperty Class

The SDKFinderProperty class defines methods to obtain and set properties in the objects found in the plugged in technology by the Orchestrator finder objects. The IDynamicFinder.getProperties method returns SDKFinderProperty objects.

```
public class SDKFinderProperty
extends java.lang.Object
```

The SDKFinderProperty class defines the following methods:

Method	Returns	Description
getAttributeName()	java.lang.String	Obtains an object attribute name
getBeanProperty()	java.lang.String	Obtains properties from a Java bean
getDescription()	java.lang.String	Obtains an object description
getDisplayName()	java.lang.String	Obtains an object display name
getPossibleResultType()	java.lang.String	Obtains the possible types of result the finder returns
getPropertyAccessor()	java.lang.String	Obtains an object property accessor
getPropertyAccessorTree()	java.lang.Object	Obtains an object property accessor tree
isHidden()	Boolean	Shows or hides the object
isShowInColumn()	Boolean	Shows or hides the object in the database column
isShowInDescription()	Boolean	Shows or hides the object description
setAttributeName(java.lang.String attributeName)	Void	Sets an object attribute name
setBeanProperty(java.lang.String beanProperty)	Void	Sets properties in a Java bean
setDescription(java.lang.String description)	Void	Sets an object description
setDisplayName(java.lang.String displayName)	Void	Sets an object display name
setHidden(boolean hidden)	Void	Show or hide the object
setPossibleResultType(java.lang.String possibleResultType)	Void	Sets the possible types of result the finder returns
setPropertyAccessor(java.lang.String propertyAccessor)	Void	Sets an object property accessor
setPropertyAccessorTree(java.lang.Object propertyAccessorTree)	Void	Sets an object property accessortree

Method	Returns	Description
<code>setShowInColumn(boolean showInTable)</code>	Void	Show or hide the object in the database column
<code>setShowInDescription(boolean showInDescription)</code>	Void	Show or hide the object description

Constructor

`SDKFinderProperty(java.lang.String attributeName, java.lang.String displayName, java.lang.String beanProperty, java.lang.String propertyAccessor)`

PluginExecutionException Class

The `PluginExecutionException` class returns an error message if the plug-in encounters an exception when it runs an operation.

```
public class PluginExecutionException
extends java.lang.Exception
implements java.io.Serializable
```

The `PluginExecutionException` class inherits the following methods from class `java.lang.Throwable`:

`fillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`, `printStackTrace`, `printStackTrace`, `setStackTrace`, `toString`
`fillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`

Constructor

`PluginExecutionException(java.lang.String message)`

PluginOperationException Class

The `PluginOperationException` class handles errors encountered during a plug-in operation.

```
public class PluginOperationException
extends java.lang.RuntimeException
implements java.io.Serializable
```

The `PluginOperationException` class inherits the following methods from class `java.lang.Throwable`:

`fillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`, `printStackTrace`, `printStackTrace`, `setStackTrace`, `toString`

Constructor

`PluginOperationException(java.lang.String message)`

HasChildrenResult Enumeration

The HasChildrenResult Enumeration declares whether a given parent has children. The `IPluginFactory.hasChildrenInRelation` method returns HasChildrenResult objects.

```
public enum HasChildrenResult
extends java.lang.Enum<HasChildrenResult>
implements java.io.Serializable
```

The HasChildrenResult enumeration defines the following constants:

- `public static final HasChildrenResult Yes`
- `public static final HasChildrenResult No`
- `public static final HasChildrenResult Unknown`

The HasChildrenResult enumeration defines the following methods:

Method	Returns	Description
<code>getValue()</code>	<code>int</code>	Returns one of the following values: <div> 1 Parent has children </div> <div> -1 Parent has no children </div> <div> 0 Unknown, or invalid parameter </div>
<code>valueOf(java.lang.String name)</code>	<code>static HasChildrenResult</code>	Returns an enumeration constant of this type with the specified name. The String must match exactly an identifier used to declare an enumeration constant of this type. Do not use whitespace characters in the enumeration name.
<code>values()</code>	<code>static HasChildrenResult[]</code>	Returns an array containing the constants of this enumeration type, in the order they are declared. This method can iterate over constants as follows: <div> <pre>for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);</pre> </div>

The HasChildrenResult enumeration inherits the following methods from class `java.lang.Enum`:

`clone`, `compareTo`, `equals`, `finalize`, `getDeclaringClass`, `hashCode`, `name`, `ordinal`, `toString`, `valueOf`

ScriptingAttribute Annotation Type

The `ScriptingAttribute` annotation type annotates an attribute from an object in the plugged in technology for use as a property in scripting.

```
@Retention(value=RUNTIME)
@Target(value={METHOD, FIELD})
public @interface ScriptingAttribute
```

The `ScriptingAttribute` annotation type has the following value:

```
public abstract java.lang.String value
```

ScriptingFunction Annotation Type

The `ScriptingFunction` annotation type annotates a method for use as a property in scripting.

```
@Retention(value=RUNTIME)
@Target(value={METHOD, CONSTRUCTOR})
public @interface ScriptingFunction
```

The `ScriptingFunction` annotation type has the following value:

```
public abstract java.lang.String value
```

ScriptingParameter Annotation Type

The `ScriptingParameter` annotation type annotates a parameter for use as a property in scripting.

```
@Retention(value=RUNTIME)
@Target(value=PARAMETER)
public @interface ScriptingParameter
```

The `ScriptingParameter` annotation type has the following value:

```
public abstract java.lang.String value
```

Elements of the vso.xml Plug-In Definition File

The `vso.xml` file contains a set of standard elements. Some of the elements are mandatory while others are optional. Each element has attributes that define values for the objects and operations you map to Orchestrator objects and operations.

In addition, elements can have zero or more child elements. A child element further defines the parent element. The same child element can appear in multiple parent elements. For example, the `description` element has no child elements, but appears as a child element for many parent elements: `module`, `example`, `trigger`, `gauge`, `finder`, `constructor`, `method`, `object`, and `enumeration`.

Each element definition that follows lists its attributes, parents and children.

module Element

A module describes a set of plug-in objects to make available to Orchestrator.

The module contains information about how data from the plugged-in technology maps to Java classes, versioning, how to deploy the module, and how the plug-in appears in the Orchestrator inventory.

The `<module>` element is optional. The `<module>` element has the following attributes:

Attributes	Value	Description
name	String	Defines the type of all the <code><finder></code> elements in the plug-in. Mandatory attribute.
version	Number	The plug-in version number, for use when reloading packages in a new version of the plug-in. Mandatory attribute.
build-number	Number	The plug-in build number, for use when reloading packages in a new version of the plug-in. Mandatory attribute.
image	Image file	The icon to display in the Orchestrator Inventory. Mandatory attribute.
display-name	String	The name that appears in the Orchestrator Inventory. Optional attribute.
interface-mapping-allowed	true or false	VMware strongly discourages interface mapping. Optional attribute.

Table 6-3. Element Hierarchy

Parent Element	Child Elements
None	<ul style="list-style-type: none"> ■ <code><description></code> ■ <code><installation></code> ■ <code><configuration></code> ■ <code><finder-datasources></code> ■ <code><inventory></code> ■ <code><finders></code> ■ <code><scripting-objects></code> ■ <code><enumerations></code>

description Element

The `<description>` elements provide descriptions of the elements of the plug-in that appear in the API Explorer documentation.

You add the text that appears in the API Explorer documentation between the `<description>` and `</description>` tags.

The `<description>` element is optional. The `<description>` element has no attributes.

Table 6-4. Element Hierarchy

Parent Elements	Child Elements
<ul style="list-style-type: none"> ■ <code><module></code> ■ <code><example></code> ■ <code><trigger></code> ■ <code><gauge></code> ■ <code><finder></code> ■ <code><constructor></code> ■ <code><method></code> ■ <code><object></code> ■ <code><enumeration></code> 	None

deprecated Element

The `<deprecated>` element marks objects and methods that are deprecated in the API Explorer documentation.

You add the text that appears in the API Explorer documentation between the `<deprecated>` and `</deprecated>` tags.

The `<deprecated>` element is optional. The `<deprecated>` element has no attributes.

Table 6-5. Element Hierarchy

Parent Elements	Child Elements
<ul style="list-style-type: none"> ■ <code><method></code> ■ <code><object></code> 	None

url Element

The `<url>` element provides a URL that points to external documentation about an object or enumeration.

You provide the URL between the `<url>` and `</url>` tags.

The `<url>` element is optional. The `<url>` element has no attributes.

Table 6-6. Element Hierarchy

Parent Elements	Child Elements
<ul style="list-style-type: none"> ■ <code><enumeration></code> ■ <code><object></code> 	None

installation Element

The `<installation>` element allows you to install a package or run a script when the server starts.

The `<installation>` element is optional. The `<installation>` element has the following attributes:

Attributes	Value	Description
mode	always, never, or version	<p>Setting the mode value results in the following behavior when the Orchestrator server starts:</p> <ul style="list-style-type: none"> ■ The action <i>always</i> runs ■ The action <i>never</i> runs ■ The action runs when the server detects a newer version of the plug-in <p>Mandatory attribute.</p>

Table 6-7. Element Hierarchy

Parent Element	Child Element
<module>	<action>

action Element

The <action> element specifies the action that runs when the Orchestrator server starts.

The <action> element attributes provide the path to the Orchestrator package or script that defines the plug-in's behavior when it starts.

The <action> element is optional. A plug-in can have an unlimited number of <action> elements. The <action> element has the following attributes.

Attributes	Value	Description
resource	String	<p>The path to the Java package or script from the root of the dar file.</p> <p>Mandatory attribute.</p>
type	install-package OR execute-script	<p>Either installs the specified Orchestrator package in the Orchestrator server, or runs the specified script. Mandatory attribute.</p>

Table 6-8. Element Hierarchy

Parent Element	Child Elements
<installation>	None

finder-datasources Element

The <finder-datasources> element is the container for the <finder-datasource> elements.

The <finder-datasources> element is optional. The <finder-datasources> element has no attributes.

Table 6-9. Element Hierarchy

Parent Element	Child Elements
<module>	<finder-datasource>

finder-datasource Element

The <finder-datasource> element points to the Java class file of the IPluginAdaptor implementation that you create for the plug-in.

You set how Orchestrator accesses the objects of the plugged-in technology in the <finder-datasource> element. The <finder-datasource> element identifies the Java class of the plug-in adapter that you create. The plug-in adapter class instantiates the plug-in factory that you create. The plug-in factory defines the methods that find objects in the plugged-in technology. You can set timeouts in the <finder-datasource> element for the finder method calls that the factory performs. Different timeouts apply to the different finder methods from the IPluginFactory interface.

The <finder-datasource> element is optional. A plug-in can have an unlimited number of <finder-datasources> elements. The <finder-datasource> element has the following attributes.

Attributes	Value	Description
name	String	Identifies the data source in the <finder> element datasource attributes. Equivalent to an XML id. Mandatory attribute.
adaptor-class	Java class	Points to the IPluginAdaptor implementation you define to create the plug-in adapter, for example, com.vmware.plugins.sample.Adaptor. Mandatory attribute.
concurrent-call	true (default) or false	Allows multiple users to access the adapter at the same time. You must set concurrent-call to false if the plug-in does not support concurrent calls. Optional attribute.
invoker-mode	direct (default) or timeout	Sets a timeout on the finder function. If set to direct, calls to finder functions never time out. If set to timeout, the Orchestrator server applies the timeout period that corresponds to the finder method. Optional attribute.
anonymous-login-mode	never (default) or always	Passes or does not pass the user's username and password to the plug-in. Optional attribute.
timeout-fetch-relation	Number; default 30 seconds	Applies to calls from findRelation(). Optional attribute.
timeout-find-all	Number; default 60 seconds	Applies to calls from findAll(). Optional attribute.

Attributes	Value	Description
timeout-find	Number; default 60 seconds	Applies to calls from find(). Optional attribute.
timeout-has-children-in-relation	Number; default 2 seconds	Applies to calls from findChildrenInRelation(). Optional attribute.
timeout-execute-plugin-command	Number; default 30 seconds	Applies to calls from executePluginCommand(). Optional attribute.

Table 6-10. Element Hierarchy

Parent Element	Child Elements
<finder-datasources>	None

inventory Element

The <inventory> element defines the root of the hierarchical list for the plug-in that appears in the Orchestrator client **Inventory** view and object selection dialog boxes.

The <inventory> element does not represent an object in the plugged-in application, but rather represents the plug-in itself as an object in the Orchestrator scripting API.

The <inventory> element is optional. The <inventory> element has the following attribute.

Attributes	Value	Description
type	An Orchestrator object type	The type of the <finder> element that represents the root of the hierarchy of objects. Mandatory attribute.

Table 6-11. Element Hierarchy

Parent Element	Child Elements
<module>	None

finders Element

The <finders> element is the container for all the <finder> elements.

The <finders> element is optional. The <finders> element has no attributes.

Table 6-12. Element Hierarchy

Parent Element	Child Element
<module>	<finder>

finder Element

The `<finder>` element represents in the Orchestrator client a type of object found through the plug-in.

The `<finder>` element identifies the Java class that defines the object the object finder represents. The `<finder>` element defines how the object appears in the Orchestrator client interface. It also identifies the scripting object that the Orchestrator scripting API defines to represent this object.

Finders act as an interface between object formats used by different types of plugged-in technologies.

The `<finder>` element is optional. A plug-in can have an unlimited number of `<finder>` elements. The `<finder>` element defines the following attributes:

Attributes	Value	Description
type	An Orchestrator object type	Type of object represented by the finder. Mandatory attribute.
datasource	<code><finder-datasource name></code> attribute	Identifies the Java class that defines the object by using the datasource refid. Mandatory attribute.
dynamic-finder	Java method	Defines a custom finder method you implement in an <code>IDynamicFinder</code> instance, to return the ID and properties of a finder programmatically, instead defining it in the <code>vso.xml</code> file. Optional attribute.
hidden	true or false (default)	If true, hides the finder in the Orchestrator client. Optional attribute.
image	Path to a graphic file	A 16x16 icon to represent the finder in hierarchical lists in the Orchestrator client. Optional attribute.
java-class	Name of a Java class	The Java class that defines the object the finder finds and maps to a scripting object. Optional attribute.
script-object	<code><scripting-object type></code> attribute	The <code><scripting-object></code> type, if any, to which to map this finder. Optional attribute.

Table 6-13. Element Hierarchy

Parent Element	Child Elements
<finders>	<ul style="list-style-type: none"> ■ <id> ■ <description> ■ <properties> ■ <default-sorting> ■ <inventory-children> ■ <relations> ■ <inventory-tabs> ■ <events>

properties Element

The <properties> element is the container for <finder><property> elements.

The <properties> element is optional. The <properties> element has no attributes.

Table 6-14. Element Hierarchy

Parent Element	Child Element
<finder>	<property>

property Element

The <property> element maps the found object's properties to Java properties or method calls.

You can call on the methods of the SDKFinderProperty class when you implement the plug-in factory to obtain properties for the plug-in factory implementation to process.

You can show or hide object properties in the views in the Orchestrator client. You can also use enumerations to define object properties.

The <property> element is optional. A plug-in can have an unlimited number of <property> elements. The <property> element has the following attributes.

Attributes	Value	Description
name	Finder name	The name the FinderResult uses to store the element. Mandatory attribute.
display-name	Finder name	The displayed property name. Optional attribute.
bean-property	Property name	<p>You use the bean-property attribute to identify a property to obtain using get and set operations. If you identify a property named MyProperty, the plug-in defines getMyProperty and setMyProperty operations.</p> <p>You set one or the other of bean-property or property-accessor, but not both. Optional attribute.</p>

Attributes	Value	Description
property-accessor	The method that obtains a property value from an object	The property-accessor attribute allows you to define an OGNL expression to validate an object's properties. You set one or the other of bean-property or property-accessor, but not both. Optional attribute.
show-in-column	true (default) or false	If true, this property shows in the Orchestrator client results table. Optional attribute.
show-in-description	true (default) or false	If true, this property shows in the object description. Optional attribute.
hidden	true or false (default)	If true, this property is hidden in all cases. Optional attribute.
linked-enumeration	Enumeration name	Links a finder property to an enumeration. Optional attribute.

Table 6-15. Element Hierarchy

Parent Element	Child Elements
<properties>	Child Elements

relations Element

The <relations> element is the container for <finder><relation> elements.

The <relations> element is optional. The <relations> element has no attributes.

Table 6-16. Element Hierarchy

Parent Element	Child Element
<finder>	<relation>

relation Element

The <relation> element defines how objects relate to other objects.

You define the relation name in the <relation> element.

The <relation> element is optional. A plug-in can have an unlimited number of <relation> elements. The <relation> element has the following attributes.

Attributes	Value	Description
name	Relation name	A name for this relation. Mandatory attribute.
type	Orchestrator object type	The type of the object that relates to another object by this relation. Mandatory attribute.
cardinality	to-one or to-many	Defines the relation between the objects as one-to-one or one-to-many. Optional attribute.

Table 6-17. Element Hierarchy

Parent Element	Child Elements
<relations>	None

id Element

The <id> element defines a method to obtain the unique ID of the object that the finder identifies.

The <id> element is optional. The <id> element has the following attributes.

Attributes	Value	Description
accessor	Method name	The accessor attribute allows you to define an OGNL expression to validate an object's properties. Mandatory attribute.

Table 6-18. Element Hierarchy

Parent Element	Child Elements
<finder>	None

inventory-children Element

The <inventory-children> element defines the hierarchy of the lists that show the objects in the Orchestrator client **Inventory** view and object selection boxes.

The <inventory-children> element is optional. The <inventory-children> element has no attributes.

Table 6-19. Element Hierarchy

Parent Element	Child Element
<finder>	<relation-link>

relation-link Element

The <relation-link> element defines the hierarchies between parent and child objects in the **Inventory** tab.

The `<relation-link>` element is optional. A plug-in can have an unlimited number of `<relation-link>` elements. The `<relation-link>` element has the following attribute.

Type	Value	Description
name	Relation name	A refid to a relation name. Mandatory attribute.

Table 6-20. Element Hierarchy

Parent Element	Child Elements
<code><inventory-children></code>	None

events Element

The `<events>` element is the container for the `<trigger>` and `<gauge>` elements.

The `<events>` element can contain an unlimited number of triggers or gauges.

The `<events>` element is optional. The `<events>` element has no attributes.

Table 6-21. Element Hierarchy

Parent Element	Child Elements
<code><finder></code>	<ul style="list-style-type: none"> ■ <code><trigger></code> ■ <code><gauge></code>

trigger Element

The `<trigger>` element declares the triggers you can use for this finder. You must implement the `registerEventPublisher()` and `unregisterEventPublisher()` methods of `IPluginAdaptor` to set triggers.

The `<trigger>` element is optional. The `<trigger>` element has the following attribute.

Type	Value	Description
name	Trigger name	A name for this trigger. Mandatory attribute.

Table 6-22. Element Hierarchy

Parent Element	Child Elements
<code><events></code>	<ul style="list-style-type: none"> ■ <code><description></code> ■ <code><trigger-properties></code>

trigger-properties Element

The `<trigger-properties>` element is the container for the `<trigger-property>` elements.

The `<trigger-properties>` element is optional. The `<trigger-properties>` element has no attributes.

Table 6-23. Element Hierarchy

Parent Element	Child Element
<trigger>	<trigger-property>

trigger-property Element

The <trigger-property> element defines the properties that identify a trigger object.

The <trigger-property> element is optional. A plug-in can have an unlimited number of <trigger-property> elements. The <trigger-property> element has the following attributes.

Type	Value	Description
name	Trigger name	A name for the trigger. Optional attribute.
display-name	Trigger name	The name that displays in the Orchestrator client. Optional attribute.
type	Trigger type	The object type that defines the trigger. Mandatory attribute.

Table 6-24. Element Hierarchy

Parent Element	Child Elements
<trigger-properties>	None

gauge Element

The <gauge> element defines the gauges you can use for this finder. You must implement `theregisterEventPublisher()` and `unregisterEventPublisher()` methods of `IPluginAdaptor` to set gauges.

The <gauge> element is optional. A plug-in can have an unlimited number of <gauge> elements. The <gauge> element has the following attributes.

Type	Value	Description
name	Gauge name	A name for the gauge. Mandatory attribute.
min-value	Number	Minimum threshold. Optional attribute.
max-value	Number	Maximum threshold. Optional attribute.
unit	Object type	Object type that defines the gauge. Mandatory attribute.
format	String	The format of the monitored value. Optional attribute.

Table 6-25. Element Hierarchy

Parent Element	Child Element
<events>	<description>

scripting-objects Element

The <scripting-objects> element is the container for the <object> elements.

The <scripting-objects> element is optional. The <scripting-objects> element has no attributes.

Table 6-26. Element Hierarchy

Parent Element	Child Element
<module>	<object>

object Element

The <object> element maps the plugged-in technology's constructors, attributes, and methods to JavaScript object types that the Orchestrator scripting API exposes.

See [Naming Plug-In Objects](#) for object naming conventions.

The <object> element is optional. A plug-in can have an unlimited number of <object> elements. The <object> element has the following attributes.

Type	Value	Description
script-name	JavaScript name	Scripting name of the class. Must be globally unique. Mandatory attribute.
java-class	Java class	The Java class wrapped by this JavaScript class. Mandatory attribute.
create	true (default) or false	If true, you can create a new instance of this class. Optional attribute.
strict	true or false (default)	If true, you can only call methods you annotate or declare in the vso.xml file. Optional attribute.
is-deprecated	true or false (default)	If true, the object maps a deprecated Java class. Optional attribute.
since-version	String	Version since the Java class is deprecated. Optional attribute.

Table 6-27. Element Hierarchy

Parent Element	Child Elements
<scripting-objects>	<ul style="list-style-type: none"> ■ <description> ■ <deprecated> ■ <url> ■ <constructors> ■ <attributes> ■ <methods> ■ <singleton>

constructors Element

The <constructors> element is the container for the <object><constructor> elements.

The <constructors> element is optional. The <constructors> element has no attributes.

Table 6-28. Element Hierarchy

Parent Element	Child Element
<object>	<constructor>

constructor Element

The <constructor> element defines a constructor method. The <constructor> method produces documentation in the API Explorer.

The <constructor> element is optional. A plug-in can have an unlimited number of <constructor> elements. The <constructor> element has no attributes.

Table 6-29. Element Hierarchy

Parent Element	Child Elements
<constructors>	<ul style="list-style-type: none"> ■ <description> ■ <parameters>

Constructor parameters Element

The <parameters> element is the container for the <constructor><parameter> elements.

The <parameters> element is optional. The <parameters> element has no attributes.

Table 6-30. Element Hierarchy

Parent Element	Child Element
<constructor>	<parameter>

Constructor parameter Element

The <parameter> element defines the constructor's parameters.

The `<parameter>` element is optional. A plug-in can have an unlimited number of `<parameter>` elements. The `<parameter>` element has the following attributes.

Type	Value	Description
name	String	Parameter name to use in API documentation. Mandatory attribute.
type	Orchestrator parameter type	Parameter type to use in API documentation. Mandatory attribute.
is-optional	true or false	If true, value can be null. Optional attribute.
since-version	String	Method version. Optional attribute.

Table 6-31. Element Hierarchy

Parent Element	Child Elements
<code><parameters></code>	None

attributes Element

The `<attributes>` element is the container for the `<object>``<attribute>` elements.

The `<attributes>` element is optional. The `<attributes>` element has no attributes.

Table 6-32. Element Hierarchy

Parent Element	Child Element
<code><object></code>	<code><attribute></code>

attribute Element

The `<attribute>` element maps the attributes of a Java class from the plugged-in technology to JavaScript attributes that the Orchestrator JavaScript engine makes available.

The `<attribute>` element is optional. A plug-in can have an unlimited number of `<attribute>` elements. The `<attribute>` element has the following attributes.

Type	Value	Description
java-name	Java attribute	Name of the Java attribute. Mandatory attribute.
script-name	JavaScript object	Name of the corresponding JavaScript object. Mandatory attribute.

Type	Value	Description
return-type	String	The type of object this attribute returns. Appears in the API Explorer documentation. Optional attribute. Note If the JavaScript return type is Properties, the supported underlying Java implementations are <code>java.util.HashMap</code> and <code>java.util.Hashtable</code> .
read-only	true or false	If true, you cannot modify this attribute. Optional attribute.
is-optional	true or false	If true, this field can be null. Optional attribute.
show-in-api	true or false	If false, this attribute does not appear in API documentation. Optional attribute.
is-deprecated	true or false	If true, the object maps a deprecated attribute. Optional attribute.
since-version	Number	The version at which the attribute was deprecated. Optional attribute.

Table 6-33. Element Hierarchy

Parent Element	Child Elements
<attributes>	None

methods Element

The <methods> element is the container for the <object><method> elements.

The <methods> element is optional. The <methods> element has no attributes.

Table 6-34. Element Hierarchy

Parent Element	Child Element
<object>	<method>

method Element

The <method> element maps a Java method from the plugged-in technology to a JavaScript method that the Orchestrator JavaScript engine exposes.

The <method> element is optional. A plug-in can have an unlimited number of <method> elements. The <method> element has the following attributes.

Type	Value	Description
java-name	Java method	Name of the Java method signature with argument types in parentheses, for example, getVms(DataStore). Mandatory attribute.
script-name	JavaScript method	Name of the corresponding JavaScript method. Mandatory attribute.
return-type	Java object type	The type this method obtains. Optional attribute. Note If the JavaScript return type is Properties, the supported underlying Java implementations are java.util.HashMap and java.util.Hashtable.
static	true or false	If true, this method is static. Optional attribute.
show-in-api	true or false	If false, this method does not appear in API documentation. Optional attribute.
is-deprecated	true or false	If true, the object maps a deprecated method. Optional attribute.
since-version	Number	The version at which the method was deprecated. Optional attribute.

Table 6-35. Element Hierarchy

Parent Element	Child Elements
<methods>	<ul style="list-style-type: none"> ■ <deprecated> ■ <description> ■ <example> ■ <parameters>

example Element

The <example> element allows you to add code examples to Javascript methods that appear in the API Explorer documentation.

The <example> element is optional. The <example> element has no attributes.

Table 6-36. Element Hierarchy

Parent Element	Child Elements
<method>	<ul style="list-style-type: none"> ■ <code> ■ <description>

code Element

The `<code>` element provides example code that appears in the API Explorer documentation.

You provide the code example between the `<code>` and `</code>` tags. The `<code>` element is optional. The `<code>` element has no attributes.

Table 6-37. Element Hierarchy

Parent Element	Child Elements
<code><example></code>	None

Method parameters Element

The `<parameters>` element is the container for the `<method>``<parameter>` elements.

The `<parameters>` element is optional. The `<parameters>` element has no attributes.

Table 6-38.

Parent Element	Child Element
<code><method></code>	<code><parameter></code>

Method parameter Element

The `<parameter>` element defines the method's input parameters.

The `<parameter>` element is optional. A plug-in can have an unlimited number of `<parameter>` elements. The `<parameter>` element has the following attributes.

Type	Value	Description
name	String	Parameter name. Mandatory attribute.
type	Orchestrator parameter type	Parameter type. Mandatory attribute.
is-optional	true or false	If true, value can be null. Optional attribute.
since-version	String	Method version. Optional attribute.

Table 6-39. Element Hierarchy

Parent Element	Child Element
<code><parameters></code>	None

singleton Element

The `<singleton>` element creates a JavaScript scripting object as a singleton instance.

A singleton object behaves in the same way as a static Java class. Singleton objects define generic objects for the plug-in to use, rather than defining specific instances of objects that Orchestrator accesses in the plugged-in technology. For example, you can use a singleton object to establish the connection to the plugged-in technology.

The `<singleton>` element is optional. The `<singleton>` element has the following attributes.

Type	Value	Description
script-name	JavaScript object	Name of the corresponding JavaScript object. Mandatory attribute.
datasource	Java object	The source Java object for this JavaScript object. Mandatory attribute.

Table 6-40. Element Hierarchy

Parent Element	Child Element
<code><object></code>	None

enumerations Element

The `<enumerations>` element is the container for the `<enumeration>` elements.

The `<enumerations>` element is optional. The `<enumerations>` element has no attributes.

Table 6-41. Element Hierarchy

Parent Element	Child Element
<code><module></code>	<code><enumeration></code>

enumeration Element

The `<enumeration>` element defines common values that apply to all objects of a certain type.

If all objects of a certain type require a certain attribute, and if the range of values for that attribute is limited, you can define the different values as enumeration entries. For example, if a type of object requires a `color` attribute, and if the only available colors are red, blue, and green, you can define three enumeration entries to define these three color values. You define entries as child elements of the enumeration element.

The `<enumeration>` element is optional. A plug-in can have an unlimited number of `<enumeration>` elements. The `<enumeration>` element has the following attribute.

Type	Value	Description
type	Orchestrator object type	Enumeration type. Mandatory attribute.

Table 6-42. Element Hierarchy

Parent Element	Child Elements
<code><enumerations></code>	<ul style="list-style-type: none"> ■ <code><url></code> ■ <code><description></code> ■ <code><entries></code>

entries Element

The <entries> element is the container for the <enumeration><entry> elements.

The <entries> element is optional. The <entries> element has no attributes.

Table 6-43. Element Hierarchy

Parent Element	Child Element
<enumeration>	<entry>

entry Element

The <entry> element provides a value for an enumeration attribute.

The <entry> element is optional. A plug-in can have an unlimited number of <entry> elements. The <entry> element has the following attributes.

Type	Value	Description
id	Text	The identifier that objects use to set the enumeration entry as an attribute. Mandatory attribute.
name	Text	The entry name. Mandatory attribute.

Table 6-44. Element Hierarchy

Parent Element	Child Elements
<entries>	None

Best Practices for Orchestrator Plug-In Development

You can improve certain aspects of the Orchestrator plug-ins that you develop by understanding the structure and content of plug-ins, as well as by understanding how to avoid specific problems.

- [Approaches for Building Orchestrator Plug-Ins](#)

You can use different approaches to build your Orchestrator plug-ins. You can start building a plug-in layer by layer or you can start building all layers of the plug-in at the same time.

- [Types of Orchestrator Plug-Ins](#)

By using plug-ins, you can integrate general-purpose libraries or utilities like XML or SSH, as well as entire systems, such as vCloud Director, with Orchestrator. Depending on the technology that you integrate with Orchestrator, plug-ins can be categorized as plug-ins for services, or general purpose plug-ins, and plug-ins for systems.

■ Plug-In Implementation

You can use certain helpful practices and techniques when you structure your plug-ins, implement the required Java classes and JavaScript objects, develop the plug-in workflows and actions, as well as provide the workflow presentation.

■ Recommendations for Orchestrator Plug-In Development

Adhering to certain certain practices when developing the different components of your Orchestrator plug-ins helps you to improve the quality of the plug-ins.

■ Documenting Plug-In User Interface Strings and APIs

When you write user interface (UI) strings for Orchestrator plug-ins and the related API documentation, follow the accepted rules of style and format.

Approaches for Building Orchestrator Plug-Ins

You can use different approaches to build your Orchestrator plug-ins. You can start building a plug-in layer by layer or you can start building all layers of the plug-in at the same time.

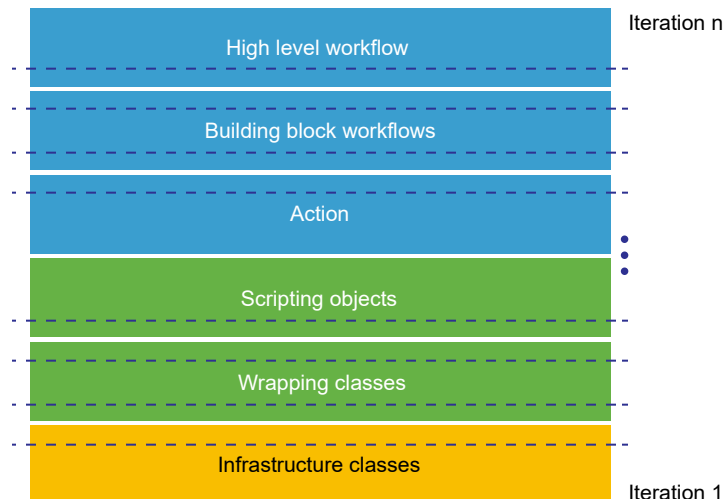
For information about plug-in layers, see [Structure of an Orchestrator Plug-In](#).

Bottom-Up Plug-In Development

A plug-in can be built layer by layer using bottom-up development approach.

Bottom-up development approach builds the plug-in layer by layer starting from the lower level layers and continuing with the higher level layers. When this approach is mixed with an interactive and iterative development approach, then part or whole layer is delivered for each iteration. At the end of the N iterations the plug-in is completely finished.

Figure 6-3. Bottom-up plug-in development



An advantage of the bottom-up plug-in development approach is that development is focused on one layer at a time.

Consider the following disadvantages of bottom-up plug-in development approach.

- The progress of the plug-in development is difficult to show until some insertions are completed.
- It does not fit very well in an Agile development practices.

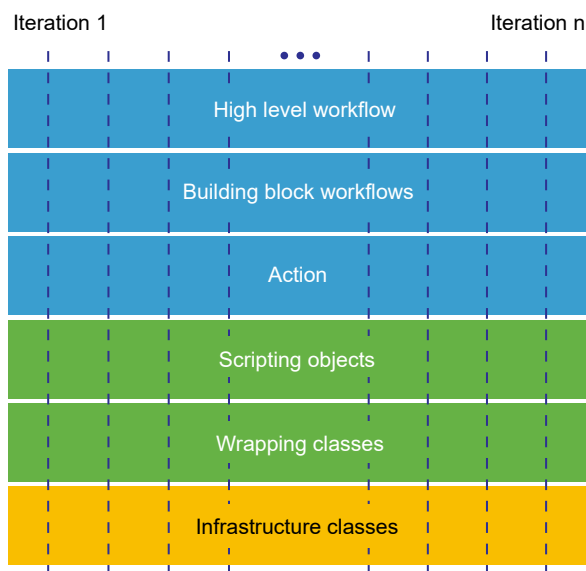
The bottom-up development process is considered good enough for small plug-ins, with reduced or non-existent set of wrapping classes, scripting objects, actions, or workflows.

Top-Down Plug-In Development

A plug-in can be built by slicing it into top-down functionality, using top-down development approach.

When the top-down approach is mixed with an Agile development process, new functionality is delivered for each iteration. As a result, at the end of the iteration N the plug-in is completely implemented.

Figure 6-4. Top-down plug-in development



The top-down plug-in development approach has the following advantages.

- The progress of the plug-in development is easy to show from the first iteration because new functionality is completed for each iteration and the plug-in can be released and used after every iteration.
- Completing a vertical slice of functionality allows for very clearly defined success criteria and definition of what has been done, as well as better communication between developers, product management, and quality assurance (QA) engineers.
- Allows the QA engineers to start testing and automating from the beginning of the development process. Such an approach results in valuable feedback and decreases the overall project delivery time frame.

A disadvantage of the top-down plug-in development approach is that the development is in progress on different layers at the same time.

You should apply the top-down plug-in development process for most plug-ins. It is appropriate for plug-ins with dynamic requirements.

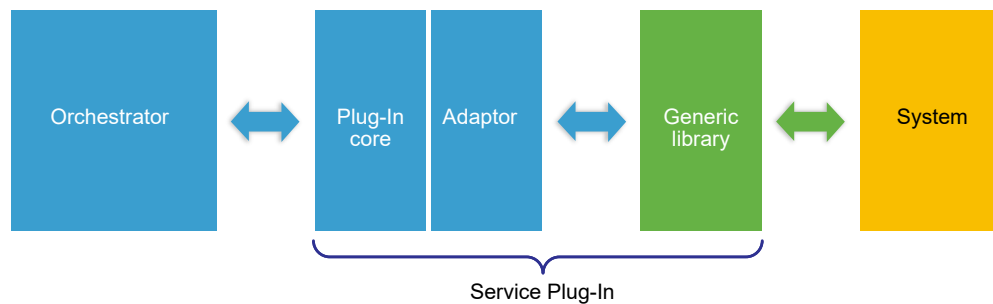
Types of Orchestrator Plug-Ins

By using plug-ins, you can integrate general-purpose libraries or utilities like XML or SSH, as well as entire systems, such as vCloud Director, with Orchestrator. Depending on the technology that you integrate with Orchestrator, plug-ins can be categorized as plug-ins for services, or general purpose plug-ins, and plug-ins for systems.

Plug-Ins for Services

Plug-ins for services or general-purpose plug-ins provide functionality that can be considered as a service inside Orchestrator.

Figure 6-5. Architecture of plug-ins for services



Plug-ins for services expose generic libraries or utilities to Orchestrator, such as XML, SSH, or SOAP. For example, the following plug-ins that are available in Orchestrator are plug-ins for services.

JDBC plug-in

Lets you use any database within a workflow.

Mail plug-in

Lets you send emails within a workflow.

SSH plug-in

Lets you open SSH connections and run commands within a workflow.

XML plug-in

Lets you manage XML documents within a workflow.

Plug-ins for services have the following characteristics.

Complexity

Plug-ins for services have low to medium levels of complexity. Plug-ins for services expose a specific library, or part of a library, inside Orchestrator so as to provide concrete functionality. For example, the XML plug-in adds an implementation of a Document Object Model (DOM) XML parser to the Orchestrator JavaScript API.

Size

Plug-ins for services are relatively small in size. They require the same basic set of classes as for all plug-ins, and other classes that offer new scripting objects to add new functionality.

Inventory

Plug-ins for services require a small inventory of objects to work, or they do not require an inventory at all. Plug-ins for services have a generic and small object model, and so, they do not need to show this model inside the Orchestrator inventory.

Plug-Ins for Systems

Plug-ins for systems connect the Orchestrator workflow engine to an external system so that you can orchestrate the external system.

Following are examples for plug-ins for systems.

vCenter Server plug-in

Lets you manage vCenter Server instances using workflows.

vCloud Director plug-in

Lets you interact with a vCloud Director installation within a workflow.

Cisco UCSM plug-in

Lets you interact with Cisco entities within a workflow.

Following are the main characteristics of plug-ins for systems.

Complexity

Plug-ins for systems have a higher level of complexity than general-purpose plug-ins, because the technologies that they expose are relatively complex. Plug-ins for systems must represent all the elements of the external system inside Orchestrator to interact with the external system and offer its functionality in Orchestrator. If the external system provides an integration mechanism, you can use it to expose the functionality of the system in Orchestrator more easily. However, besides representing the elements of the external system in Orchestrator, plug-ins for systems might also need to offer high scalability, provide a caching mechanism, deal with events and notifications, and so on.

Size

Plug-ins for system are medium to big in size. Plug-ins for systems require many classes apart from the basic set of classes because usually they offer a large number of scripting objects.

Plug-ins for systems might require some other helper and auxiliary classes that will interact with them.

Inventory

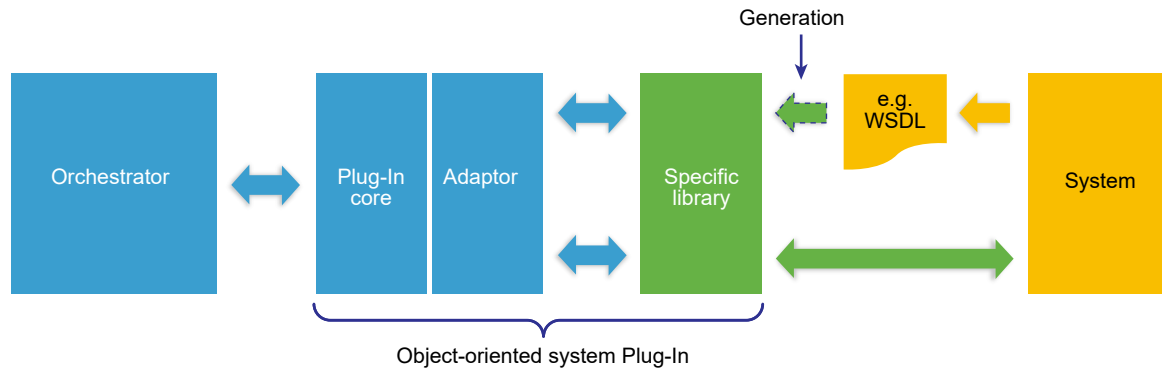
Usually, plug-ins for systems have a large number of objects, and you must expose these objects properly in the inventory so that you can locate them and work with them easily in Orchestrator. Because of the large number of objects that plug-ins for systems need to expose, you should build auxiliary tool or a process to auto-generate as much code as possible for the plug-in. For example, the vCenter Server plug-in provides such a tool.

Plug-Ins for Object-Oriented Systems

Object-oriented systems offer an interaction mechanism that is based on objects and RPC.

The most widely used model for an object-oriented system is the Web service model that uses SOAP. The objects inside this model have a set of attributes that are related to the state of the objects and offer a set of remote methods that are invoked on the target system side.

Figure 6-6. Plug-Ins for Object-Oriented Systems



You can consider the following when you implement plug-ins for object-oriented systems.

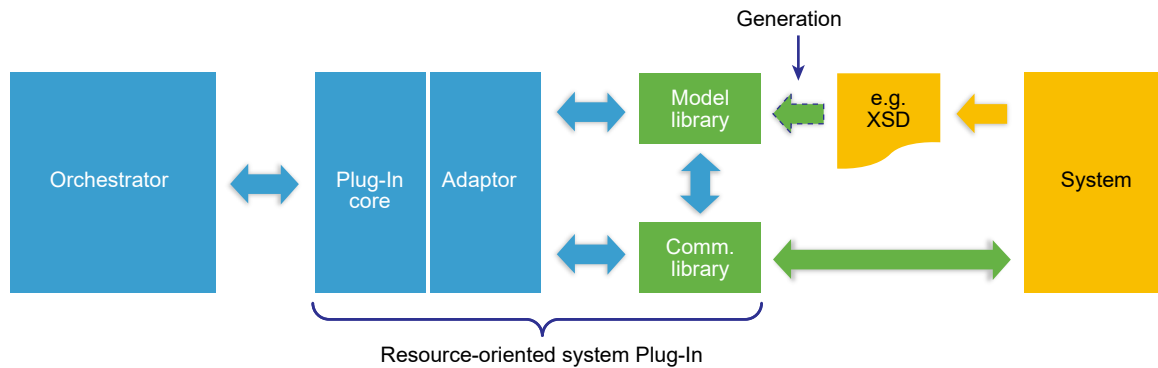
- If you use SOAP, you can use the WSDL file to generate a set of classes that combine the object model and the communication mechanism.
- This object model is almost everything that you have to expose inside Orchestrator.

Plug-Ins for Resource-Oriented Systems

Resource-oriented systems provide an interaction mechanism that is based on resources and simple operations that use HTTP methods.

The most representative model for a resource-oriented system is the REST model, combined for example with XML. The objects inside this model have a set of attributes that are related to their state. To invoke methods on the target system (communication mechanism), you must use the standard HTTP methods such as GET, POST, PUT, and so on, and follow some conventions.

Figure 6-7. Plug-ins for resource-oriented systems



You can consider the following when you develop plug-ins for resource-oriented systems.

- If you use REST or only HTTP with XML, you get one or more XML schema files to be able to read and write messages. From these schemas, you can generate a set of classes that define the object model. This set of classes only defines the state of the objects because the operations are defined implicitly with the HTTP methods, for example, as defined in the vCloud Director plug-in, or explicitly with some specific XML messages, such as the Cisco UCSM plug-in.
- You need to implement the communication mechanism in another set of classes. This set of classes defines a new object model that interacts with the original object model. The object model for the communication mechanism consists of objects and methods only.
- You can expose both the original object model and the object model for the communication mechanism inside Orchestrator. This might add some complexity depending on how both object models are exposed, and on whether you are merging related objects from both sides (to simulate an object-oriented system) or keeping them separate.

Plug-In Implementation

You can use certain helpful practices and techniques when you structure your plug-ins, implement the required Java classes and JavaScript objects, develop the plug-in workflows and actions, as well as provide the workflow presentation.

■ [Project Structure](#)

You can apply a standard structure for the projects of your Orchestrator plug-ins.

■ [Project Internals](#)

You can apply certain approaches when implementing your plug-in, for example, cache objects, bring objects in background, clone objects, and so on. By following such approaches, you can improve the performance of your plug-ins, avoid concurrency problems, and improve the responsiveness of the Orchestrator client.

■ Workflow Internals

You can implement a workflow to monitor long-time operations that your Orchestrator plug-in performs.

■ Workflows and Actions

To ease the workflow development and usage, you can use certain good practices.

■ Workflow Presentation

When you create the presentation of a workflow, you should apply certain structure and rules.

Project Structure

You can apply a standard structure for the projects of your Orchestrator plug-ins.

You can use a standard Maven structure with modules for your plug-in projects to bring clarity in where every piece of functionality resides.

Table 6-45. Structure of a Plug-In Project

Module	Description
/myAwesomePlugin-plugin	The root of the plug-in project.
/o11nplugin-myAwesomePlugin	The module that composes the final plug-in DAR file.
/o11nplugin-myAwesomePlugin-config	The module that contains the plug-in configuration Web application. It generates a standard WAR file.
/o11nplugin-myAwesomePlugin-core	The module that contains all the classes that implement any of the standard Orchestrator plug-in interfaces and other auxiliary classes that they use. It generates a standard JAR file.
/o11nplugin-myAwesomePlugin-model	The module that contains all the classes that help you integrate the third-party technology with Orchestrator through the plug-in. The classes should not contain any direct reference to the standard Orchestrator plug-in APIs.
/o11nplugin-myAwesomePlugin-package	The module that imports an external Orchestrator package file with actions and workflows to include it inside the final plug-in DAR file. The module is optional.

Project Internals

You can apply certain approaches when implementing your plug-in, for example, cache objects, bring objects in background, clone objects, and so on. By following such approaches, you can improve the performance of your plug-ins, avoid concurrency problems, and improve the responsiveness of the Orchestrator client.

Cache Objects

Your plug-in can interact with a remote service, and this interaction is provided by local objects that represent remote objects on the service side. To achieve good performance of the plug-in as well as good responsiveness of the Orchestrator UI, you can cache the local objects instead of getting them every time from the remote service. You can consider the scope of the cache, for example, one cache for all the plug-in clients, one cache per user of the plug-in, and one cache per user of the third-party service. When implemented, your caching mechanism is integrated with the plug-in interface for finding and invalidating objects.

Bring Objects in Background

If you have to show large lists of objects in the plug-in inventory and do not have a fast way to retrieve those objects, you can bring objects in background. You can bring object in background, for example, by having objects with two states, `fake` and `loaded`. Assume that the fake objects are very easy to create and provide the minimal information that you have to show in the inventory, such as name and ID. Then it would be possible to always return fake objects, and when all the information (the real object) is really needed, the using entity or the plug-in can invoke a method `load` automatically to get the real object. You can even configure the process of loading objects to start automatically after the fake objects are returned, to anticipate the actions of the using entity.

Clone Objects to Avoid Concurrency Problems

If you use a cache for your plug-in, you have to clone objects. Use of a cache that always returns the same instance of an object to every entity that requests it can have unwanted effects. For example, entity A requests object O, and the entity views the object in the inventory with all its attributes. At the same time, entity B requests object O as well, and entity A runs a workflow that starts changing the attributes of object O. At the end of its run, the workflow invokes the object's update method to update the object on the server side. If entity A and entity B get the same instance of object O, entity A views in the inventory all the changes that entity B performs, even before the changes are committed on the server side. If the run goes fine, it should not be a problem, but if the run fails, the attributes of object O for entity A are not reverted. In such a case, if the cache (the find operations of the plug-in) returns a clone of the object instead of the same instance all the time, each using entity views and modifies its own copy, avoiding concurrency issues, at least within Orchestrator.

Notify Changes to Others

Problems might occur when you use a cache and clone objects simultaneously. The biggest one is that the object that is using entity views might not be the latest version that is available for the object. For example, if an entity displays the inventory, the objects are loaded once, but at the same time, if another entity is changing some of the objects, the first entity does not view the changes. To avoid this problem, you can use the `PluginWatcher` and `IPluginPublisher` methods from the Orchestrator plug-in API to notify that something has changed to allow other instances of Orchestrator clients to see the changes. This also applies to a unique instance of the

Orchestrator client when changes from one object from the inventory affect other objects of the inventory, and they need to be notified too. The operations that are prone to use notifications are adding, updating, and deleting objects when these objects, or some properties of these objects, are shown in the inventory.

Enable Finding Any Object at Any Time

You must implement the `find` method of the `IPluginFactory` interface to find objects just by type and ID. The `find` method can be invoked directly after restarting Orchestrator and resuming a workflow.

Simulate a Query Service if You Do Not Have One

The Orchestrator client can require querying for some objects in specific cases or showing them not as a tree but as a list or a table, for example. This means that your plug-in must be able to query for some set of objects at any moment. If the third-party technology offers a query service, you need to adapt and use this service. Otherwise, you should be able to simulate a query service, despite of the higher complexity or the lower performance of the solution.

Find Methods Should Not Return Runtime Exceptions

The methods from the `IPluginFactory` interface that implement the searches inside the plug-in should not throw controlled or uncontrolled runtime exceptions. This might be the cause of strange *validation error* failures when a workflow is running. For example, between two nodes of a workflow, the `find` method is invoked if an output from the first node is an input of the second node. At that moment, if the object is not found because of any runtime exception, you might get no more information than a *validation error* in the Orchestrator client. After that, it depends on how the plug-in logs the exceptions in to get more or less information inside the log files.

Workflow Internals

You can implement a workflow to monitor long-time operations that your Orchestrator plug-in performs.

You can implement a workflow for monitoring long-time running operations such as task monitoring. This workflow can be based on Orchestrator triggers and waiting events. You must consider that a workflow that is blocked waiting for a task can be resumed as soon as the Orchestrator server starts. The plug-in must be able to get all the required information to resume the monitoring process properly.

The monitoring workflow or the task that it can use internally should provide a mechanism to specify the polling rate and a possible timeout.

The process of debugging a piece of scripting code inside a workflow is not easy, especially if the code does not invoke any Java code. Because of this, sometimes the only option is to use the logging methods offered by the default Orchestrator scripting objects.

Workflows and Actions

To ease the workflow development and usage, you can use certain good practices.

Start Developing Workflows as Building Blocks

A building block can be a simple workflow that requires a few input parameters and returns a simple output. If you have a rich set of building blocks, you can create higher-level workflows easily, and you can offer a better set of tools for composing complex workflows.

Create Higher-Level Workflows Based on Smaller Components

If you have to develop a complex workflow with several inputs and internal steps, you can split it into smaller and simpler building block workflows and actions.

Create Actions Whenever Possible

You can create actions to achieve additional flexibility when you develop workflows.

- To create complex objects or parameters for scripting methods easily
- To avoid repeating common pieces of code all the time
- To perform UI validations

Workflows Should Invoke Actions Whenever Possible

Actions can be invoked directly as nodes inside the workflow schema. This can keep the workflow schema simpler, because you do not need to add scripting code blocks to invoke a single action.

Fill In the Expected Information

Provide information for every element of a workflow or an action.

- Provide a description of the workflow or action.
- Provide a description of the input parameters.
- Provide a description of the outputs.
- Provide a description of the attributes for the workflows.

Keep the Version Information Updated

When you version plug-ins, add meaningful comments with information such as major updates to the plug-in, important implementation details, and so on.

Workflow Presentation

When you create the presentation of a workflow, you should apply certain structure and rules.

Use the following properties for the workflow inputs in the workflow presentation.

Table 6-46. Properties for Workflow Inputs

Properties	Usage
Show in Inventory	Use this property to help the user to run a workflow from the inventory view.
Specify a root object to be shown in the chooser	Use this property to help the user to select inputs. If the root object can be refreshed in the presentation, is an attribute, or is retrieved by an object method, you need to create or set an appropriate action to refresh the object in the presentation.
Maximum string length	Use this property for long strings such as names, descriptions, file paths, and so on.
Minimum string length	Use this property to avoid empty strings from the testing tools.
Custom validation	Implement non-simple validations with actions.

Organize the inputs with steps and display group. Such organization helps the user identify and distinguish all the input parameters of a workflow.

Recommendations for Orchestrator Plug-In Development

Adhering to certain practices when developing the different components of your Orchestrator plug-ins helps you to improve the quality of the plug-ins.

Table 6-47. Useful Practices in Plug-In Implementation

Component	Item	Description
General	Access to third-party API	Plug-ins should provide simplified methods for accessing the third-party API wherever possible.
	Interface	Plug-ins should provide a coherent and standard interface for users, even when the API does not.
Action	Scripting objects	You should create actions for every creation, modification, deletion, and all other methods available for a scripting object.
	Description	The description of an action should describe what the action does instead of how it works.
	Scripting	When you use scripting to get the properties or methods of an object, you can check whether the object value is different from null or undefined.
	Deprecation	If an action is deprecated, the comment or the throw statement should indicate the replacement action, or the action should call a new replacement action so that solutions that are built on the deprecated version of the action do not fail.
Workflow	User interface operations in the orchestrated technology	You should create a workflow for every operation that is available in the user interface of the orchestrated technology.
	Description	The description of a workflow should describe what the workflow does instead of how it works.

Table 6-47. Useful Practices in Plug-In Implementation (continued)

Component	Item	Description
	Presentation property mandatory input	You must set the mandatory input property for all mandatory workflow inputs.
	Presentation property default value	If you develop a workflow that configures an entity, the workflow presentation should load the default configuration values for this entity. For example, if you develop a workflow that is named Host Configuration, the presentation of the workflow must load the default values of the host configuration.
	Presentation property Show in inventory	You must set the Show in inventory property so that you have contextual workflows on inventory objects.
	Presentation property specify a root parameter	You should use this property in workflows when it is not necessary to browse the inventory from the tree root .
	Workflow validation	You must validate workflows and fix all errors.
	Object creation	All workflows that create a new object should return the new object as an output parameter.
	Deprecation	If a workflow is deprecated, the comment or the throw statement should indicate the replacement workflow, or the deprecated workflow should call a new replacement workflow to ensure that solutions that are built on previous versions of the workflow do not fail.
Inventory	Host disconnection	If your inventory contains a connection to a host and this host becomes unavailable, you should indicate that the host is disconnected. You can do this either by renaming the root object by appending – disconnected or by removing the tree of objects underneath this object, in the same manner as the vCloud Director plug-in does.
	Select value as list property	An inventory object must be selectable as treeview or a list.
	Host manager	If the plug-in implements a host object for the target system, then a parent hostmanager root object should exist with properties for adding, removing, or editing host properties.
	Getting or updating objects	If a query service is running on the orchestrated technology, you should use it for getting multiple objects.
	Child discovery	If you need to retrieve child objects separately, the retrieval process must be multithreaded and non-blocking on a single error.
	Orchestrator object change	All workflows that can change the state of an element in the inventory must update the inventory to avoid having objects out of synchronization.

Table 6-47. Useful Practices in Plug-In Implementation (continued)

Component	Item	Description
	External object change	You can use a notification mechanism to notify about changes in the orchestrated technology that occur as a result of operations that are performed outside of Orchestrator. In case such operations lead to removal of objects from the orchestrated technology, you must refresh the inventory accordingly to avoid failures or loss of data. For example, if a virtual machine is deleted from vCenter Server, the vCenter Server plug-in updates the inventory to remove the object of the removed virtual machine.
	Finder object	Finder objects should have properties that can be used to differentiate objects. These are typically the properties that are present in the user interface.
Scripting object	Implementation	The <code>equals</code> method must be implemented to insure that <code>==</code> operation works on the same object as in some cases the object might have two instances.
	Plug-in object properties	Objects that have parent objects should implement a parent property.
	Plug-in object properties	Objects that have child objects should implement <code>GET</code> methods that return arrays of child objects.
	Inventory objects	Inventory objects should be searchable with <code>Server.find</code> .
		All inventory objects should be serializable so they can be used as input or output attributes in a workflow.
	Constructor and methods	In most cases, scriptable objects should have either a constructor, or should be returned by other object attributes or methods.
	Object ID	Objects that have an ID that is issued from an external system should use an internal ID to ensure that no ID duplication occurs when you are orchestrating more than one server.
	Searching for objects	<code>search</code> or <code>find</code> methods should implement a filter so that the specified name or ID can be found instead of just all objects. For example, the Orchestrator server has a <code>Server.FindById</code> method that allows finding a plug-in object by its ID. To do this, the method must be implemented for each findable object in the plug-in.
	Trigger	If possible, triggers should be available for objects that change so that Orchestrator can have policies triggered on various events. For example, to determine when a new virtual machine is added, powered on, powered off, and so on, Orchestrator can monitor a trigger or an event in the vCenter plug-in on the <code>Datacenter</code> object.
	Object properties	Objects that reside in other plug-ins should have properties for being easily converted from one plug-in object to another. For example, virtual machine objects need to have a <code>moref</code> (managed object reference ID).

Table 6-47. Useful Practices in Plug-In Implementation (continued)

Component	Item	Description
	Session manager	If you are connecting to a remote server that can have a different session, the plug-in should implement a shared session and a session per user.
Trigger	Trigger	All long operations and blocking methods should be able to start asynchronously with a task returned, and generate a trigger event on completion.
Enumerations	Enums	Enumerations for a given type should have an inventory object that allows selecting from the different values in the enumeration.
Logging	Logs	Methods should implement different log levels.
Versioning	Plug-in version	The plug-in version should follow standards and be updated along with the plug-in update.
API documentation	Methods	Methods that are described in the API documentation should never throw the exception <code>no xyz method / property</code> on an object. Instead, methods should return <code>null</code> when no properties are available and be documented with details when these properties are not available.
	<code>vso.xml</code>	All objects, methods, and properties must be documented in <code>vso.xml</code> .

Documenting Plug-In User Interface Strings and APIs

When you write user interface (UI) strings for Orchestrator plug-ins and the related API documentation, follow the accepted rules of style and format.

General Recommendations

- Use the official names for VMware products involved in the plug-in. For example, use the official names for the following products and VMware terminology.

Correct Term	Do Not Use
vCenter Server	VC or vCenter
vCloud Director	vCloud

- End all workflow descriptions with a period. For example, `Creates a new Organization.` is a workflow description.
- Use a text editor with a spell checker to write the descriptions and then move them to the plug-in.
- Ensure that the name of the plug-in exactly matches the approved third-party product name that it is associated with.

Workflows and Actions

- Write informative descriptions. One or two sentences are enough for most of the actions and workflows.
- Higher-level workflows might include more extensive descriptions and comments.
- Start descriptions with a verb, for example, `Creates....`. Do not use self-referential language like `This workflow creates.`
- Put a period at the end of descriptions that are complete sentences.
- Describe what a workflow or action does instead of how it is implemented.
- Workflows and actions usually are included in folders and packages. Include a small description for these folders and packages as well. For example, a workflow folder can have a description similar to `Set of workflows related to vApp Template management.`

Parameters of Workflows and Actions

- Start workflow and action descriptions with a descriptive noun phrase, for example, `Name of.`. Do not use a phrase like `It's the name of.`
- Do not put a period at the end of parameter and action descriptions. They are not complete sentences.
- Input parameters of workflows must specify a label with appropriate names in the presentation view. In many cases, you can combine related inputs in a display group. For example, instead of having two inputs with the labels `Name of the Organization` and `Full name of the Organization`, you can create a display group with the label `Organization` and place the inputs `Name` and `Full name` in the `Organization` group.
- For steps and display groups, add descriptions or comments that appear in the workflow presentation as well.

Plug-In API

- The documentation of the API refers to all of the documentation in the `vso.xml` file and the Java source files.
- For the `vso.xml` file, use the same rules for the descriptions of finder objects and scripting objects with their methods that you use for workflows and actions. Descriptions of object attributes and method parameters use the same rules as the workflow and action parameters.
- Avoid special characters in the `vso.xml` file and include the descriptions inside a `<![CDATA[insert your description here!]]>` tag.
- Use the standard Javadoc style for the Java source files.

Creating Plug-Ins by Using Maven

7

The Orchestrator Appliance provides a repository containing Maven artifacts, which you can use to create plug-in projects from archetypes.

The repository is hosted at `https://orchestrator_server:8281/vco-repo/` or `http://orchestrator_server:8280/vco-repo/`, in case your Maven version does not support the HTTPS protocol. This location is embedded in the `pom.xml` file of standard Orchestrator Maven plug-in projects. You can only access the repository if you have deployed the Orchestrator Appliance.

This chapter includes the following topics:

- [Create an Orchestrator Plug-In with Maven from an Archetype](#)
- [Maven Archetypes](#)
- [Maven-Based Plug-In Development Best Practices](#)

Create an Orchestrator Plug-In with Maven from an Archetype

You can create a standard Orchestrator Maven plug-in from an archetype by running commands in the command-line interface.

Prerequisites

- Verify that you have installed Orchestrator Appliance 5.5.1 or later.
- Verify that you have installed Apache Maven 3.0.4 or 3.0.5.

Procedure

- 1 Create a project in interactive mode by choosing an archetype.

```
mvn archetype:generate -DarchetypeCatalog=https://orchestrator_server:8281/vco-repo/archetype-catalog.xml -DrepoUrl=https://orchestrator_server:8281/vco-repo -Dmaven.repo.remote=https://orchestrator_server:8281/vco-repo -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true
```

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

- 2 (Optional) If you cannot access the repository over HTTPS, you can access it over HTTP. If you access the repository over HTTP or have a valid SSL certificate, you can create a project without using the `-Dmaven.wagon.http.ssl.allowall=true` flag.

```
mvn archetype:generate -DarchetypeCatalog=http://orchestrator_server:8280/vco-repo/archetype-catalog.xml -DrepoUrl=http://orchestrator_server:8280/vco-repo -Dmaven.repo.remote=http://orchestrator_server:8280/vco-repo -Dmaven.wagon.http.ssl.insecure=true
```

- 3 Navigate to the project directory and build the plug-in.

```
cd project_dir && mvn clean install -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true
```

Results

If the build process is successful, the plug-in `.dar` file is generated in the DAR module's `target/` directory.

Maven Archetypes

You can use a set of predefined Maven archetypes as templates for developing Orchestrator plug-ins.

The following table describes the default Maven archetypes available in Orchestrator.

Table 7-1. Default Maven Archetypes

Archetype	Description
<code>com.vmware.o11n:011n-plugin-archetype-simple</code>	<code>com.vmware.o11n:011n-plugin-archetype-simple</code>
<code>com.vmware.o11n:011n-package-archetype</code>	A content-only Maven project, which can be used to keep packages in source form for better interaction with RCS, diff, post-processing, and so on.
<code>com.vmware.o11n:011n-client-archetype-rest</code>	A simple command-line tool, which communicates with the Orchestrator REST API and calls a workflow.
<code>com.vmware.o11n:011n-plugin-archetype-inventory</code>	A plug-in that demonstrates inventory use. The plug-in implements a repository, an adapter, and a factory for a single type. The inventory is stored in a file on a disk.

Table 7-1. Default Maven Archetypes (continued)

Archetype	Description
<code>com.vmware.o11n:011n-archetype-inventory-annotation</code>	A plug-in whose <code>vso.xml</code> descriptor is generated on top of annotations.
<code>com.vmware.o11n:011n-archetype-spring</code>	A plug-in that uses Spring-based SDK, provides a DI-enabled environment, and adds higher-level services in comparison to standard plug-in APIs.
<code>com.vmware.o11n:011n-plugin-archetype-modeldriven</code>	An archetype that generates a plug-in skeleton for building plug-ins with <code>ModelDriven</code> .

Maven-Based Plug-In Development Best Practices

You can improve the process for delivering Orchestrator plug-ins created with Maven by performing a set of tasks.

Using a Repository Manager

If you are creating plug-ins in a larger organization, use an enterprise repository manager to set up the default Orchestrator Appliance repository to be added as a proxy repository. Using a central repository improves management and plug-in project collaboration. When you complete the first build in the new repository, the repository manager caches the artifacts from the Orchestrator Appliance repository and you can turn off the default repository.

Locking Workflows

After you verify that all workflows in your plug-in work as expected, lock them to prevent unauthorized modifications. By locking workflows, you ensure that the basic functions of the plug-in cannot be compromised. If users must modify a default workflow for a specific purpose, they can create a copy of the original workflow and edit that copy.

There are two ways to produce release builds with locked workflows.

- Pass the `-DallowedMask=vf` parameter to Maven.
- Edit the `pom.xml` and change the value of the `allowedMask` parameter to `vf`.

```
<allowedMask>vf</allowedMask>
```

Using a Package-Signing Certificate

Use a self-signed certificate or a certificate signed by a Certificate Authority, to ensure the integrity and authenticity of the plug-ins. Store the certificate in the keystore under the `_dunesrsa_alias_ alias`, by importing it with the `keytool` in the JDK.

There are two ways to specify the path to the keystore file and the keystore password.

- Define the `-DkeystoreLocation` and `-DkeystorePassword` command-line parameters for the `MAVEN_OPTS` variable.
- Edit the `pom.xml` file to insert the values manually. For example,

```
<keystore>path to the keystore file</keystore>  
<storepass>keystore password</storepass>
```

If no keystore is imported, the `.package` file is signed with the `archetype.keystore` file.