Developing Plug-Ins with VMware vCenter Orchestrator

vCenter Orchestrator 5.5

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

EN-001139-00

vmware®

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

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

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

docfeedback@vmware.com

Copyright © 2011–2013 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at http://www.vmware.com/go/patents.

VMware is a registered trademark or trademark of VMware, Inc. in the United States and other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

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

Contents

Developing Plug-Ins with VMware vCenter Orchestrator 9

1 Overview of Plug-Ins 11
- Structure of an Orchestrator Plug-In 12
- Exposing an External API to Orchestrator 13
- Components of a Plug-In 13
- Role of the vso.xml File 14
- Roles of the Plug-In Adapter 15
- Roles of the Plug-In Factory 16
- Role of Finder Objects 16
- Role of Scripting Objects 17
- Role of Event Handlers 17

2 Contents and Structure of a Plug-In 19
- Defining the Application Mapping in the vso.xml File 19
- Format of the vso.xml Plug-In Definition File 20
- Naming Plug-In Objects 21
  - Plug-In Object Naming Conventions 22
- File Structure of the Plug-In 22

3 Create an Orchestrator Plug-In 25
- Accessing the Orchestrator Plug-In API 26
- Obtain an Application to Plug in to Orchestrator 27
- Components of the Solar System Application 27
  - CelestialBody.java Class 28
  - Star.java Class 28
  - Planet.java Class 28
  - Moon.java Class 29
  - ISolarSystemListener.java Class 29
  - SolarSystemEventHandler.java Class 29
  - SolarSystemRepository.java Class 30
- Components of the Solar System Plug-In 30
- Create a Plug-In Factory 31
  - Set Up the Plug-In Factory Implementation 32
  - Set Up Event Listeners and Notification Handlers 33
  - Find Objects By Identifier in the Plugged-In Technology 34
  - Find Objects in the Plugged-In Technology By a Query 35
  - Find Objects By Relation Type in the Plugged-In Technology 36
  - Discover Whether an Object has Children of a Given Relation Type 37
- Create a Plug-In Event Listener 38
  - Set Up the Event Listener Implementation 38

Register the Event Listener with the Plugged-In Technology 39
Notify Orchestrator of Events in the Plugged-In Technology 40
Create a Plug-In Event Generator 41
Set Up the Event Generator 42
Create Event Publishers 43
Define and Publish Events to Orchestrator 44
Create a Plug-In Workflow Trigger 46
Set Up the Workflow Trigger 47
Create Instances of the PluginTrigger Class 48
Set the Properties that a Workflow Trigger Monitors 48
Create Plug-In Watchers 50
Set Up the Watcher Implementation 50
Create Instances of the PluginWatcher Class 52
Publish Plug-In Watchers 52
Define Objects and Methods to Map to the Orchestrator JavaScript API 55
Create a Plug-In Adapter 56
Set Up the Plug-In Adapter Implementation 57
Instantiate the Plug-In Factory 57
Manage Plug-In Events 59
Add Plug-In Watchers 60
Add a Tab to the Configuration Interface 61
Set Up the Configuration Adapter 62
Load and Save Configuration Information in the Configuration Server 63
Create a Configuration Action to Obtain Configuration Information from the User 65
Create a Struts-Based Web Application to Add to the Configuration Interface 67
Map the Application in the vso.xml File 70
Set Up the Global Plug-In Information 71
Map Objects in the Plugged-In Technology to Scripting Types and Inventory Objects 72
Define Enumerations 74
Map Classes and Methods to Classes and Methods in the JavaScript API 75
Create the Plug-In DAR Archive 77
Build the Solar System Application and Plug-In 78
Contents of the Solar System DAR File 78
Install a Plug-In in the Orchestrator Server 79
Interact with the Solar System Application by Using Orchestrator 80
View Plug-In Scripting Objects in the JavaScript API 80
Run Workflows on Plug-In Objects in the Inventory 81
Monitor Plug-In Events by Using Policies 82
Monitor Plug-In Events by Using Workflows 83
Access Plug-In Objects and Operations by Using a Web View 83

4 API Enhancements for Plug-In Development 85
Orchestrator Annotations API 85
Enable Annotation-Based Configuration 85
Annotating Objects 86
Java-Based Configuration API for the Plug-In Definition File 86
Using Java-Based Configuration 87
Orchestrator Spring-Based Plug-In API 88
Spring-Based API Basic Configuration 88

Orchestrator Workflow Generation API 89
Generating Actions 89
Generating Workflows 89
Orchestrator SSL Configuration API 90
SSL Configuration Methods 90
The HostValidator Helper Class 92

5 Orchestrator Plug-In API Reference 93
IAop Interface 94
IConfigurationAdaptor Interface 94
IDynamicFinder Interface 95
IPluginAdaptor Interface 95
IPluginEventPublisher Interface 96
IPluginFactory Interface 97
IPluginNotificationHandler Interface 97
IPluginPublisher Interface 98
WebConfigurationAdaptor Interface 98
BaseAction Class 99
ConfigurationError Class 99
PluginLicense Class 99
PluginTrigger Class 100
PluginWatcher Class 101
QueryResult Class 101
SDKFinderProperty Class 102
SDKHelper Class 103
PluginExecutionException Class 104
PluginLicenseException Class 104
PluginOperationException Class 104
ConfigurationError.Severity Enumeration 105
ErrorLevel Enumeration 105
HasChildrenResult Enumeration 106
ScriptingAttribute Annotation Type 107
ScriptingFunction Annotation Type 107
ScriptingParameter Annotation Type 108

6 Elements of the vso.xml Plug-In Definition File 109
module Element 110
configuration Element 111
description Element 112
deprecated Element 112
url Element 112
installation Element 113
action Element 113
webview-components-library Element 113
finder-datasources Element 114
finder-datasource Element 114
inventory Element 115
finders Element 115

finder Element 116
properties Element 117
property Element 117
relations Element 118
relation Element 118
id Element 118
inventory-children Element 119
relation-link Element 119
events Element 119
trigger Element 119
trigger-properties Element 120
trigger-property Element 120
gauge Element 120
scripting-objects Element 121
object Element 121
constructors Element 122
constructor Element 122
Constructor parameters Element 122
Constructor parameter Element 122
attributes Element 123
attribute Element 123
methods Element 124
method Element 124
example Element 125
code Element 125
Method parameters Element 125
Method parameter Element 125
singleton Element 126
enumerations Element 126
enumeration Element 126
entries Element 127
entry Element 127

7 Best Practices for Orchestrator Plug-In Development 129
Approaches for Building Orchestrator Plug-Ins 129
Bottom-Up Plug-In Development 129
Top-Down Plug-In Development 130
Types of Orchestrator Plug-Ins 131
Plug-Ins for Services 131
Plug-Ins for Systems 132
Plug-In Implementation 134
Project Structure 134
Project Internals 135
Workflow Internals 136
Workflows and Actions 136
Workflow Presentation 137
Recommendations for Orchestrator Plug-In Development 138
Documenting Plug-In User Interface Strings and APIs 140

Index 143

Developing Plug-Ins with VMware vCenter Orchestrator

Developing Plug-Ins with VMware vCenter Orchestrator provides information about developing plug-ins with VMware vCenter Orchestrator.

Intended Audience

This information is intended for plug-in developers who are familiar with virtual machine technology, datacenter operations, and vCenter Orchestrator.

Overview of Plug-Ins 1

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 on page 12

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

- Exposing an External API to Orchestrator on page 13

  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 on page 13

  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 on page 14

  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 on page 15

  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 on page 16

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

- Role of Finder Objects on page 16

  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 on page 17

  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 on page 17

  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.

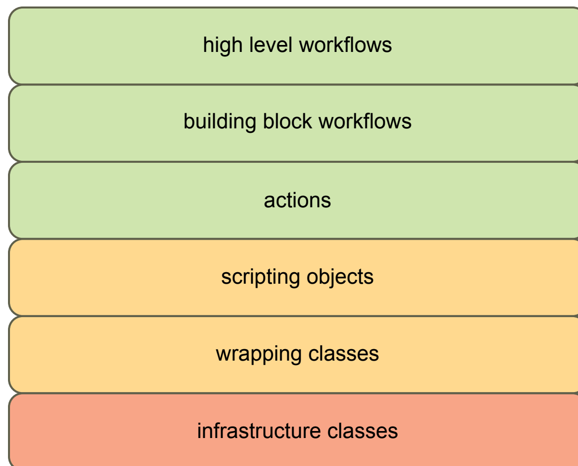
Structure of an Orchestrator Plug-In

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

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

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

Figure 1-1. Structure of a vCO 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, Web views, 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

You can add a tab to the Orchestrator configuration interface in which you can configure the plug-in after you install it. For example, you can add a tab to the Orchestrator configuration interface in which users provide information about the environment where the plugged-in technology runs. 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. The plug-in configuration tab is optional.

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 Chapter 5, "Orchestrator Plug-In API Reference," on page 93. For an examination of an example implementation of the `IPluginAdaptor` interface, see "Create a Plug-In Adapter," on page 56.

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 Chapter 5, "Orchestrator Plug-In API Reference," on page 93. For an examination of an example implementation of the `IPluginFactory` interface, see "Create a Plug-In Factory," on page 31.

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 worklows that are waiting for that event. Watchers are optional.

Contents and Structure of a Plug-In 2

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 configuration interface to import a DAR file to the Orchestrator server.

- Defining the Application Mapping in the vso.xml File on page 19

  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 on page 20

  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 on page 21

  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.

- File Structure of the Plug-In on page 22

  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 Chapter 6, "Elements of the vso.xml Plug-In Definition File," on page 109.

Format of the vso.xml Plug-In Definition File

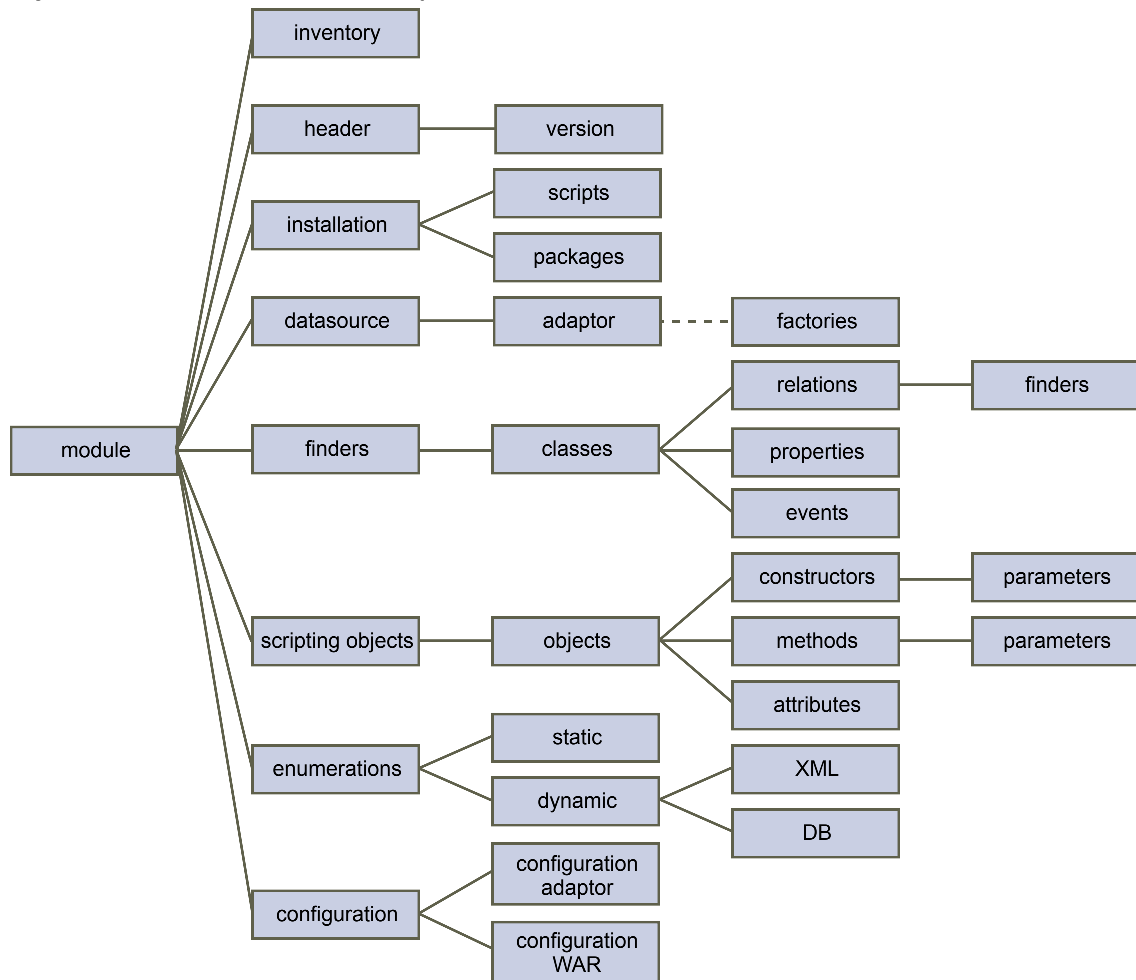
The `vso.xml` file defines how the Orchestrator server interacts with the plugged-in technology. You must include a reference to every type of object or operation to expose to Orchestrator in the `vso.xml` file.

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 2-1. 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 2-1 shows rules that apply to individual object types.

Table 2-1. Plug-In Object Naming Rules

Object Type	Naming Rules
Plug-In	■ Defined in the `<module>` element in the `vso.xml` 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	■ Defined in the `<finder>` elements in the `vso.xml` file. ■ Must adhere to Java class naming conventions. ■ Must be unique in the plug-in. 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 `VirtualMachine` object type from the vCenter Server plug-in appears in the Orchestrator scripting API as `VC:VirtualMachine`.
Scripting object	■ Defined in the `<scripting-object>` elements in the `vso.xml` 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 `VirtualMachine` class from the vCenter Server plug-in appears in the Orchestrator scripting API as the `VcVirtualMachine` 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 2-2. Structure of the DAR Archive

Folders	Description
plug-in_name\VSO-INF\	Contains the `vso.xml` file that defines the mapping of the objects in the plugged-in technology to Orchestrator objects. The `VSO-INF` folder and the `vso.xml` file are mandatory.
plug-in_name\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.
plug-in_name\resources\	Contains resource files that the plug-in requires. The `resources` folder can include the following types of element: ■ 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, Web views, and other resources that interact with the objects that you access by using the plug-in. You can organize resources in subfolders. For example, `resources\images\`, `resources\scripts\`, or `resources\packages\`. The `resources` folder is optional.
plug-in_name\webapps\	Contains the WAR file of the Web application that adds a tab for the plug-in to the Orchestrator configuration interface or the files of a Web view for the plug-in. The `webapps` folder is optional.

You use the Orchestrator configuration interface to import a DAR file to the Orchestrator server.

Create an Orchestrator Plug-In 3

To create a plug-in to use Orchestrator to manage an external application, you must create a plug-in adapter and a plug-in factory, create any event handlers, and map the objects from the plugged-in application to Orchestrator objects in the `vso.xml` file.

The procedure to create a plug-in consists of several subprocedures. These procedures demonstrate the plug-in creation process by examining the Java classes, resources, and `vso.xml` file for a plug-in to an example Java application. The example application that these procedures examine represents the solar system. The example contains Java objects to represent the Sun, the planets, and their moons. The Java objects also define operations that you can perform on the objects. The Orchestrator plug-in for this application allows you to use Orchestrator to manage the solar system application. When you install the example plug-in, you can use Orchestrator to perform the operations on the objects of the solar system application by running workflows and setting policies.

Procedure

1 Accessing the Orchestrator Plug-In API on page 26

   The Orchestrator plug-in API provides Java interfaces that you implement to create the plug-in adapter and plug-in factory. The plug-in adapter and factory expose the objects and operations of the plugged-in technology to the Orchestrator server.

2 Obtain an Application to Plug in to Orchestrator on page 27

   To create a plug-in, you must have an application to expose for Orchestrator to manage.

3 Components of the Solar System Application on page 27

   The solar system application replicates a solar system and includes objects to represent stars, planets, and moons. The solar system application also defines operations that you can perform on these objects.

4 Components of the Solar System Plug-In on page 30

   The solar system plug-in implements a plug-in adapter, plug-in factory, and event handlers to expose the objects and functions of the solar system application to Orchestrator.

5 Create a Plug-In Factory on page 31

   To create a plug-in factory, you create a Java class that implements the `IPluginFactory` interface from the Orchestrator plug-in API.

6 Create a Plug-In Event Listener on page 38

   Plug-in event listeners allow Orchestrator to monitor events that occur in the plugged-in technology. To create a plug-in event listener, you create a Java class that implements the `IPluginNotificationHandler` interface from the Orchestrator plug-in API.

7 Create a Plug-In Event Generator on page 41

You can create one or more event generators in a plug-in to perform operations on the objects in the plugged-in technology. The event generator generates events that the Orchestrator plug-in, rather than the plugged-in technology, defines.

8 Create a Plug-In Workflow Trigger on page 46

You can create plug-in workflow triggers to monitor events in the plugged-in technology on behalf of a Wait Event element in a workflow. To create a workflow trigger, you create a Java class that implements the `PluginTrigger` class from the Orchestrator plug-in API.

9 Create Plug-In Watchers on page 50

Plug-in watchers watch triggers on behalf of workflows that are waiting for the event that the trigger starts. To create a plug-in watcher, you create a Java class that implements the `PluginWatcher` class from the Orchestrator plug-in API. You publish the watcher on the Orchestrator notification server by implementing the `IPluginPublisher` interface.

10 Define Objects and Methods to Map to the Orchestrator JavaScript API on page 55

You can map the object types, classes, and methods of the plugged-in technology and the plug-in itself to JavaScript types, classes, and methods that you add to the Orchestrator JavaScript API.

11 Create a Plug-In Adapter on page 56

To create a plug-in adapter, you create a Java class that implements the `IPluginAdaptor` interface from the Orchestrator plug-in API. The adapter instantiates the plug-in factory and event management implementations.

12 Add a Tab to the Configuration Interface on page 61

You can add a tab to the Orchestrator configuration interface to allow users to provide information to the plug-in configuration that is specific to their environment or preferences.

13 Map the Application in the vso.xml File on page 70

The `vso.xml` file defines how Orchestrator accesses and interacts with the plugged-in technology. The `vso.xml` file maps objects and operations in the plugged-in technology and in the plug-in implementation to Orchestrator objects and operations.

14 Create the Plug-In DAR Archive on page 77

The final stage in the creation of a plug-in is to create the DAR archive that you import to Orchestrator.

15 Install a Plug-In in the Orchestrator Server on page 79

After you create the plug-in DAR file, you must install it in the Orchestrator server. You install plug-ins in the Orchestrator configuration interface.

16 Interact with the Solar System Application by Using Orchestrator on page 80

After you install a plug-in in the Orchestrator server, you can use the objects that it adds to the Orchestrator JavaScript API to create workflows, actions, policies, Web views, and so on. You use these items to interact with the plugged-in technology using Orchestrator.

Accessing the Orchestrator Plug-In API

The Orchestrator plug-in API provides Java interfaces that you implement to create the plug-in adapter and plug-in factory. The plug-in adapter and factory expose the objects and operations of the plugged-in technology to the Orchestrator server.

The plug-in API includes interfaces, classes, and annotations that you can use when you create the plug-in adapter, factory, and event management implementations. For the full list of the classes in the Orchestrator plug-in API, see Chapter 5, “Orchestrator Plug-In API Reference,” on page 93.

Locating the Plug-In API Java Archives

Orchestrator provides the classes of the plug-in API in the Orchestrator plug-in API Java archive (JAR) file, . To develop the plug-in adapter and factory implementations, you must include the file in your classpath. You might also need the utility classes that the archive provides.

Table 3-1. Locations of JAR File and Utility Class Archive

Option	Location
If you installed the standalone version of Orchestrator.	
If the vCenter Server installer installed Orchestrator.	

If you develop a plug-in that requires a tab in the Orchestrator configuration interface, you must include the file in your classpath.

Table 3-2. Location of the Orchestrator Configuration Tab JAR File

Option	Location
If you installed the standalone version of Orchestrator.	`*install-directory*\VMware\Orchestrator\configuration\jetty\lib\ext`
If the vCenter Server installer installed Orchestrator.	`*install-directory*\VMware\Infrastructure\Orchestrator\configuration\jetty\lib\ext`

Obtain an Application to Plug in to Orchestrator

To create a plug-in, you must have an application to expose for Orchestrator to manage.

The solar system example application demonstrates how to create a plug-in. The vCO Plug-in SDK ZIP file that you can download from the VMware Communities site contains the source files for the solar system application, and the source files for its plug-in implementation.

You can examine the source files of the solar system application and solar system plug-in. You can modify the source files to adapt and extend the solar system application and solar system plug-in, and build a DAR file to incorporate your modifications in the plug-in.

Procedure

1 Download the vCO Plug-in SDK ZIP file from the VMware Communities site.

2 Unzip the bundle to an appropriate location.

3 Navigate to the following location to view the files of the solar system application and the solar system plug-in.

   install_directory\VMware-VCO-Plug-In-SDK\samples\vCOPluginSDKSamplePlugin-SolarSystem

Components of the Solar System Application

The solar system application replicates a solar system and includes objects to represent stars, planets, and moons. The solar system application also defines operations that you can perform on these objects.

You can find the source files of the solar system application in the VMware-VCO-Plug-In-SDK\samples\vCOPluginSDKSamplePlugin-SolarSystem\src\o11nplugin-solarsystem-model\src\main\java\com\vmware\solarsystem\ folder in the vCO Plug-in SDK samples bundle.

For simplicity, the solar system application runs in the JVM of the Orchestrator server when you install the solar system plug-in. You can create plug-ins for technologies that run independently of Orchestrator by defining how Orchestrator connects to the application in a function in the plug-in. For example, you can add connection information in a tab for the plug-in in the Orchestrator configuration interface.

NOTE The source files of the solar system example application and solar system plug-in are provided for reference purposes, so that you can see the details of the application that the solar system plug-in exposes and of the plug-in implementation. If you adapt the code of the solar system application or the solar system plug-in, you can build the application and the DAR file to incorporate your adaptations.

CelestialBody.java Class

The `CelestialBody.java` class is a serializable class that defines a generic celestial body, that can be a star, a planet, or a moon.

The `CelestialBody` class declares the following constructor and methods:

- `CelestialBody()` constructor, to create a generic celestial body instance
- `getId()` method, to return an object identifier
- `setId()` method, to set an object identifier
- `getName()` method, to return an object name
- `setName()` method, to set an object name

Star.java Class

The `Star.java` class extends `CelestialBody` to create star objects.

The `Star` class adds the following constructor and methods:

- `Star()` constructor, to create star instances
- `getCircumference()` method, to return the circumference of a star
- `setCircumference()` method, to set the circumference of a star
- `getSurfaceTemp()` method, to return the surface temperature of a star
- `setSurfaceTemp()` method, to set the surface temperature of a star
- `getPlanets()` method, to return a list of planets that orbit a star
- `addPlanet()` method, to add a planet to the list of planets that orbit a star
- `removePlanet()` method, to remove a planet from the list of planets that orbit a star

Planet.java Class

The `Planet.java` class extends `CelestialBody` to create planet objects.

The `Planet` class declares the following constructor and methods:

- `Planet()` constructor, to create planet instances
- `getCircumference()` method, to return the circumference of a planet
- `setCircumference()` method, to set the circumference of a planet
- `getGravity()` method, to return the gravity of a planet
- `setGravity()` method, to set the gravity of a planet
- `getMoons()` method, to return a list of moons that orbit a planet

- `addMoon()` method, to add a moon to the list of moons that orbit a planet
- `removeMoon()` method, to remove a moon from the list of moons that orbit a planet
- `getStarId()` method, to return the identifier of the star that the planet orbits
- `setStarId()` method, to set the identifier of the star that the planet orbits

Moon.java Class

The `Moon.java` class extends `CelestialBody` to create moon objects.

The `Moon` class declares the following constructor and methods:

- `Moon()` constructor, to create moon instances
- `getVolume()` method, to return the volume of a moon
- `setVolume()` method, to set the volume of a moon
- `getPlanetId()` method, to return the identifier of the planet that this moon orbits
- `setPlanetId()` method, to set the identifier of the planet that this moon orbits

ISolarSystemListener.java Class

The `ISolarSystemListener.java` class extends `java.util.EventListener` to create a listener that monitors events in the solar system application.

The `ISolarSystemListener` class declares the following methods:

- `circumferenceChanged()`, to monitor changes in the circumference of a planet
- `gravityChanged()`, to monitor changes in the gravity of a planet
- `planetAdded()`, to monitor the creation of new planets
- `planetRemoved()`, to monitor the destruction of planets

SolarSystemEventHandler.java Class

The `SolarSystemEventHandler.java` class creates an array of `ISolarSystemListener` instances and defines methods to handle the events that the `ISolarSystemListener` instances observe.

The `SolarSystemEventHandler` defines the following methods:

- `registerListener()`, to add a listener to the array of `ISolarSystemListener` instances
- `unregisterListener()`, to remove a listener from the array of `ISolarSystemListener` instances
- `fireCircumferenceChanged()`, to register a change in the circumference of a planet
- `fireGravityChanged()`, to register a change in the gravity of a planet
- `firePlanetAdded()`, to register the creation of a planet
- `firePlanetRemoved()`, to register the destruction of a planet

SolarSystemRepository.java Class

The `SolarSystemRepository.java` class implements all of the classes of the solar system application to create an instance of a solar system.

When the solar system application runs, it creates a unique `SolarSystemRepository` instance that represents Earth's solar system. The `SolarSystemRepository` starts a `SolarSystemEventHandler` instance to monitor events in the solar system, and creates instances of the `Star`, `Planet`, and `Moon` classes that represent the Sun, Earth, Mars, Titan, and so on. The `SolarSystemRepository` constructor calls the `Star.addPlanet()` and `Planet.addMoon()` methods to add the planets to the Sun and the moons to the planets, and sets their respective names, identifiers, and attributes.

The `SolarSystemRepository` class defines the following constructor and methods:

- `SolarSystemRepository()` constructor, to create a unique `SolarSystemRepository` instance and a `SolarSystemEventHandler` instance
- `getUniqueInstance()`, to return a unique `SolarSystemRepository` instance
- `getStar()`, to return the star of this solar system instance
- `getAllPlanets()`, to return a list of the planets that orbit the star
- `getPlanetById()`, to return a planet by its unique identifier
- `getAllMoons()`, to return a list of the moons that orbit a planet
- `getMoonById()`, to return a moon by its unique identifier

Components of the Solar System Plug-In

The solar system plug-in implements a plug-in adapter, plug-in factory, and event handlers to expose the objects and functions of the solar system application to Orchestrator.

You find the source files of the solar system plug-in in the `VMware-VCO-Plug-In-SDK\samples\vCOPluginSDKSamplePlugin-SolarSystem\src\o11nplugin-solarsystem-core\src\main\java\com\vmware\orchestrator\api\sample\solarsystem\` folder in the vCO Plug-in SDK samples bundle .

The `SolarSystemConfigureAction.java` file is in the `o11nplugin-solarsystem-config` directory rather than in `o11nplugin-solarsystem-core`.

NOTE The source files of the solar system example application and solar system plug-in are provided for reference purposes, so that you can see the details of the application that the solar system plug-in exposes and of the plug-in implementation. If you adapt the code of the solar system application or the solar system plug-in, you can build the application and the DAR file to incorporate your adaptations.

The following table lists the Java files of the solar system application.

Table 3-3. Source Files for the Solar System Plug-In Implementation

Class Name	Description
`SolarSystemAdapter.java`	Implements the `IPluginAdaptor` interface that defines for Orchestrator the entry point of the solar system application. Instantiates the solar system factory and creates instances of event generators, publishers, and watchers.
`SolarSystemFactory.java`	Implements the `IPluginFactory` interface that defines how Orchestrator uses the plug-in to find solar system objects, and how to perform operations on those objects.

Table 3-3. Source Files for the Solar System Plug-In Implementation (Continued)

Class Name	Description
`SolarSystemEventGenerator.java`	Defines methods to publish events to Orchestrator and a method to generate solar flares on `Star` objects in the solar system application. Creates a `StarFlareEventListener` object that listens for solar flare events on `Star` objects in the solar system application.
`SolarSystemEventListener.java`	Implements the `IPluginNotificationHandler` interface, registers listeners with the notification handler to listen for events in the solar system application, and sends notifications of the events to Orchestrator.
`SolarSystemTriggerGenerator.java`	Creates triggers that allow you to start solar flare events in the solar system application from Orchestrator.
`SolarSystemWatchersManager.java`	Implements `StarFlareEventListener` to monitor solar flare events in the solar system application and performs functions in Orchestrator if the solar flare exceeds a certain magnitude.

Create a Plug-In Factory

To create a plug-in factory, you create a Java class that implements the `IPluginFactory` interface from the Orchestrator plug-in API.

These procedures present the steps involved in creating a plug-in factory. To illustrate the process, they present code from the `SolarSystemFactory` class from the solar system plug-in.

You can download the vCO Plug-in SDK ZIP file from the VMware Communities site to obtain the sources of the solar system example application and plug-in.

For a description of the role of the plug-in factory and the other components of a plug-in, see Chapter 1, "Overview of Plug-Ins," on page 11. For information about all of the methods and parameters of the factory interface, see "IPluginFactory Interface," on page 97.

Procedure

1 Set Up the Plug-In Factory Implementation on page 32

   To create a plug-in factory, you create an implementation of the `IPluginFactory` interface from the Orchestrator plug-in API.

2 Set Up Event Listeners and Notification Handlers on page 33

   You activate event listeners and notification handlers for a plug-in in the factory implementation.

3 Find Objects By Identifier in the Plugged-In Technology on page 34

   You can find objects by their identifier in the plugged-in technology by using the `IPluginFactory.find()` method.

4 Find Objects in the Plugged-In Technology By a Query on page 35

   You can find objects in the plugged-in technology by defining a query in the `IPluginFactory.findAll()` method.

5 Find Objects By Relation Type in the Plugged-In Technology on page 36

   You can find objects by their relationship to other objects in the plugged-in technology by using the `IPluginFactory.findRelation()` method. You can also determine whether an object has any dependent child objects by using the `IPluginFactory.hasChildrenInRelation()` method.

6 Discover Whether an Object has Children of a Given Relation Type on page 37

You implement the `IPluginFactory.hasChildrenInRelation()` method to discover whether an object relates to any children by a given type of relation.

Set Up the Plug-In Factory Implementation

To create a plug-in factory, you create an implementation of the `IPluginFactory` interface from the Orchestrator plug-in API.

Prerequisites

- Verify that you have an application to plug in to Orchestrator.
- Verify that you have access to the Orchestrator plug-in API JAR file.

Procedure

1 Create and save a Java file for the plug-in factory implementation named *ApplicationName*`Factory.java`.

In the solar system example, the factory class is named `SolarSystemFactory.java`.

2 Declare the package that contains the Java classes of the plug-in implementation.

The solar system example declares the following package.

```
package com.vmware.orchestrator.api.sample.solarsystem;
```

3 Import the following Orchestrator plug-in API classes with a Java `import` statement.

```
import ch.dunes.vso.sdk.api.HasChildrenResult;
import ch.dunes.vso.sdk.api.IPluginFactory;
import ch.dunes.vso.sdk.api.IPluginNotificationHandler;
import ch.dunes.vso.sdk.api.PluginExecutionException;
import ch.dunes.vso.sdk.api.QueryResult;
```

4 Import the following classes of the application to plug in with a Java `import` statement.

```
import com.vmware.solarsystem.Planet;
import com.vmware.solarsystem.SolarSystemRepository;
```

5 Import any other classes that the factory implementation requires.

In the solar system example, the factory implementation requires the following classes.

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import org.apache.log4j.Logger;
```

6 Declare a public class that implements the `IPluginFactory` interface from the Orchestrator plug-in API.

The solar system example factory declares the `SolarSystemFactory` class.

```
public class SolarSystemFactory implements IPluginFactory {
}
```

What to do next

Set up event listeners and notifications in the plug-in factory.

Set Up Event Listeners and Notification Handlers

You activate event listeners and notification handlers for a plug-in in the factory implementation.

The plug-in adapter creates one plug-in factory for each connection between Orchestrator and the plugged-in technology. Consequently, you set up event listeners and notification handlers in the plug-in factory to listen for events through the connection and to send notifications about the events that the listeners discover.

Prerequisites

- Set up the factory implementation class.
- Declare a public class that implements the `IPluginFactory` interface.

Procedure

1 Set up logging so that Orchestrator can record in the logs the events that occur in the plugged-in technology.

The solar system example uses an instance of `org.apache.log4j.Logger` to log events.

```
public class SolarSystemFactory implements IPluginFactory {
        private static final Logger log = Logger.getLogger(SolarSystemFactory.class);
}
```

2 Set up a notification handler by implementing the `IPluginNotificationHandler` interface from the Orchestrator API.

The `SolarSystemFactory` constructor gets an instance of `IPluginNotificationHandler` named `notificationHandler`.

```
public class SolarSystemFactory implements IPluginFactory {
        private static final Logger log = Logger.getLogger(SolarSystemFactory.class);
        public SolarSystemFactory(IPluginNotificationHandler notificationHandler) {
        }
}
```

3 Create an instance of an event listener that implements the `java.util.EventListener` class.

The solar system plug-in factory creates an instance of the `SolarSystemEventListener` class. The `SolarSystemEventListener` instance monitors an instance of the `SolarSystemRepository` class from the solar system application.

```
public class SolarSystemFactory implements IPluginFactory {
        private static final Logger log = Logger.getLogger(SolarSystemFactory.class);
        public SolarSystemFactory(IPluginNotificationHandler notificationHandler) {
                super();
                new SolarSystemEventListener(
                        SolarSystemRepository.getUniqueInstance(), notificationHandler);
        }
}
```

NOTE The `SolarSystemEventListener` class is an implementation of the `ISolarSystemListener` listener that the solar system application defines. `ISolarSystemListener` implements `java.util.EventListener`. For information about the implementation of `SolarSystemEventListener`, see "Create a Plug-In Event Listener," on page 38.

You set up the event listeners and notification handlers in the plug-in factory, to listen for events in the plugged-in technology and to send notifications about the events.

What to do next

Define methods in the plug-in factory to find objects in the plugged-in technology by name, type, and by their relation to other objects.

Find Objects By Identifier in the Plugged-In Technology

You can find objects by their identifier in the plugged-in technology by using the `IPluginFactory.find()` method.

All instances of objects in the plugged-in technology must have a unique name or identifier for Orchestrator to find them. The `IPluginFactory.find()` method uses the type and identifier to find an object in the plugged-in technology and returns objects of the type `java.lang.Object`.

Prerequisites

- Set up the factory implementation class.
- Create a public constructor that implements the `IPluginFactory` interface.

Procedure

1 Declare the `IPluginFactory.find()` method to find objects of the type `java.lang.Object`.

```
public Object find(String type, String id) {
}
```

2 Write in the logs the type and identifier of the objects that the plug-in factory finds.

```
public Object find(String type, String id) {
 log.debug("find: " + type + ", " + id);
}
```

3 Call the appropriate methods from the plugged-in technology to obtain the identifiers of objects of each different type.

The `SolarSystemFactory` class uses an `if-else` statement to call the `SolarSystemRepository.getStar()`, `getPlanetById()`, and `getMoonById()` methods.

```
public Object find(String type, String id) {
  log.debug("find: " + type + ", " + id);
  if (type.equals("Star")) {
    return SolarSystemRepository.getUniqueInstance().getStar();
  } else if (type.equals("Planet")) {
    return SolarSystemRepository.getUniqueInstance().getPlanetById(id);
  } else if (type.equals("Moon")) {
    return SolarSystemRepository.getUniqueInstance().getMoonById(id);
  } else if (type.equals("Galaxy")) {
    return null; // No object for galaxy defined yet
  } else {
    throw new IndexOutOfBoundsException("Type " + type + "
    + unknown for plugin SolarSystem");
  }
}
```

You implemented the `IPluginFactory.find()` method to find objects by identifier in the plugged-in technology.

What to do next

Define methods in the plug-in factory to find all objects of a certain type.

Find Objects in the Plugged-In Technology By a Query

You can find objects in the plugged-in technology by defining a query in the `IPluginFactory.findAll()` method.

The `findAll()` method takes the object type and a query as parameters. You can define the syntax of the query in the `IPluginFactory` implementation to narrow the search. If you do not define query syntax, `findAll()` returns all objects of the specified type. You can ignore the query and find objects by type, ignore the type and find objects by query, or find objects by both query and type.

The `findAll()` method returns a `QueryResult` object that contains a list of the objects of the corresponding type that the plugged-in technology contains. For information about `QueryResult` objects, see "QueryResult Class," on page 101.

Prerequisites

- Set up the factory implementation class.
- Create a public constructor that implements the `IPluginFactory` interface.

Procedure

1 Declare the `IPluginFactory.findAll()` method to obtain a `QueryResult` object.

```
public QueryResult findAll(String type, String query) {
}
```

2 Write in the logs the type of the objects that the plug-in factory finds and any additional query that narrows the search.

```
public QueryResult findAll(String type, String query) {
  log.debug("findAll: " + type + ", " + query);
}
```

3 Call the appropriate methods from the plugged-in technology to obtain objects of each different type.

The `SolarSystemFactory` class uses an `if-else` statement to call the `SolarSystemRepository.getStar()`, `getAllPlanets()`, and `getAllMoons()` methods.

```
public QueryResult findAll(String type, String query) {
  log.debug("findAll: " + type + ", " + query);
  List<?> list; // The list can contain any element from the plug-in
  if (type.equals("Star")) {
    list = Arrays.asList(SolarSystemRepository.getUniqueInstance().getStar());
  } else if (type.equals("Planet")) {
    list = SolarSystemRepository.getUniqueInstance().getAllPlanets();
  } else if (type.equals("Moon")) {
    list = SolarSystemRepository.getUniqueInstance().getAllMoons();
  } else if (type.equals("Galaxy")) {
    list = Collections.emptyList();
  } else {
  throw new IndexOutOfBoundsException("Type " + type +
                      " unknown for SolarSystem plug-in ");
  }
  return new QueryResult(list);
}
```

The `SolarSystemFactory` implementation of the `findAll()` method does not define a custom query to narrow the search, so it returns a list of all the objects of each given type in the `QueryResult` object.

You defined methods to find objects by their type in the plugged-in technology.

What to do next

Define methods in the plug-in factory to find all objects that relate to other objects by a certain relation type.

Find Objects By Relation Type in the Plugged-In Technology

You can find objects by their relationship to other objects in the plugged-in technology by using the `IPluginFactory.findRelation()` method. You can also determine whether an object has any dependent child objects by using the `IPluginFactory.hasChildrenInRelation()` method.

The `IPluginFactory.findRelation()` returns all of the dependent child objects that relate to a parent object by a certain relation type.

The `IPluginFactory.hasChildrenInRelation()` method returns `HasChildrenResult` objects to confirm whether a parent object has any dependent child objects that relate to it by a given relation type. The possible values of a `HasChildrenResult` object are `yes`, `no`, or `unknown`. For information about `HasChildrenResult` objects, see "HasChildrenResult Enumeration," on page 106.

You define the relations between the objects in the plugged-in technology in the `vso.xml` file for the plug-in.

Prerequisites

- Set up the factory implementation class.
- Create a public constructor that implements the `IPluginFactory` interface.

Procedure

1. Declare the `IPluginFactory.findRelation()` method to return a `java.util.List` instance that lists all the child objects that relate to a parent object by a given relation.

```
public List findRelation(String parentType, String parentId, String relationName) {
}
```

2. Write in the logs the type and identifier of the parent object and the name of the relationship that the child objects have to the parent.

```
public List findRelation(String parentType, String parentId, String relationName) {
  log.debug("findRelation: " + parentType + ", " + parentId + ", " + relationName);
}
```

3. Call the appropriate methods from the plugged-in technology to obtain lists of child objects that relate to their parent objects by different types of relations.

   The `SolarSystemFactory` class uses an `if-else` statement to call the `SolarSystemRepository.getAllPlanets()` method to return a list of all of the planets that relate to a particular star by the `OrbitingPlanets` relation. The `if-else` statement also calls `Planet.getMoons()` to return a list of all of the moons that relate to a particular planet by the `OrbitingMoons` relation.

```
public List findRelation(String parentType, String parentId, String relationName) {
  log.debug("findRelation: " + parentType + ", " + parentId + ", " + relationName);
  if (parentId == null) {
    return Arrays.asList(SolarSystemRepository.getUniqueInstance().getStar());
  }
  if (parentType.equals("Star")) {
    if (relationName.equals("OrbitingPlanets")) {
      return SolarSystemRepository.getUniqueInstance().getAllPlanets();
  } else {
    throw new IndexOutOfBoundsException("Unknown relation name: "
      + relationName);
```

```
        }
      }
      if (parentType.equals("Planet")) {
        if (relationName.equals("OrbitingMoons")) {
          Planet parentPlanet =
            SolarSystemRepository.getUniqueInstance().getPlanetById(parentId);
            if (parentPlanet != null) {
              return parentPlanet.getMoons();
            }
            return Collections.emptyList();
      } else {
        throw new IndexOutOfBoundsException("Unknown relation name: "
          + relationName);
        }
      } else {
        return Collections.emptyList();
      }
    }
```

You defined methods in the `IPluginFactory` implementation to find objects in the plugged-in technology that relate to other objects by a certain relation type.

What to do next

Discover whether an object relates to any child objects by a given type of relation.

Discover Whether an Object has Children of a Given Relation Type

You implement the `IPluginFactory.hasChildrenInRelation()` method to discover whether an object relates to any children by a given type of relation.

You can implement an `if-else` statement in the `hasChildrenInRelation()` method to check for child objects that relate to a parent by a certain relation type. For example, you can implement a function that uses the `hasChildrenInRelation()` method in the solar system example to check whether a `Planet` object has any moons.

The possible return values of the `hasChildrenInRelation()` method are `Yes`, `No`, and `Unknown`. If you do not implement the `hasChildrenInRelation()` method, it returns `Unknown`.

Prerequisites

- Set up the factory implementation class.
- Create a public constructor that implements the `IPluginFactory` interface.

Procedure

◆ Declare the `IPluginFactory.hasChildrenInRelation()` method to discover whether an object has any children of a certain relation type.

  The `SolarSystemFactory` example does not fully implement the `hasChildrenInRelation()` method and returns `unknown` in all cases.

```
public HasChildrenResult hasChildrenInRelation(String parentType,
    String parentId, String relationName) {
  return HasChildrenResult.Unknown;
}
```

You called the `IPluginFactory.hasChildrenInRelation()` method to discover whether an object has any children of a certain relation type.

What to do next

Create an event listener object to allow Orchestrator to monitor events in the plugged-in technology.

Create a Plug-In Event Listener

Plug-in event listeners allow Orchestrator to monitor events that occur in the plugged-in technology. To create a plug-in event listener, you create a Java class that implements the `IPluginNotificationHandler` interface from the Orchestrator plug-in API.

These procedures present the steps involved in creating a plug-in event listener. To illustrate the process, they present code from the `SolarSystemEventListener` class from the solar system example application.

You can download the vCO Plug-in SDK ZIP file from the VMware Communities site to obtain the sources of the solar system example application and plug-in.

For a description of the role of plug-in event listeners and the other components of a plug-in, see Chapter 1, "Overview of Plug-Ins," on page 11. For information about all the methods and parameters of the `IPluginNotificationHandler` interface, see "IPluginNotificationHandler Interface," on page 97.

Procedure

1 Set Up the Event Listener Implementation on page 38

To create a plug-in event listener, you implement the `java.util.Eventlistener` interface and create an instance of the `IPluginNotification` interface from the Orchestrator plug-in API.

2 Register the Event Listener with the Plugged-In Technology on page 39

To monitor events in a plugged-in technology, you must register the event listener from the plug-in with the plugged-in technology and implement a notification handler.

3 Notify Orchestrator of Events in the Plugged-In Technology on page 40

Event listeners implement the `IPluginNotificationHandler` interface from the Orchestrator plug-in API to notify Orchestrator of events in the plugged-in technology.

Set Up the Event Listener Implementation

To create a plug-in event listener, you implement the `java.util.Eventlistener` interface and create an instance of the `IPluginNotification` interface from the Orchestrator plug-in API.

Prerequisites

- Download the bundle of Orchestrator examples.
- Unzip the examples bundle to an appropriate location.

Procedure

1 Create and save a Java file for the plug-in event listener implementation named *ApplicationName*`EventListener.java`.

In the solar system example, the event listener class is named `SolarSystemEventListener.java`.

2 Declare the package that contains the Java classes of the plug-in implementation.

The solar system example declares the following package.

```
package com.vmware.orchestrator.api.sample.solarsystem;
```

3 Import the Orchestrator plug-in API classes with a Java import statement.

In the solar system example, the event listener requires the following class:

```
import ch.dunes.vso.sdk.api.IPluginNotificationHandler;
```

4 Import any other classes that the event listener implementation requires.

   In the solar system example, the event listener requires the following classes from the solar system application:

```
import com.vmware.solarsystem.ISolarSystemListener;
import com.vmware.solarsystem.Planet;
import com.vmware.solarsystem.SolarSystemRepository;
```

5 Declare a public class for the event listener that implements the `java.util.Eventlistener` interface.

   In the solar system example, the event listener declares the following class:

```
public class SolarSystemEventListener implements ISolarSystemListener {
}
```

   The `ISolarSystemListener` class from the solar system application is a subclass of `java.util.Eventlistener`.

6 Create an instance of the `IPluginNotificationHandler` interface.

   The solar system event listener creates an `IPluginNotificationHandler` instance named `notficationHandler`.

```
public class SolarSystemEventListener implements ISolarSystemListener {
    IPluginNotificationHandler notficationHandler;
}
```

You set up the plug-in event listener class.

What to do next

Register the plug-in event listener with the plugged-in technology.

Register the Event Listener with the Plugged-In Technology

To monitor events in a plugged-in technology, you must register the event listener from the plug-in with the plugged-in technology and implement a notification handler.

An event listener requires access to the main class of the application to which it listens for events. An event listener also requires an instance of the `IPluginNotificationHandler` interface from the Orchestrator plug-in API, to send notifications to Orchestrator if the events occur.

Prerequisites

- Set up the event listener implementation class.
- Declare a public class that implements the `java.util.EventListener` interface.
- Create an instance of the `IPluginNotificationHandler` interface.

Procedure

1 Create a public constructor to create event listener instances.

   The solar system example creates a constructor that takes as parameters an instance of the `SolarSystemRepository` class from the solar system application and an instance of the `IPluginNotificationHandler` interface.

```
public SolarSystemEventListener(SolarSystemRepository solarSystemRepository,
    IPluginNotificationHandler notificationHandler) {
}
```

2 Register an instance of the event listener with the plugged-in technology.

   You register the event listener by using the listener registration mechanism that the plugged-in technology defines.

   The `SolarSystemEventListener()` constructor registers the event listener with the solar system application by calling the `SolarSystemRepository.registerListener()` method.

```
public SolarSystemEventListener(SolarSystemRepository solarSystemRepository,
    IPluginNotificationHandler notificationHandler) {
        solarSystemRepository.registerListener(this);
}
```

3 Associate an instance of the `IPluginNotificiationHandler` interface to the event listener.

   The `SolarSystemEventListener()` constructor uses the `this` Java keyword to add an instance of `IPluginNotificiationHandler` to the event listener.

```
public SolarSystemEventListener(SolarSystemRepository solarSystemRepository,
    IPluginNotificationHandler notificationHandler) {
        solarSystemRepository.registerListener(this);
        this.notificationHandler = notificationHandler;
}
```

You created an event listener that registers an event listener and a notification handler with the plugged-in technology. The event listener listens for events in the plugged-in technology and sends notifications to Orchestrator.

What to do next

Call the methods of the `IPluginNotificationHandler` interface to notify Orchestrator of events in the plugged-in technology.

Notify Orchestrator of Events in the Plugged-In Technology

Event listeners implement the `IPluginNotificationHandler` interface from the Orchestrator plug-in API to notify Orchestrator of events in the plugged-in technology.

The `IPluginNotificationHandler` interface defines methods that you implement in the event listener to notify Orchestrator of changes in state of the objects that the event listener monitors in the plugged-in technology.

The `SolarSystemEventListener` class monitors objects in a `SolarSystemRepository` instance for changes in state that the following methods cause:

- `Planet.setCircumference()`, that changes the circumference of a planet.
- `Planet.setGravity()`, that changes the gravity of a planet.
- `Star.addPlanet()`, that adds a planet to a star.
- `Star.removePlanet()`, that removes a planet from a star.

The events that the `SolarSystemEventListener` class monitors are all events that the solar system application defines.

Prerequisites

- Set up the event listener implementation class.
- Declare a public class that implements the `java.util.EventListener` interface.
- Create an instance of the `IPluginNotificationHandler` interface.
- Register the event listener and the notification handler instances with the plugged-in technology.

Procedure

1 Create methods that implement the `IPluginNotificationHandler.notifyElementUpdated()` method to notify Orchestrator of changes to an existing object.

   The `SolarSystemEventListener` class creates the following methods to inform Orchestrator of changes to the circumference and gravity of a particular `Planet` object:

```
public void circumferenceChanged(Planet planet) {
    notificationHandler.notifyElementUpdated("Planet", planet.getId());
}
public void gravityChanged(Planet planet) {
    notificationHandler.notifyElementUpdated("Planet", planet.getId());
}
```

2 Create methods that implement the `IPluginNotificationHandler.notifyElementInvalidate()` method to notify Orchestrator of changes in relations between objects.

   The `SolarSystemEventListener` class creates the following method to notify Orchestrator that a child `Planet` object is added to a parent `Star` object.

```
public void planetAdded(Planet planet) {
    notificationHandler.notifyElementInvalidate("Star", "SUN");
}
```

3 Create methods that implement the `IPluginNotificationHandler.notifyElementDeleted()` method to notify Orchestrator of the removal of an object.

   The `SolarSystemEventListener` class creates the following method to notify Orchestrator that a child `Planet` object is deleted from its parent `Star`.

```
public void planetRemoved(Planet planet) {
    notificationHandler.notifyElementDeleted("Planet", planet.getId());
}
```

You implemented the methods of the `IPluginNotificationHandler` interface to notify Orchestrator of events that occur on the objects in the plugged-in technology.

What to do next

Create an event generator to push to the plugged-in technology events that the Orchestrator plug-in defines.

Create a Plug-In Event Generator

You can create one or more event generators in a plug-in to perform operations on the objects in the plugged-in technology. The event generator generates events that the Orchestrator plug-in, rather than the plugged-in technology, defines.

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.

The solar system plug-in features an event generator that implements an event publisher and creates a method to generate solar flares on a `Star` object in a `SolarSystemRepository` instance. These procedures present the steps involved in creating a plug-in event generator. To illustrate the process, they present code from the `SolarSystemEventGenerator` class. The package of workflows, actions, and resources that accompany the solar system example contains a policy template that monitors a gauge that the `SolarSystemEventGenerator` class publishes to the policy engine.

You can download the vCO Plug-in SDK ZIP file from the VMware Communities site to obtain the sources of the solar system example application and plug-in.

For a description of the role of the plug-in events and the other components of a plug-in, see Chapter 1, "Overview of Plug-Ins," on page 11. For information about all of the methods and parameters of the event publisher interface, see "IPluginEventPublisher Interface," on page 96.

Procedure

1 Set Up the Event Generator on page 42

   You can create `IPluginEventPublisher` instances and methods to generate events directly in the plug-in adaptor implementation. However, the solar system class creates these objects in a separate class.

2 Create Event Publishers on page 43

   You can create `IPluginEventPublisher` instances to publish event gauges and event triggers to the Orchestrator policy engine. Policies run in the Orchestrator server and monitor objects through plug-ins.

3 Define and Publish Events to Orchestrator on page 44

   The `IPluginEventPublisher` interface allows you to publish to the Orchestrator policy engine events that you define in the plug-in that occur in the plugged-in technology.

Set Up the Event Generator

You can create `IPluginEventPublisher` instances and methods to generate events directly in the plug-in adaptor implementation. However, the solar system class creates these objects in a separate class.

Prerequisites

- Verify that you have an application to plug in to Orchestrator.
- Verify that you have access to the Orchestrator plug-in API JAR file.

Procedure

1 Create and save a Java file for the plug-in event generator class named *ApplicationName*`EventGenerator.java`.

   In the solar system example, the event generator class is named `SolarSystemEventGenerator.java`.

2 Declare the package that contains the Java classes of the plug-in implementation.

   The solar system example declares the following package.

```
package com.vmware.orchestrator.api.sample.solarsystem;
```

3 Import the following Orchestrator plug-in API classes with a Java `import` statement.

```
import ch.dunes.vso.sdk.api.IPluginEventPublisher;
import ch.dunes.vso.sdk.api.IPluginFactory;
```

4 Import any other classes that the event generator requires.

   In the solar system example, the event generator requires the following classes:

```
import java.util.Collections;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import org.apache.log4j.Logger;
```

5 Declare a public class for the event generator implementation.

   The solar system example factory declares the `SolarSystemEventGenerator` class.

```
public class SolarSystemEventGenerator {
}
```

6 Set up logging so that Orchestrator can record in the logs the events that the event generator generates.

   The solar system example uses an instance of `org.apache.log4j.Logger` to log events.

```
public class SolarSystemEventGenerator {
    private static final Logger log = Logger.getLogger(SolarSystemEventGenerator.class);
}
```

7 Create an instance of the event generator class for other classes in the plug-in implementation to use.

   The `SolarSystemEventGenerator` declares a static instance of itself named `_solarSystemEventGenerator` that the plug-in adapter class instantiates when it runs.

```
public class SolarSystemEventGenerator {
    private static final Logger log = Logger.getLogger(SolarSystemEventGenerator.class);
    public final static SolarSystemEventGenerator _solarSystemEventGenerator =
        new SolarSystemEventGenerator();
}
```

You set up and instantiated a plug-in event generator class.

What to do next

Create instances of the `IPluginEventPublisher` interface to monitor objects in the plugged-in technology.

Create Event Publishers

You can create `IPluginEventPublisher` instances to publish event gauges and event triggers to the Orchestrator policy engine. Policies run in the Orchestrator server and monitor objects through plug-ins.

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 `SolarSystemEventGenerator` class creates `IPluginEventPublisher` instances. The `SolarSystemEventGenerator` class defines methods to add and remove the `IPluginEventPublisher` instances in the Orchestrator policy engine.

Prerequisites

Set up the event generator class to create event generator instances.

Procedure

1 Create one or more instances of the `IPluginEventPublisher` interface with which to register the objects to monitor.

   The `SolarSystemEventGenerator` class creates a map to contain all of the `IPluginEventPublisher` instances.

```
private final Map<String, List<IPluginEventPublisher>> policyElements =
Collections.synchronizedMap(new HashMap<String, List<IPluginEventPublisher>>());
```

2 Define a method to register the objects to monitor with the IPluginEventPublisher instances.

   The SolarSystemEventGenerator class defines a method that takes as parameters the type and identifier of the object to monitor and an IPluginEventPublisher instance with which to monitor the object. The addPolicyElement() method adds an IPluginEventPublisher instance for each object to the hashtable of IPluginEventPublisher instances.

```
public void addPolicyElement(String sdkType, String id, IPluginEventPublisher publisher) {
  String key = sdkType + "' / '" + id;
  log.info("Registering element to watch : '" +  key + "'");

  List<IPluginEventPublisher> publishers = policyElements.get(key);
  if (publishers == null) {
    publishers = Collections.synchronizedList(new LinkedList<IPluginEventPublisher>());
    policyElements.put(key, publishers);
  }
  publishers.add(publisher);
}
```

3 (Optional) Define a method to remove objects from the list of objects to monitor.

   The SolarSystemEventGenerator class defines a method that takes as parameters the type and identifier of the object to monitor and an IPluginEventPublisher instance with which the objects are registered. The removePolicyElement() method removes an IPluginEventPublisher instance for the identified object from the hashtable of IPluginEventPublisher instances.

```
public void removePolicyElement(String sdkType, String id, IPluginEventPublisher publisher) {
  String key = sdkType + "' / '" + id;
  log.info("Unregistering element to watch : '" + key + "'");

  List<IPluginEventPublisher> publishers = policyElements.get(key);
  publishers.remove(publisher);
  if (publishers.isEmpty()) {
    policyElements.remove(key);
  }
}
```

You created IPluginEventPublisher instances that publish to Orchestrator policies the events that occur on objects in the plugged-in technology.

What to do next

Define an event to push from Orchestrator to the plugged-in technology.

Define and Publish Events to Orchestrator

The IPluginEventPublisher interface allows you to publish to the Orchestrator policy engine events that you define in the plug-in that occur in the plugged-in technology.

You can use the methods of the IPluginEventPublisher interface to set gauges and triggers that Orchestrator policies monitor.

The SolarSystemEventGenerator class defines a method that generates solar flares of a given magnitude on Star objects in the solar system application. By implementing a method to create solar flares in the solar system plug-in, the plug-in adds a function that does not exist in the solar system application. The generateFlareEvent() method that the plug-in defines registers a policy gauge with the Orchestrator policy engine. An Orchestrator policy can watch this gauge for solar flares that exceed a certain magnitude.

Prerequisites

- Set up the event generator class to create event generator instances.
- Create instances of the `IPluginEventPublisher` interface.

Procedure

1 Create functions that define event listeners and the events that the event listeners monitor.

The `SolarSystemEventGenerator` class declares an interface from which to create listener instances, and adds a method to the interface to create flare events of a certain magnitude on a given `Star` object.

```
public interface StarFlareEventListener{
    void starFlareEvent(String starid, double magnitude);
}
```

2 Create listener instances to listen for the events that the plug-in defines.

The `SolarSystemEventGenerator` instantiates the `StarFlareEventListener` interface to create a listener named `starFlareEventListener`.

```
private StarFlareEventListener starFlareEventListener;
```

3 Create functions to generate events in the plugged-in technology.

The `SolarSystemEventGenerator` defines the `generateFlareEvent()` method that takes an object type, identifier, and a magnitude value as parameters. The method writes in the logs that the plug-in generated a flare event on a given object.

```
public void generateFlareEvent(String sdkType, String id, double magnitude) {
    String key = sdkType + "' / '" + id;
    log.info("Generate Flare Event for : '" + key + "'");
}
```

4 Register the objects to monitor with `IPluginEventPublisher` instances.

The `SolarSystemEventGenerator.generateFlareEvent()` method adds `IPluginEventPublisher` instances for each object to the `policyElements` hashtable of publishers that the `SolarSystemEventGenerator` class creates.

```
public void generateFlareEvent(String sdkType, String id, double magnitude) {
    String key = sdkType + "' / '" + id;
    log.info("Generate Flare Event for : '" + key + "'");
    List<IPluginEventPublisher> publishers = policyElements.get(key);
}
```

5 Call the `IPluginEventPublisher.pushGauge()` or `IPluginEventPublisher.pushTrigger()` methods to publish gauges or triggers to the Orchestrator policy engine.

The `SolarSystemEventGenerator.generateFlareEvent()` method calls the `pushGauge()` method to publish a gauge with the Orchestrator policy engine. The `generateFlareEvent()` method passes the object type, identifier, and magnitude value to the `pushGauge()` method, sets the gauge name to `Flare` and the type of value that the gauge monitors to `magnitude`.

```
public void generateFlareEvent(String sdkType, String id, double magnitude) {
    String key = sdkType + "' / '" + id;
    log.info("Generate Flare Event for : '" + key + "'");
    List<IPluginEventPublisher> publishers = policyElements.get(key);
    if (publishers != null) {
        for (IPluginEventPublisher publisher : publishers) {
            publisher.pushGauge(sdkType, id, "Flare", "magnitude", magnitude);
    }
}
```

6 Call the functions that define the events to generate and monitor.

   To generate flare events, the `SolarSystemEventGenerator.generateFlareEvent()` method calls the `StarFlareEventListener.starFlareEvent()` method that the `SolarSystemEventGenerator` class defines.

```
public void generateFlareEvent(String sdkType, String id, double magnitude) {
    String key = sdkType + "' / '" + id;
    log.info("Generate Flare Event for : '" + key + "'");
    List<IPluginEventPublisher> publishers = policyElements.get(key);
    if (publishers != null) {
        for (IPluginEventPublisher publisher : publishers) {
            publisher.pushGauge(sdkType, id, "Flare", "magnitude", magnitude);
    }
    if (sdkType.equals("Star")) {
        starFlareEventListener.starFlareEvent(id, magnitude);
    }
}
```

You registered a policy gauge or a trigger that an Orchestrator policy can watch for events in the plugged-in technology. If the events occur, the policy starts an operation in the Orchestrator server.

What to do next

Create plug-in triggers to start operations in the Orchestrator server when certain events occur in the plugged-in technology.

Create a Plug-In Workflow Trigger

You can create plug-in workflow triggers to monitor events in the plugged-in technology on behalf of a Wait Event element in a workflow. To create a workflow trigger, you create a Java class that implements the `PluginTrigger` class from the Orchestrator plug-in API.

When a workflow trigger detects a change in the properties of an object that it is monitoring, it sends a notification to any Orchestrator workflows that are waiting for the event. When the workflow receives the notification from the workflow trigger, it stops waiting and resumes its run.

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.

The solar system plug-in features a workflow trigger that creates an instance of the `PluginTrigger` class to monitor solar flare events on a `Star` object in a `SolarSystemRepository` instance. These procedures present the steps involved in creating a workflow trigger. To illustrate the process, they present code from the `SolarSystemTriggerGenerator` class. The package of workflows, actions, and resources that accompany the solar system example provides a workflow that contains a Wait Event element that monitors the workflow trigger for solar flare events.

For a description of the role of the plug-in triggers and the other components of a plug-in, see Chapter 1, "Overview of Plug-Ins," on page 11. For information about all the methods and parameters of the plug-in trigger class, see "PluginTrigger Class," on page 100.

Procedure

1 Set Up the Workflow Trigger on page 47

   To create a workflow trigger, you create an implementation of the `PluginTrigger` class from the Orchestrator plug-in API.

2 Create Instances of the PluginTrigger Class on page 48

   You create a workflow trigger by instantiating the `PluginTrigger` class.

3 Set the Properties that a Workflow Trigger Monitors on page 48

   Workflow triggers monitor changes in the properties of an object in the plugged-in technology. When workflow triggers detect a change in the properties of an object, they notify any workflows in the Orchestrator server that are waiting for this event.

Set Up the Workflow Trigger

To create a workflow trigger, you create an implementation of the `PluginTrigger` class from the Orchestrator plug-in API.

Prerequisites

- Verify that you have an application to plug in to Orchestrator.
- Verify that you have access to the Orchestrator plug-in API JAR file.

Procedure

1. Create and save a Java file for the workflow trigger implementation.

   In the solar system example, the workflow trigger class is named `SolarSystemTriggerGenerator`.

2. Declare the package that contains the Java classes of the plug-in implementation.

   The solar system example declares the following package.

   ```
   package com.vmware.orchestrator.api.sample.solarsystem;
   ```

3. Import the Orchestrator plug-in API classes with a Java `import` statement.

   The `SolarSystemTriggerGenerator` class requires the following classes:

   ```
   import ch.dunes.vso.sdk.api.IPluginFactory;
   import ch.dunes.vso.sdk.api.PluginTrigger;
   ```

4. Import the classes of the application to plug in with a Java `import` statement.

   The `SolarSystemTriggerGenerator` class requires the following class:

   ```
   import com.vmware.solarsystem.Star;
   ```

5. Import any other classes that the workflow trigger implementation requires.

   The `SolarSystemTriggerGenerator` class requires the following class:

   ```
   import java.util.Properties;
   ```

6. Declare a public class to contain the workflow trigger implementation.

   The `SolarSystemTriggerGenerator` class declares the following class:

   ```
   public class SolarSystemTriggerGenerator {
   }
   ```

You set up the workflow trigger implementation.

What to do next

Implement the `PluginTrigger` class from the Orchestrator plug-in API to create workflow trigger instances.

Create Instances of the PluginTrigger Class

You create a workflow trigger by instantiating the `PluginTrigger` class.

The `PluginTrigger` class defines a constructor that you can use to create workflow trigger instances. The `PluginTrigger` constructor can define properties such as the name of the workflow trigger, a timeout period, and the type and identifier of the object that the workflow trigger monitors.

Prerequisites

- Set up the workflow trigger implementation class.
- Declare a public class to contain the workflow trigger implementation.

Procedure

1. Create an instance of the `PluginTrigger` class by calling the `PluginTrigger()` constructor.

   The `SolarSystemTriggerGenerator` class defines the `newTrigger()` method to create a workflow trigger named `trigger`.

   ```
   private PluginTrigger newTrigger()  {
       PluginTrigger trigger = new PluginTrigger();
       return trigger;
   }
   ```

2. Call the methods of the `PluginTrigger` class to set the basic properties of the workflow trigger.

   The `SolarSystemTriggerGenerator` class calls the `PluginTrigger.setModuleName()` method to set the name of the workflow trigger to the same name as the plug-in itself and calls `PluginTrigger.setTimeout()` to deactivate the timeout period.

   ```
   private PluginTrigger newTrigger()  {
       PluginTrigger trigger = new PluginTrigger();
       trigger.setModuleName(SolarSystemAdapter.pluginName);
       trigger.setTimeout(-1);
       return trigger;
   }
   ```

   NOTE If you find objects by their type or identifier, you implement the `setSdkType()` and `setSdkId()` methods to set triggers on objects.

You used the `PluginTrigger()` constructor to create workflow trigger instances to notify waiting workflows when defined events occur.

What to do next

Set the properties that the workflow trigger monitors in the objects in the plugged-in technology.

Set the Properties that a Workflow Trigger Monitors

Workflow triggers monitor changes in the properties of an object in the plugged-in technology. When workflow triggers detect a change in the properties of an object, they notify any workflows in the Orchestrator server that are waiting for this event.

You set the properties that a workflow trigger monitors by passing a `java.util.Properties` list to a `PluginTrigger` instance.

Prerequisites

- Set up the workflow trigger class.

- Create `PluginTrigger` instances.

Procedure

1. Declare variables for the object and object properties that the workflow trigger monitors.

   The `SolarSystemTriggerGenerator` class declares variables for the `Star` object that it monitors and for the magnitude of any solar flare events that occur on that star.

   ```
   public static final String STAR_ID = "star_id";
   public static final String MAGNITUDE = "magnitude";
   ```

2. Create an instance of the `PluginTrigger` class in which to set the properties to monitor.

   The `SolarSystemTriggerGenerator` class calls the `SolarSystemTriggerGenerator.newTrigger()` method to create a trigger instance.

   ```
   public PluginTrigger createStarFlareTrigger(Star star, double magnitude) {
       PluginTrigger trigger = newTrigger();
       return trigger;
   }
   ```

3. Create an instance of a `java.util.Properties` list to contain the properties to monitor.

   The `SolarSystemTriggerGenerator` class create a properties list named `props`.

   ```
   public PluginTrigger createStarFlareTrigger(Star star, double magnitude){
       PluginTrigger trigger = newTrigger();
       Properties props = new Properties();
       return trigger;
   }
   ```

4. Call the `Properties.setProperty()` method to add the properties to monitor to the properties list.

   The `SolarSystemTriggerGenerator` class adds the identifier of the star object and the value of the magnitude of the solar flare event to the properties list `props`.

   ```
   public PluginTrigger createStarFlareTrigger(Star star, double magnitude){
       PluginTrigger trigger = newTrigger();
       Properties props = new Properties();
       props.setProperty(STAR_ID, star.getId());
       props.setProperty(MAGNITUDE, Double.toString(magnitude));
       return trigger;
   }
   ```

5. Call the `PluginTrigger.setProperties()` method to add the properties list to the workflow trigger instance.

   The `SolarSystemTriggerGenerator` class adds the properties list `props` to the workflow trigger, to provide the identifier of the star object and the value of the flare event to monitor.

   ```
   public PluginTrigger createStarFlareTrigger(Star star, double magnitude){
       PluginTrigger trigger = newTrigger();
       Properties props = new Properties();
       props.setProperty(STAR_ID, star.getId() );
       props.setProperty(MAGNITUDE, Double.toString(magnitude));
       trigger.setProperties(props);
       return trigger;
   }
   ```

You added a list of properties to a workflow trigger so that it can monitor the value of a given property in an object. If the properties of the object change, the workflow trigger notifies any workflows that are waiting for that event.

What to do next

Create plug-in watchers to watch the triggers for events.

Create Plug-In Watchers

Plug-in watchers watch triggers on behalf of workflows that are waiting for the event that the trigger starts. To create a plug-in watcher, you create a Java class that implements the `PluginWatcher` class from the Orchestrator plug-in API. You publish the watcher on the Orchestrator notification server by implementing the `IPluginPublisher` interface.

These procedures present the steps involved in creating a plug-in watcher. To illustrate the process, they present code from the `SolarSystemWatchersManager` class from the solar system plug-in.

You can download the vCO Plug-in SDK ZIP file from the VMware Communities site to obtain the sources of the solar system example application and plug-in.

For a description of the role of plug-in event watchers and the other components of a plug-in, see Chapter 1, "Overview of Plug-Ins," on page 11. For information about all of the methods and parameters of the plug-in watcher class and publisher interface, see "PluginWatcher Class," on page 101 and "IPluginPublisher Interface," on page 98.

Procedure

1 Set Up the Watcher Implementation on page 50

   You can create `PluginWatcher` instances and methods to generate events directly in the plug-in adaptor. However, the solar system example creates the watcher instances in a separate class named `SolarSystemWatchersManager`.

2 Create Instances of the PluginWatcher Class on page 52

   You create a plug-in watcher to watch a workflow trigger by instantiating the `PluginWatcher` class. When the event that the workflow trigger defines occurs, the plug-in watcher notifies any workflows that are waiting for that event.

3 Publish Plug-In Watchers on page 52

   You implement the `IPluginPublisher` interface to publish plug-in watchers to the Orchestrator notification mechanism.

Set Up the Watcher Implementation

You can create `PluginWatcher` instances and methods to generate events directly in the plug-in adaptor. However, the solar system example creates the watcher instances in a separate class named `SolarSystemWatchersManager`.

Prerequisites

- Download the bundle of Orchestrator examples.
- Unzip the examples bundle to an appropriate location.

Procedure

1 Create and save a Java file for the plug-in watcher class.

   In the solar system example, the watcher class is named `SolarSystemWatchersManager.java`.

2 Declare the package that contains the Java classes of the plug-in implementation.

   The solar system example declares the following package.

```
package com.vmware.orchestrator.api.sample.solarsystem;
```

3 Import the Orchestrator plug-in API classes with a Java import statement.

   In the solar system example, the event watcher requires the following classes:

```
import ch.dunes.vso.sdk.api.IPluginPublisher;
import ch.dunes.vso.sdk.api.PluginWatcher;
```

4 Import any classes that the plug-in implementation or plugged-in technology defines.

   In the solar system example, the watcher requires the following class that the `SolarSystemEventGenerator` class defines:

```
import com.vmware.orchestrator.api.sample.solarsystem.
 SolarSystemEventGenerator.StarFlareEventListener;
```

5 Import any other classes that the watcher implementation requires.

   In the solar system example, the event watcher requires the following classes:

```
import java.util.Collections;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import org.apache.log4j.Logger;
```

6 Declare a public class for the event generator implementation.

   The solar system example watcher declares the `SolarSystemWatchersManager` class, that implements the `StarFlareEventListener` class.

```
public class SolarSystemWatchersManager implements StarFlareEventListener {
}
```

   The `StarFlareEventListener` class listens for solar flare events of a certain magnitude.

7 Set up logging so that Orchestrator can record in the logs the events that the watcher observes.

   The solar system example uses an instance of `org.apache.log4j.Logger` to log events.

```
public class SolarSystemWatchersManager implements StarFlareEventListener {
    private static final Logger log = Logger.getLogger(SolarSystemWatchersManager.class);
}
```

8 Declare a public constructor to create instances of the watcher implementation class.

   The `SolarSystemWatchersManager` class creates a constructor that creates instances of `SolarSystemWatchersManager`. The `SolarSystemWatchersManager` instances contain the `starFlareEventListener` event listener that the `SolarSystemEventGenerator` class defines.

```
public SolarSystemWatchersManager() {

SolarSystemEventGenerator.solarSystemEventGenerator.addStarFlareUniqueEventListener(this);
}
```

You set up a plug-in event watcher class to create watcher instances to watch for events from triggers.

What to do next

Create instances of the `PluginWatcher` class.

Create Instances of the PluginWatcher Class

You create a plug-in watcher to watch a workflow trigger by instantiating the `PluginWatcher` class. When the event that the workflow trigger defines occurs, the plug-in watcher notifies any workflows that are waiting for that event.

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.

Prerequisites

- Set up the plug-in watcher implementation class.
- Declare a public constructor to instantiate the plug-in watcher implementation.

Procedure

1. Create one or more instances of the `PluginWatcher` class with which to register the workflow triggers to monitor.

   The `SolarSystemWatchersManager` class creates a hashtable to contain all of the `PluginWatcher` instances.

   ```
   private final Map<String, PluginWatcher> watchers = Collections.synchronizedMap(new
   HashMap<String, PluginWatcher>());
   ```

2. Obtain watcher instances by calling the `PluginWatcher.getId()` method.

   The `SolarSystemWatchersManager` class defines a method that adds a `PluginWatcher` instance to the hashtable of `PluginWatcher` instances.

   ```
   public void addWatcher(PluginWatcher watcher) {
       watchers.put(watcher.getId(), watcher);
   }
   ```

3. (Optional) Remove watcher instances after an event occurs.

   The `SolarSystemWatchersManager` class defines a method that removes a `PluginWatcher` instance from the hashtable of `PluginWatcher` instances.

   ```
   public void removeWatcher(String watcherId) {
       watchers.remove(watcherId);
   }
   ```

You created instances of the `PluginWatcher` class to watch workflow triggers for events.

What to do next

Register the watcher instances with `IPluginPublisher` instances to publish the watchers to the Orchestrator notification mechanism.

Publish Plug-In Watchers

You implement the `IPluginPublisher` interface to publish plug-in watchers to the Orchestrator notification mechanism.

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.

Prerequisites

- Set up the plug-in watcher implementation class.

- Declare a public constructor to instantiate the plug-in watcher implementation.
- Create instances of the `PluginWatcher` class.

Procedure

1. Create an instance of the `IPluginPublisher` interface.

   The `SolarSystemWatchersManager` class declares the following variable for the `IPluginPublisher` instance.

   ```
   private IPluginPublisher pluginPublisher;
   ```

2. Define a method to add the `IPluginPublisher` instance to the plug-in adapter implementation.

   The `SolarSystemWatchersManager` class declares the following method that the `SolarSystemAdapter` class calls.

   ```
   public void setPluginPublisher(IPluginPublisher pluginPublisher) {
       this.pluginPublisher = pluginPublisher;
   }
   ```

3. Define the event for which the watcher watches and that the publisher publishes to the Orchestrator notification mechanism.

   The `SolarSystemWatchersManager` class defines the `starFlareEvent()` method that takes a `Star` object and a `magnitude` value as parameters. The `starFlareEvent()` method also gets the hashtable of watchers and creates a list of watchers to remove from the hashtable after the event occurs.

   ```
   public void starFlareEvent(String starid, double magnitude) {
       synchronized (watchers) {
           List<String> watchersToRemove = new LinkedList<String>();
       }
   }
   ```

4. Call the `PluginWatcher.getTrigger()` and `PluginTrigger.getProperties()` methods to obtain the properties to watch in the trigger.

   The `SolarSystemWatchersManager.starFlareEvent()` method extracts the `STAR_ID` and `MAGNITUDE` properties from the trigger and adds them to a `PluginWatcher` instance in the hashtable.

   ```
   public void starFlareEvent(String starid, double magnitude) {
       synchronized (watchers) {
           List<String> watchersToRemove = new LinkedList<String>();
           for (PluginWatcher watcher : watchers.values()) {
               Properties props = watcher.getTrigger().getProperties();
               String wStarId = props.getProperty(SolarSystemTriggerGenerator.STAR_ID);
               String wMagnitude = props.getProperty(SolarSystemTriggerGenerator.MAGNITUDE);
           }
       }
   }
   ```

5 Define the event and publish it to the Orchestrator notification mechanism by calling the `IPluginPublisher.pushWatcherEvent()` method.

   The `SolarSystemWatchersManager.starFlareEvent()` method checks whether the magnitude value exceeds a maximum magnitude value. If the maximum magnitude value is exceeded, the `starFlareEvent()` method writes the flare event in the logs and publishes the event to the Orchestrator notification mechanism.

```
public void starFlareEvent(String starid, double magnitude) {
    synchronized (watchers) {
        List<String> watchersToRemove = new LinkedList<String>();
        for (PluginWatcher watcher : watchers.values()) {
            Properties props = watcher.getTrigger().getProperties();
            String wStarId = props.getProperty(SolarSystemTriggerGenerator.STAR_ID);
            String wMagnitude = props.getProperty(SolarSystemTriggerGenerator.MAGNITUDE);
            if (wStarId != null && wStarId.equals(starid)) {
                double wMagnLimit = Double.parseDouble(wMagnitude);
                if (magnitude >= wMagnLimit) {
                    log.info("pushWatcherEvent() for id '" + watcher.getId() + "'");
                    pluginPublisher.pushWatcherEvent(watcher.getId(), null);
                }
            }
        }
    }
}
```

6 (Optional) Remove the watchers from the notification mechanism after the events occur.

   The `SolarSystemWatchersManager.starFlareEvent()` method adds the watcher to the list of watchers to remove and defines a method to remove that list of watchers from the hashtable.

```
public void starFlareEvent(String starid, double magnitude) {
    synchronized (watchers) {
        List<String> watchersToRemove = new LinkedList<String>();
        for (PluginWatcher watcher : watchers.values()) {
            Properties props = watcher.getTrigger().getProperties();
            String wStarId = props.getProperty(SolarSystemTriggerGenerator.STAR_ID);
            String wMagnitude = props.getProperty(SolarSystemTriggerGenerator.MAGNITUDE);
            if (wStarId != null && wStarId.equals(starid)) {
                double wMagnLimit = Double.parseDouble(wMagnitude);
                if (magnitude >= wMagnLimit) {
                    log.info("pushWatcherEvent() for id '" + watcher.getId() + "'");
                    pluginPublisher.pushWatcherEvent(watcher.getId(), null);
                    watchersToRemove.add(watcher.getId());
                }
            }
        }
        for (String toRemove : watchersToRemove) {
            watchers.remove(toRemove);
        }
    }
}
```

You defined an event for a watcher to watch and published the event to the Orchestrator notification mechanism. Orchestrator notifies any workflows that are waiting for that event that it has occurred.

What to do next

Define objects and methods to add to the Orchestrator JavaScript API.

Define Objects and Methods to Map to the Orchestrator JavaScript API

You can map the object types, classes, and methods of the plugged-in technology and the plug-in itself to JavaScript types, classes, and methods that you add to the Orchestrator JavaScript API.

You can add objects and functions to the JavaScript API that do not exist in the plugged-in technology by defining them in the plug-in implementation. Adding objects and functions to the JavaScript API allows you to include the objects and functions in Orchestrator actions and workflows, to perform operations on objects in the plugged-in technology.

You map the objects and functions that you define in the plug-in implementation in `<scripting-object>` elements in the `vso.xml` file. For information about how to map objects and functions to the JavaScript API in the `vso.xml` file, see "Map the Application in the vso.xml File," on page 70.

The `vso.xml` file in the solar system example maps to JavaScript objects all of the objects and methods that the solar system application defines. The `vso.xml` file also maps the following objects and methods from the solar system plug-in implementation to the Orchestrator JavaScript API.

- `SolarSystemEventGenerator` scripting class
- `SolarSystemEventGenerator.generateFlareEvent()` scripting method
- `SolarSystemTriggerGenerator` scripting class
- `SolarSystemTriggerGenerator.createStarFlareTrigger()` scripting method

To create a class to map to the Orchestrator JavaScript API, you add an instance of that class to an instance of the `IPluginFactory` implementation by defining a method named `createScriptingSingleton()`. When the plug-in adaptor instantiates the factory, it also instantiates the class to add to the JavaScript API.

Prerequisites

You created at least one class of the plug-in implementation, for example the adaptor, the factory, or an event generator implementation.

Procedure

1. Create an instance of a class to map to the Orchestrator JavaScript API in one of the classes of the plug-in implementation.

   The `SolarSystemEventGenerator` class defines the following constructor to create instances of itself:

   ```
   public final static SolarSystemEventGenerator solarSystemEventGenerator =
      new SolarSystemEventGenerator();
   ```

2. Define a method named `createScriptingSingleton()` that accesses the `IPluginFactory` implementation of the plug-in.

   The `SolarSystemEventGenerator` class creates the following instance of that class:

   ```
   public static SolarSystemEventGenerator createScriptingSingleton(IPluginFactory factory) {
   }
   ```

3 Implement the `createScriptingSingleton()` method to return to the factory an instance of the class to map to the JavaScript API.

   The `SolarSystemEventGenerator` class returns the `_solarSystemEventGenerator` instance.

```
public static SolarSystemEventGenerator createScriptingSingleton(IPluginFactory factory) {
   return solarSystemEventGenerator;
}
```

You created an instance of a class that the `vso.xml` file can map to a scripting class in the Orchestrator JavaScript API. The `vso.xml` file of the solar system example maps the `_solarSystemEventGenerator` instance to the `SolarSystemEventGenerator` scripting class in the JavaScript API.

What to do next

Implement the plug-in adapter to instantiate all of the classes and objects that you defined in the plug-in.

Create a Plug-In Adapter

To create a plug-in adapter, you create a Java class that implements the `IPluginAdaptor` interface from the Orchestrator plug-in API. The adapter instantiates the plug-in factory and event management implementations.

These procedures present the steps involved in creating a plug-in adapter. To illustrate the process, they present code from the `SolarSystemAdapter` class from the solar system example application. You can download the vCO Plug-in SDK ZIP file from the VMware Communities site to obtain the sources of the solar system example application.

For a description of the role of the plug-in adapter and the other components of a plug-in, see Chapter 1, "Overview of Plug-Ins," on page 11. For information about all of the methods and parameters of the adapter interface, see "IPluginAdaptor Interface," on page 95.

Procedure

1 Set Up the Plug-In Adapter Implementation on page 57

   To create a plug-in adapter, you create an implementation of the `IPluginAdaptor` interface from the Orchestrator plug-in API.

2 Instantiate the Plug-In Factory on page 57

   You instantiate the plug-in factory in the plug-in adapter. The adapter creates one factory instance for every connection between Orchestrator and the plugged-in technology.

3 Manage Plug-In Events on page 59

   The plug-in adapter manages the events that occur in the plugged-in technology by defining event generators and event publishers.

4 Add Plug-In Watchers on page 60

   Plug-in watchers monitor the events that workflow triggers define. When an event occurs, Orchestrator notifies any workflows that are waiting for that event.

Set Up the Plug-In Adapter Implementation

To create a plug-in adapter, you create an implementation of the `IPluginAdaptor` interface from the Orchestrator plug-in API.

Procedure

1 Create and save a Java file for the plug-in adapter implementation named *ApplicationName*`Adapter.java`.

In the solar system example, the adapter class is `SolarSystemAdapter.java`.

2 Declare a package to contain the plug-in implementation.

The solar system example declares the following package to contain the adapter, factory, and event handler implementations:

```
package com.vmware.orchestrator.api.sample.solarsystem;
```

3 Import the following Orchestrator plug-in API interfaces, classes, and enumerations with a Java `import` statement.

```
import ch.dunes.vso.sdk.api.IPluginAdaptor;
import ch.dunes.vso.sdk.api.IPluginEventPublisher;
import ch.dunes.vso.sdk.api.IPluginFactory;
import ch.dunes.vso.sdk.api.IPluginNotificationHandler;
import ch.dunes.vso.sdk.api.IPluginPublisher;
import ch.dunes.vso.sdk.api.PluginLicense;
import ch.dunes.vso.sdk.api.PluginLicenseException;
import ch.dunes.vso.sdk.api.PluginWatcher;
```

4 Import any other classes that the adapter implementation requires.

In the solar system example, the adapter implementation requires the following classes:

```
import javax.security.auth.login.LoginException;
import org.jboss.logging.Logger;
```

5 Declare a public constructor that implements the `IPluginAdaptor` interface from the Orchestrator plug-in API.

The solar system adapter declares the `SolarSystemAdapter` constructor.

```
public class SolarSystemAdapter implements IPluginAdaptor {
}
```

6 Set up a logger to write to the logs the events that occur in the adapter.

The solar system adapter uses an instance of `org.jboss.logging.Logger` to log events.

```
private static final Logger log = Logger.getLogger(SolarSystemAdapter.class);
```

What to do next

Create an instance of the `IPluginFactory` implementation.

Instantiate the Plug-In Factory

You instantiate the plug-in factory in the plug-in adapter. The adapter creates one factory instance for every connection between Orchestrator and the plugged-in technology.

Prerequisites

- Create an implementation of the `IPluginFactory` interface.

- Set up the adapter implementation class.
- Create a public constructor that implements the `IPluginAdaptor` interface.

Procedure

1 Declare the variables that the adapter class uses in its method calls.

The solar system adapter declares variables for the plug-in factory and plug-in name.

```
public class SolarSystemAdapter implements IPluginAdaptor {
    private SolarSystemFactory factory;
    static String pluginName;
}
```

2 Create an instance of the plug-in factory class that implements the `IPluginFactory` interface.

The solar system adapter calls the `IPluginAdaptor.createPluginFactory()` method to create an instance of the `SolarSystemFactory` interface, if one does not exist already.

```
public IPluginFactory createPluginFactory(String sessionID, String username,
    String password, IPluginNotificationHandler notificationHandler)
    throws SecurityException, LoginException, PluginLicenseException {
        if (factory == null) {
            factory = new SolarSystemFactory();
        }
    return factory;
}
```

3 Set the plug-in name.

The `IPluginAdaptor.setPluginName()` method gets the name from the `vso.xml` file.

The solar system adapter uses the `pluginName` variable to set the name of the plug-in.

```
public void setPluginName(String pluginName) {
    SolarSystemAdapter.pluginName = pluginName;
}
```

4 (Optional) Install any licenses that Orchestrator requires to access the plugged-in technology.

You obtain licenses by calling the `IPluginAdaptor.installLicenses()` method to instantiate an array of `PluginLicense` objects.

```
public void installLicenses(PluginLicense[] licenses) throws
    PluginLicenseException {
}
```

5 (Optional) Uninstall an existing plug-in factory.

A plug-in creates a factory instance for every client session that opens between Orchestrator and a plugged-in technology. You can remove unnecessary plug-in factories to clean up the Orchestrator server by calling the `IPluginAdaptor.uninstallPluginFactory()` method.

The solar system plug-in does not implement the `IPluginAdaptor.uninstallPluginFactory()` method. You can uninstall factories by implementing a function in the `uninstallPluginFactory()` method declaration.

```
public void uninstallPluginFactory(IPluginFactory plugin) {
}
```

You instantiated the `IPluginFactory` implementation, set the name for the plug-in, obtained any licenses that the plug-in connection requires, and potentially defined a function to remove old factory instances from the server.

What to do next

Instantiate event generators and publishers.

Manage Plug-In Events

The plug-in adapter manages the events that occur in the plugged-in technology by defining event generators and event publishers.

Prerequisites

- Create a public constructor that implements the `IPluginAdaptor` interface.
- Instantiate the plug-in factory.

Procedure

1. Define the method to generate plug-in events.

   You can define the events to manage directly in the adapter implementation. However, the solar system plug-in implementation defines the events in a separate class, `SolarSystemEventGenerator`.

   `SolarSystemAdapter` defines the following `getEventGenerator()` method to obtain an instance of the `SolarSystemEventGenerator` class.

   ```
   private SolarSystemEventGenerator getEventGenerator() {
       return SolarSystemEventGenerator.solarSystemEventGenerator;
   }
   ```

2. (Optional) If Orchestrator monitors the plugged-in application for events, you can register an instance of the `IPluginEventPublisher` interface with the Orchestrator policy engine by calling the `IPluginAdaptor.registerEventPublisher()` method.

   The solar system example adapter creates the following `IPluginEventPublisher` instance and registers it with the Orchestrator policy engine by calling the `SolarSystemEventGenerator.addPolicyElement()` method.

   ```
   public void registerEventPublisher(
       String type, String id, IPluginEventPublisher publisher) {
           getEventGenerator().addPolicyElement(type, id, publisher);
   }
   ```

3. (Optional) Unregister an `IPluginEventPublisher` from the Orchestrator policy engine.

   The solar system example adapter unregisters an `IPluginEventPublisher` instance and removes it from the Orchestrator policy engine by calling the `SolarSystemEventGenerator.removePolicyElement()` method.

   ```
   public void unregisterEventPublisher(
       String type, String id, IPluginEventPublisher publisher) {
           getEventGenerator().removePolicyElement(type, id, publisher);
   }
   ```

You instantiated an event generator and optionally defined methods to register and unregister an event publisher with the Orchestrator policy engine.

What to do next

Add and remove watchers that monitor workflow triggers. When the events that the workflow triggers define occur, Orchestrator notifies any workflows that are waiting for that event.

Add Plug-In Watchers

Plug-in watchers monitor the events that workflow triggers define. When an event occurs, Orchestrator notifies any workflows that are waiting for that event.

Prerequisites

- Create a public constructor that implements the `IPluginAdaptor` interface.
- Instantiate the plug-in factory.
- Create event generators and publishers.

Procedure

1 Create an instance of the plug-in watcher implementation.

   The `SolarSystemAdapter` class creates an instance of the `SolarSystemWatchersManager` class.

```
private static final SolarSystemWatchersManager watchersManager =
 new SolarSystemWatchersManager();
```

2 Add a watcher instance to the plug-in adaptor.

   You add a watcher to the adapter by calling the `IPluginAdaptor.addWatcher()` method.

   The solar system example defines a method to add the `SolarSystemWatchersManager` instance.

```
public void addWatcher(PluginWatcher watcher) {
    log.info( "Adding watcher '" + watcher + "'" );
        watchersManager.addWatcher(watcher);
}
```

3 Remove a watcher from the plug-in adaptor.

   You remove a watcher by calling the `IPluginAdaptor.removeWatcher()` method.

   The solar system example defines a method to remove a `SolarSystemWatchersManager` instance.

```
public void removeWatcher(String watcherId) {
    log.info( "Removing watcher '" + watcherId + "'" );
        watchersManager.removeWatcher(watcherId);
}
```

4 Instantiate the plug-in publisher that publishes events from the watchers to the Orchestrator notification mechanism.

   The solar system example defines a method that calls the `SolarSystemWatchersManager.setPluginPublisher()` method to instantiate a plug-in publisher.

```
public void setPluginPublisher(IPluginPublisher pluginPublisher) {
    watchersManager.setPluginPublisher(pluginPublisher);
}
```

You added watchers to the plug-in adapter implementation to monitor the events that workflow triggers generate.

What to do next

Add a tab for the plug-in to the Orchestrator configuration interface.

Add a Tab to the Configuration Interface

You can add a tab to the Orchestrator configuration interface to allow users to provide information to the plug-in configuration that is specific to their environment or preferences.

To add a configuration tab for a plug-in to the configuration interface, you implement the `IConfigurationAdaptor` interface. You can also use the `SDKHelper` and extend the `BaseAction` classes from the Orchestrator plug-in API. You create a configuration adapter that accesses the classes of the plug-in and the plugged-in technology for the Orchestrator configuration server. You define configuration actions to obtain and save the configuration information that the user provides by using the configuration tab.

You must create an Apache Struts-based Web application to create the layout of the tab in the configuration interface. The Struts Web application uses the methods that you define in the `IConfigurationAdaptor` implementation to add configuration operations for the users to perform in the configuration tab for the plug-in. The Struts Web application submits to the Orchestrator server the information that the user enters in the configuration tab.

Creating a plug-in configuration tab requires several steps. Code from the `SolarSystemConfigurationAdapter` and `SolarSystemConfigureAction` classes from the solar system plug-in is included in the steps.

You can download the vCO Plug-in SDK ZIP file from the VMware Communities site to obtain the sources of the solar system example application and plug-in.

For information about all of the methods and parameters of the configuration adapter interface, see "IConfigurationAdaptor Interface," on page 94. For information about the additional methods that the `SDKHelper` class provides, see "SDKHelper Class," on page 103.

Note It is a good practice to provide the configuration capabilities that are available in the Orchestrator configuration interface through workflows, which users can run in the Orchestrator client. For information about developing workflows, see *Developing with VMware vCenter Orchestrator*.

Procedure

1 Set Up the Configuration Adapter on page 62

   To create a tab in the configuration interface for a plug-in, you create an implementation of the `IConfigurationAdaptor` interface from the Orchestrator plug-in API. You also call the methods of the `SDKHelper` class.

2 Load and Save Configuration Information in the Configuration Server on page 63

   The `IConfigurationAdaptor` interface provides methods to load and save configuration information in the Orchestrator configuration server. The configuration adapter uses these methods to locate and update the configuration information for a plug-in by setting plug-in properties.

3 Create a Configuration Action to Obtain Configuration Information from the User on page 65

   Orchestrator uses the Apache Struts framework to pass to the Orchestrator server the configuration information that the user provides in the configuration interface.

4 Create a Struts-Based Web Application to Add to the Configuration Interface on page 67

   The tab that you add to the Orchestrator configuration interface is an Apache Struts-based Web application. You define the layout of the page by using HTML or JavaServer Pages (JSP) and add actions to the page by implementing the Struts framework.

Set Up the Configuration Adapter

To create a tab in the configuration interface for a plug-in, you create an implementation of the `IConfigurationAdaptor` interface from the Orchestrator plug-in API. You also call the methods of the `SDKHelper` class.

Prerequisites

- Verify that you have an application to plug in to Orchestrator.
- Verify that you have access to the Orchestrator plug-in API JAR file.
- Implement the plug-in adapter and factory interfaces.

Procedure

1 Create and save a Java file for the plug-in configuration adapter implementation.

In the solar system example, the configuration adapter class is named `SolarSystemConfigurationAdaptor.java`.

2 Declare the package that contains the Java classes of the plug-in configuration implementation.

The solar system example declares the following package.

```
com.vmware.orchestrator.api.sample.solarsystem.config;
```

3 Import the following classes of the plug-in configuration API with a Java import statement.

```
import ch.dunes.vso.sdk.conf.ConfigurationError;
import ch.dunes.vso.sdk.conf.IConfigurationAdaptor;
import ch.dunes.vso.sdk.helper.SDKHelper;
```

4 Import the following classes of the application to plug in with a Java `import` statement.

```
import com.vmware.solarsystem.Planet;
import com.vmware.solarsystem.SolarSystemRepository;
```

5 Import any other classes that the configuration adapter implementation requires.

In the solar system example, the configuration adapter implementation requires the following classes:

```
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.PropertyResourceBundle;
import java.util.ResourceBundle;
```

6 Declare a public class that implements the `IConfigurationAdaptor` interface.

```
public class SolarSystemConfigurationAdaptor implements IConfigurationAdaptor {
}
```

You set up the plug-in configuration adapter implementation.

What to do next

Define methods to load and save plug-in configuration information.

Load and Save Configuration Information in the Configuration Server

The `IConfigurationAdaptor` interface provides methods to load and save configuration information in the Orchestrator configuration server. The configuration adapter uses these methods to locate and update the configuration information for a plug-in by setting plug-in properties.

You also create methods in the configuration adapter to define the values that the user can configure. Orchestrator reloads this information each time this user connects to the Orchestrator server. You can validate the information that the user provides.

In the solar system example, the `SolarSystemConfigurationAdaptor` class defines methods to allow users to select their home planet and to define Pluto as either a planet or as a dwarf planet. The `SolarSystemConfigurationAdaptor` class implements the methods of the `IConfigurationAdaptor` interface to load and save configuration information and to validate the information that the users provide. The methods that `SolarSystemConfigurationAdaptor` defines are implemented by the `SolarSystemConfigureAction` class.

Prerequisites

- Implement the plug-in adapter and factory interfaces.
- Set up the configuration adapter implementation class.

Procedure

1 Declare variables for the plug-in name and for the configurable values.

   The `SolarSystemConfigurationAdaptor` class declares the following variables:

```
private static final String pluginName = "SolarSystem";
private String homePlanet = "";
private String plutoClassifiedAsAPlanet = "yes";
```

2 Create a class loader to load the configuration adapter classes into the configuration server.

   You can use the standard Java class `ResourceBundle` to locate packages of classes.

```
private static final ResourceBundle bundle =
PropertyResourceBundle.getBundle("com.vmware.orchestrator.api.sample.solarsystem.config.packa
ge");
```

3 Define methods to obtain and set the configurable values.

   The `SolarSystemConfigurationAdaptor` class uses the `bundle.getString()` method to get the properties of Pluto from the plug-in configuration server and adds them to a map. The `SolarSystemConfigurationAdaptor` class also defines methods to set whether Pluto is a planet and to set the user's home planet.

```
public Map<String, String> getPlutoClassifyList(){
    Map<String, String> classification = new HashMap<String, String>();
    classification.put("no", bundle.getString("select.pluto.classify.dwarfPlanet"));
    classification.put("yes", bundle.getString("select.pluto.classify.planet"));
    return classification;
}
public List<Planet> getAllPlanets(){
    return SolarSystemRepository.getUniqueInstance().getAllPlanets();
}
public String getHomePlanet() {
    return homePlanet;
}
public void setHomePlanet(String homePlanet) {
```

```
    this.homePlanet = homePlanet;
}
public void setPlutoClassifiedAsAPlanet(String plutoClassifiedAsAPlanet) {
    this.plutoClassifiedAsAPlanet = plutoClassifiedAsAPlanet;
}
public String getPlutoClassifiedAsAPlanet() {
    return plutoClassifiedAsAPlanet;
}
```

4 Implement the `IConfigurationAdaptor.saveConfiguration()` method to save configuration information to the configuration server by setting plug-in properties.

   The `SolarSystemConfigurationAdaptor.loadConfiguration()` method creates a `Properties` list to contain the configurable properties. The values that `SolarSystemConfigurationAdaptor.saveConfiguration()` method adds to the list are the values that the methods defined in Step 3 set. The `SolarSystemConfigurationAdaptor.saveConfiguration()` method calls the `SDKHelper.savePropertiesForPluginName()` to save the `Properties` list to the configuration server.

```
public void saveConfiguration(OutputStream stream) throws IOException {
    synchronized (SDKHelper.class) {
        Properties prop = new Properties();
        prop.setProperty("solar.system.home.planet", homePlanet);
        prop.setProperty("solar.system.isPlutoClassifiedAsAPlanet",
plutoClassifiedAsAPlanet);
        if (stream == null) {
            SDKHelper.savePropertiesForPluginName(prop, pluginName);
        }
    }
}
```

5 Implement the `IConfigurationAdaptor.loadConfiguration()` method to load configuration information from the configuration server into the Orchestrator server.

   The `SolarSystemConfigurationAdaptor.loadConfiguration()` method creates a `Properties` list to contain the configurable properties. The `SolarSystemConfigurationAdaptor.loadConfiguration()` method calls the `SDKHelper.getConfigurationPathForPluginName()` and `SDKHelper.loadPropertiesForPluginName()` methods to get the properties from the plug-in and adds them to the `Properties` list.

```
public void loadConfiguration(InputStream stream) throws IOException {
    synchronized (SDKHelper.class) {
        String path = SDKHelper.getConfigurationPathForPluginName(pluginName);
        if (new File(path).exists()) {
            Properties prop = SDKHelper.loadPropertiesForPluginName(pluginName);
            homePlanet = prop.getProperty("solar.system.home.planet", homePlanet);
            plutoClassifiedAsAPlanet =
prop.getProperty("solar.system.isPlutoClassifiedAsAPlanet", plutoClassifiedAsAPlanet);
        }
    }
}
```

6 Implement the `IConfigurationAdaptor.setPluginName()` method to set the name of the plug-in in the configuration server.

   The `SolarSystemConfigurationAdaptor` class does not add any additional code to the `IConfigurationAdaptor.setPluginName()` method.

```
public void setPluginName(String name) {
}
```

7 Implement the `IConfigurationAdaptor.validateConfiguration()` method to validate the configuration information.

   The `IConfigurationAdaptor.validateConfiguration()` method returns a `ConfigurationError` instance for each validation error. The `SolarSystemConfigurationAdaptor` class returns null in the event of an invalid configuration property. You can implement more sophisticated code to perform more stringent validation of the values that the user provides in the configuration interface.

```
public ConfigurationError[] validateConfiguration() {
    return null;
}
```

You implemented the methods of the `IConfigurationAdaptor` and `SDKHelper` classes to load and save configuration information in the configuration server and defined methods to obtain and set the configurable values.

What to do next

Obtain configuration information that the user sets in the configuration interface by implementing the Apache Struts framework.

Create a Configuration Action to Obtain Configuration Information from the User

Orchestrator uses the Apache Struts framework to pass to the Orchestrator server the configuration information that the user provides in the configuration interface.

You can implement the action that passes configuration information from the Orchestrator configuration interface to the configuration server directly in the configuration adapter implementation. However, the solar system example defines this action in a separate class named `SolarSystemConfigureAction`. The `SolarSystemConfigureAction` class uses the methods that the `SolarSystemConfigurationAdaptor` class defines to load and save configuration information. The `SolarSystemConfigureAction` class implements the Apache Struts framework to obtain the configuration information from the configuration interface and pass it to the Orchestrator configuration server. The tab that you add to the configuration interface is a Struts Web application.

Prerequisites

- Implement the plug-in adapter and factory interfaces.
- Set up the configuration adapter implementation class.
- Implement the methods of the `IConfigurationAdaptor` interface to load, save, and validate configuration information.

Procedure

1 Create and save a Java file for the plug-in configuration action implementation.

   In the solar system example, the configuration action class is named `SolarSystemConfigureAction.java`.

2 Declare the package that contains the Java classes of the plug-in configuration implementation.

   The solar system example declares the following package.

```
com.vmware.orchestrator.api.sample.solarsystem.config;
```

3 Import the classes of the Orchestrator API with a Java import statement.

   The `SolarSystemConfigureAction` class requires the following classes:

```
import ch.dunes.vso.configuration.web.commons.BaseAction;
import ch.dunes.vso.sdk.conf.ConfigurationError;
```

4 Import the third-party Java classes that the configuration action requires.

   The `SolarSystemConfigureAction` class requires the following classes:

```
import org.apache.log4j.Logger;
import com.opensymphony.xwork2.ModelDriven;
```

5 Declare a public class that implements the `BaseAction` and `ModelDriven` classes.

   The Orchestrator `BaseAction` class defines how Orchestrator interacts with the Struts framework. The OpenSymphony XWork2 `ModelDriven` class pushes objects to the Struts framework.

   The `SolarSystemConfigureAction` class implements these classes and creates a generic instance of the `SolarSystemConfigurationAdaptor` class.

```
public class SolarSystemConfigureAction extends BaseAction
 implements ModelDriven<SolarSystemConfigurationAdaptor> {
}
```

6 Define the variables that the `BaseAction` implementation requires.

   The `SolarSystemConfigureAction` class declares variables for the unique identifier of each instance of the serializable `BaseAction` class, a logger, an array of `ConfigurationError` instances, and the configuration adapter instance.

```
private static final long serialVersionUID = 1L;
private static Logger log = Logger.getLogger(SolarSystemConfigureAction.class);
private ConfigurationError[] configErrors;
private SolarSystemConfigurationAdaptor solarSystemConfigurationAdaptor;
```

7 Implement the `BaseAction.prepare()` method to instantiate the configuration adapter and load the configuration information.

   The `SolarSystemConfigureAction` class creates an instance of the `SolarSystemConfigurationAdaptor` class and calls the `SolarSystemConfigurationAdaptor.loadConfiguration()` method.

```
public void prepare() throws Exception {
    solarSystemConfigurationAdaptor = new SolarSystemConfigurationAdaptor();
    solarSystemConfigurationAdaptor.loadConfiguration(null);
}
```

8 Implement the `BaseAction.execute()` method to log and validate the configuration information.

   The `SolarSystemConfigureAction` class implements the `BaseAction.execute()` method to record in the logs the result of calling the `SolarSystemConfigurationAdaptor.validateConfiguration()` method.

```
public String execute() throws Exception {
    log.debug("SolarSystemConfigureAction execute method");
    configErrors = solarSystemConfigurationAdaptor.validateConfiguration();
    return SUCCESS;
}
```

9 Implement a `save()` method to save the configuration information.

   The `SolarSystemConfigureAction` class implements the `save()` method to record in the logs the result of calling the `SolarSystemConfigurationAdaptor.saveConfiguration()` method.

```
public String save() throws Exception {
    log.debug("SolarSystemConfigureAction save method");
    solarSystemConfigurationAdaptor.saveConfiguration(null);
    configErrors = solarSystemConfigurationAdaptor.validateConfiguration();
    return SUCCESS;
}
```

10 Add error handling to the implementation of the configuration action.

The `SolarSystemConfigureAction` class returns an array of errors if the configuration is invalid.

```
public ConfigurationError[] getConfigErrors() {
    return configErrors;
}
public void setConfigErrors(ConfigurationError[] configErrors) {
    this.configErrors = configErrors;
}
public int getConfigErrorsSize() {
    return configErrors.length;
}
```

11 Add the configuration adapter to the Struts Web application in the configuration interface by implementing the `ModelDriven.getModel()` class from the OpenSymphony XWork2 framework.

The `SolarSystemConfigureAction` class passes an instance of the `SolarSystemConfigurationAdaptor` to the Struts framework.

```
public SolarSystemConfigurationAdaptor getModel() {
    return solarSystemConfigurationAdaptor;
}
```

You created the configuration action that instantiates the configuration adapter and implements the Orchestrator `BaseAction` and OpenSymphony `ModelDriven` classes. The `BaseAction` and `ModelDriven` classes pass configuration information from the Orchestrator configuration interface to the Orchestrator server through the Struts framework.

What to do next

Create a Struts-based Web application to add a tab to the Orchestrator configuration interface.

Create a Struts-Based Web Application to Add to the Configuration Interface

The tab that you add to the Orchestrator configuration interface is an Apache Struts-based Web application. You define the layout of the page by using HTML or JavaServer Pages (JSP) and add actions to the page by implementing the Struts framework.

To add a configuration tab to the Orchestrator configuration interface, you must include a Web application archive (WAR) file in the DAR file of the plug-in.

The DAR file of the solar system plug-in contains a built WAR file for the solar system configuration tab. You can also examine the files of the solar system configuration Web application in the bundle of source files of the solar system plug-in.

- `o11nplugin-solarsystem-config\src\main\webapp\index.jsp`
- `o11nplugin-solarsystem-config\src\main\webapp\WEB-INF\web.xml`
- `o11nplugin-solarsystem-config\src\main\webapp\WEB-INF\pages\configure.jsp`
- `o11nplugin-solarsystem-config\src\main\resources\struts.xml`

You must be familiar with Web application development technologies, including the Struts framework and JSP. See the Apache Struts documentation for information about Struts. For details of all the directories and files that the WAR file contains, see "Contents of the Solar System Configuration WAR File," on page 69.

Prerequisites

- Implement the plug-in adapter and factory interfaces.
- Create the configuration adapter and configuration action implementations.

Procedure

1. Create an index page for the configuration tab that implements the JavaServer Pages Standard Tag Library (JSTL).

   The index page must refer to the JSTL definition.

   ```
   <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
   ```

2. Create a Web application definition XML file that provides information about the plug-in to the Orchestrator configuration server.

   The Web application definition file references the XML schemas, locates the index page, accesses resource files, and implements security.

3. Create a JSP page that defines the layout and contents of the configuration tab.

   The configuration tab can include any types of buttons, forms, lists, or menus that JSP provides.

   The solar system configuration tab includes a button that the user can use to set the `plutoClassifiedAsAPlanet` property to designate Pluto as a planet or a dwarf planet, a list from which they can select their home planet from the `allPlanets` properties list, and a save button that calls a Struts action named `ConfigureSave`.

   ```
   <div id="c_content">
    <s:form action="ConfigureSave" method="POST" validate="true">
     <s:radio key="select.pluto.classify.text" name="plutoClassifiedAsAPlanet"
      list="plutoClassifyList"/>
     <s:select key="select.home.planet" list="allPlanets" listValue="name" listKey="id"
      name="homePlanet"/>
     <s:submit type="input" key="save.button"/>
    </s:form>
   </div>
   ```

4. Create a Struts configuration file that implements the Struts framework to pass to the configuration server the configuration information that the user enters.

   The Struts configuration file in the solar system example implements the OpenSymphoyOpenSymphony XWork2 `ActionSupport` class to pass the results of the `ConfigureSave` action to the Struts framework. The `ConfigureSave` action calls the `SolarSystemConfigureAction.save()` method.

   ```
   <action name="Default" class="com.opensymphony.xwork2.ActionSupport">
    <result name="success" type="chain">Configure</result>
   </action>

   <action name="Configure"
    class="com.vmware.orchestrator.api.sample.solarsystem.config.SolarSystemConfigureAction">
     <result name="success">/WEB-INF/pages/configure.jsp</result>
   </action>

   <action name="ConfigureSave"
    class="com.vmware.orchestrator.api.sample.solarsystem.config.SolarSystemConfigureAction"
    method="save">
     <result name="success">/WEB-INF/pages/configure.jsp</result>
   </action>
   ```

5. Copy all of the required JAR files in a directory named `lib` in the Web application directory.

   You must include the JAR files that contain the implementations of the configuration adapter and actions, and JAR files for any other technologies that configuration implementation uses.

6 Create a Web application archive to contain the Web application files.

   A WAR file is a JAR file that you rename to `.war`.

   In the solar system example, the configuration Web application files are stored in WAR file named `o11nplugin-solarsystem-config.war`.

You created a Struts-based Web application that contains all of the Web application files and the Java implementations of the configuration adapter and action.

What to do next

Map the application and the plug-in implementation to Orchestrator objects in the `vso.xml` file.

Contents of the Solar System Configuration WAR File

You add a configuration tab to the Orchestrator configuration interface by creating a Web application archive (WAR) file that contains the implementations of the configuration adapter and configuration actions, the layout of the tab, and the Struts configuration files.

The DAR file of the solar system plug-in contains a built WAR file for the solar system configuration tab. You can also examine the files of the solar system configuration Web application in the bundle of source files of the solar system plug-in.

The solar system DAR file contains a WAR file named `o11nplugin-solarsystem-config.war`.

If you modify the source files of the solar system configuration WAR file, you must rebuild the DAR file of the solar system plug-in by using the Ant build tool. Rebuilding the plug-in DAR file rebuilds the WAR file of the solar system configuration tab. See "Build the Solar System Application and Plug-In," on page 78.

The following table lists the directories and files that the built WAR file for the solar system configuration tab contains.

Table 3-4. Contents of the Solar System Configuration WAR File

Directory	Filename	Description
`o11nplugin-solarsystem-config\`	`index.jsp`	JSP file that defines the layout of the configuration tab for the plug-in.
`o11nplugin-solarsystem-config\WEB-INF\`	`web.xml`	Sets up the configuration tab by implementing the appropriate XML schemas, locating the index page, accessing resource files, and implementing security.
`o11nplugin-solarsystem-config\WEB-INF\classes\`	`struts.xml`	Struts framework configuration file that references the Java classes and methods of the configuration actions.
`o11nplugin-solarsystem-config\WEB-INF\classes\com\vmware\orchestrator\api\sample\solarsystem\config\`	`SolarSystemConfigureAction.class`	Java class that defines the configuration actions that the configuration tab performs.
`o11nplugin-solarsystem-config\WEB-INF\lib\`	`o11nplugin-solarsystem-core-1.0.0.jar`	JAR file that contains the binaries of the solar system plug-in implementation.

Table 3-4. Contents of the Solar System Configuration WAR File (Continued)

Directory	Filename	Description
`o11nplugin-solarsystem-config\WEB-INF\lib\`	`o11nplugin-solarsystem-model-1.0.0.jar`	JAR file that contains the binaries of the solar system application.
`o11nplugin-solarsystem-config\WEB-INF\pages\`	`configure.jsp`	JSP file that adds buttons and a drop-down list to the configuration tab. The buttons and list implement the configuration actions that the `SolarSystemConfigureAction` class defines and that `struts.xml` maps to the Struts framework.

Map the Application in the vso.xml File

The `vso.xml` file defines how Orchestrator accesses and interacts with the plugged-in technology. The `vso.xml` file maps objects and operations in the plugged-in technology and in the plug-in implementation to Orchestrator objects and operations.

You can use the objects and operations that you map to create Orchestrator workflows, policies, and actions to interact with the plugged-in technology by using Orchestrator.

You find the `vso.xml` file for the solar system example in the following location in the solar system source files in the Orchestrator examples bundle:

`o11nplugin-solarsystem\src\main\dar\VSO-INF\`

For full descriptions of all of the elements of a `vso.xml` file and all of their attributes, see Chapter 6, "Elements of the vso.xml Plug-In Definition File," on page 109.

Prerequisites

- Verify that you have an application to plug in to Orchestrator.
- Implement the plug-in adapter and factory interfaces.
- Implement the configuration adapter interface and create the configuration Web application.

Procedure

1. Set Up the Global Plug-In Information on page 71

   To create a plug-in, you must point Orchestrator to the relevant XML schema definition and the source files for the application and plug-in. You must also define the behavior of the plug-in when Orchestrator starts and provide a root object for the hierarchy of objects that the plug-in exposes.

2. Map Objects in the Plugged-In Technology to Scripting Types and Inventory Objects on page 72

   To allow Orchestrator to access objects in a plugged-in application, you must define how and where the plug-in finds those objects.

3. Define Enumerations on page 74

   You can define enumerations in the `vso.xml` file to set global values that apply to all objects of a certain category.

4. Map Classes and Methods to Classes and Methods in the JavaScript API on page 75

   Orchestrator monitors objects in the plugged-in application and performs operations on them by running workflows, policies, and actions. You map in the `vso.xml` file the classes and methods from the plugged-in technology and from the plug-in implementation to JavaScript classes and methods in the Orchestrator JavaScript API.

Set Up the Global Plug-In Information

To create a plug-in, you must point Orchestrator to the relevant XML schema definition and the source files for the application and plug-in. You must also define the behavior of the plug-in when Orchestrator starts and provide a root object for the hierarchy of objects that the plug-in exposes.

Procedure

1 Create a file named `vso.xml`.

2 Set up the `<module>` element to provide basic information about the plug-in, including a pointer to the Orchestrator plug-in XML schema definition.

   The `<module>` element in the `vso.xml` file for the solar system example sets the plug-in name to `SolarSystem`, sets the version number, and provides the path in the DAR archive to the icon that represents this plug-in in the Orchestrator **Inventory** view and selection dialog boxes.

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.vmware.com/support/orchestrator/plugin-4-1.xsd"
    name="SolarSystem" version="1.0.0" build-number="4" image="images/solarSystem-16x16.png">
```

3 Provide a description of the plug-in in the `<description>` element.

   The following example shows a `<description>` element for the solar system.

```
<description>Example plug-in to a solar system application.</description>
```

4 Add a tab for the plug-in to the configuration interface by referencing the configuration adapter in the `<configuration>` element.

   The `<configuration>` element in the `vso.xml` file for the solar system example identifies an icon for the solar system plug-in tab in the file structure of the DAR file, references the configuration adapter implementation and the configuration Web application, and activates validation of the information that the user provides.

```
<configuration
 icon="images/solarSystem_32x32.png"
 adaptor-class=
  "com.vmware.orchestrator.api.sample.solarsystem.config.SolarSystemConfigurationAdaptor"
 configuration-war="o11nplugin-solarsystem-config.war"
    validation="enabled" />
```

5 Set the `<installation>` and `<action>` elements to define the behavior of the plug-in when the Orchestrator server starts.

   The solar system example sets the `version` mode to restart the plug-in whenever a new version is detected, and provides the path to a package of Orchestrator workflows, policies, and a Web view in the file structure of the DAR file. Orchestrator installs this package when the plug-in starts.

```
<installation mode="version">
        <action type="install-package" resource="packages/com.vmware.solarsystem.package" />
    </installation>
```

6 Set the root of the hierarchy of object types in the `<inventory>` element.

   The solar system plug-in defines the root of the hierarchy that represents the plug-in in the Orchestrator scripting API as an object of the type `Galaxy`. All of the other solar system objects relate to the `Galaxy` object.

```
<inventory type="Galaxy"/>
```

You set up the elements that identify the plug-in to Orchestrator, added a tab to the configuration interface, defined the start-up behavior, and defined the root scripting object type for the objects in the plug-in.

What to do next

Define the types of objects that Orchestrator finds through the plug-in by mapping the finder objects that the plug-in factory implementation defines to `<finder>` elements in the `vso.xml` file.

Map Objects in the Plugged-In Technology to Scripting Types and Inventory Objects

To allow Orchestrator to access objects in a plugged-in application, you must define how and where the plug-in finds those objects.

The objects you map in the `vso.xml` file appear as scripting types in the Orchestrator JavaScript API. Instances of these objects appear in the Orchestrator inventory.

Prerequisites

You must have created the `vso.xml` file, defined how Orchestrator identifies the plug-in, and referenced the configuration adapter.

Procedure

1. Set the data sources for the plug-in `<finder>` elements in the `<finder-datasources>` element.

   The plug-in adapter implementation is the point of access to all the classes of the plug-in.

   The solar system plug-in `vso.xml` file sets the name of the data source to `solar-datasource` and points the `<finder>` elements to the `SolarSystemAdapter` class that instantiates the `SolarSystemFactory` and the other classes of the plug-in.

   ```
   <finder-datasources>
    <finder-datasource name="solar-datasource"
     adaptor-class=
      "com.vmware.orchestrator.api.sample.solarsystem.SolarSystemAdapter"
     anonymous-login-mode="internal"/>
   </finder-datasources>
   ```

2. Define how the plug-in finds objects in the plugged-in technology in `<finder>` elements.

   The following extract from the solar system `vso.xml` file shows the `<finder>` element for objects of the type `Star`.

   ```
   <finders>
    <finder type="Star" datasource="solar-datasource"
     java-class="com.vmware.solarsystem.Star"
     script-object="Star" image="images/sun_16x16.png">
     [...]
    </finder>
    [...]
   </finders>
   ```

   The `<finder>` element for `Star` objects obtains their data from the data source that the `<finder-datasource>` element defines. The `Star` finder type represents instances of the `com.vmware.solarsystem.Star` class in the Orchestrator inventory. The finder element for `Star` objects defines a `Star` scripting type that appears in the Orchestrator JavaScript API.

3 Obtain the identifier of the object in the <id> element.

   The solar system example obtains the identifier of the object by calling the getId() method that the solar system application's CelestialBody class defines.

```
<id accessor="getId()" />
```

4 Define the object's relations in the <relations> element.

   The solar system example defines a relation named OrbitingPlanets to relate objects of the type Planet to the Star object that this <finder> element finds.

```
<relations>
  <relation type="Planet" name="OrbitingPlanets"/>
</relations>
```

5 Set the hierarchy of objects in the Orchestrator inventory tab according to their relation to the parent.

   The solar system example places all objects related to Star objects type by the OrbitingPlanets relation immediately beneath the star in the inventory hierarchy.

```
<inventory-children>
  <relation-link name="OrbitingPlanets"></relation-link>
</inventory-children>
```

6 Set the object's properties in the <properties> element.

   The solar system example defines name, circumference, and surfaceTemp properties for all Star objects. The bean-property property allows Orchestrator to create get and set methods in the scripting API to obtain and set these properties.

```
<properties>
  <property display-name="Name" name="name"
    bean-property="name"/>
  <property display-name="Circumference" name="circumference"
    bean-property="circumference"/>
  <property display-name="Surface Temperature" name="surfaceTemp"
    bean-property="surfaceTemp"/>
</properties>
```

7 Set the events that can occur on the object in the <events> element.

   Events can be either gauges or triggers.

   In the solar system example, the SolarSystemEventGenerator class defines a generateFlareEvent() method to generate solar flares on Star objects. The <gauge> element monitors the values of the flare events that occur on Star objects.

```
<events>
  <gauge min-value="0" name="Flare" unit="number">
    <description>Magnitude of the flare</description>
  </gauge>
</events>
```

You defined a <finder> element to find objects of a certain type in the plugged-in application. The objects that the finder finds appear as scripting types in the Orchestrator JavaScript API and instances of these types appear in the Orchestrator inventory.

What to do next

Define enumerations to set values that apply to all objects of a certain type.

Solar System Finder Mappings

The `vso.xml` file for the solar system example maps objects from the solar system application to objects that appear as scripting types in the Orchestrator JavaScript API. Instances of these objects appear in the Orchestrator inventory.

The following table lists the mappings that the `vso.xml` file defines for each type of object that the solar system application defines.

Table 3-5. Solar System Finder Mappings

Scripting Type	Source Class	Inventory Children	Properties	Events
`Galaxy`	None	`Stars`	None	None
`Star`	`Star`, defined by application	`OrbitingPlanets`	■ `name` ■ `circumference` ■ `surfaceTemp`	Flare
`Planet`	`Planet`, defined by application	`OrbitingMoons`	■ `name` ■ `circumference` ■ `gravity`	None
`Moon`	`Moon`, defined by application	None	■ `name` ■ `volume`	None

Define Enumerations

You can define enumerations in the `vso.xml` file to set global values that apply to all objects of a certain category.

The categories that you set in the `vso.xml` file appear as enumerations in the Orchestrator JavaScript API.

Prerequisites

Set up the plug-in and define `<finder>` elements in the `vso.xml` file.

Procedure

1 Define an enumeration for a certain object type in the `<enumerations>` element.

   The solar system example defines enumerations to set a `PlanetCategory` enumeration on `Planet` objects.

```
<enumerations>
  <enumeration type="PlanetCategory">
    <description>Define the category of a Planet</description>
  [...]
</enumeration>
```

2 Define entries for the enumerations that apply values to objects in the given object category.

   The solar system example defines values that represent different types of planet.

```
<entries>
  <entry id="gaz"
    name="Huge Gaz">Huge planet with only gaz atmosphere.
      No Physical core.</entry>
  <entry id="earth"
    name="Earth">You could live on this planet.</entry>
  <entry id="desert"
    name="Desert">Planet without water.</entry>
  <entry id="ice"
```

```
    name="Ice">Planet with water but completely frozen.</entry>
  <entry id="other"
    name="Other">Does not fit into any category.</entry>
</entries>
```

The vso.xml file of the solar system example also defines a StarCategory enumeration that allows you to define a Star object as a blue dwarf, a nova, or a yellow sun.

You defined enumerations that can apply to all objects in a certain category.

What to do next

Map the classes and methods of the plugged-in technology and the plug-in implementation to JavaScript classes and methods in the Orchestrator JavaScript API.

Map Classes and Methods to Classes and Methods in the JavaScript API

Orchestrator monitors objects in the plugged-in application and performs operations on them by running workflows, policies, and actions. You map in the vso.xml file the classes and methods from the plugged-in technology and from the plug-in implementation to JavaScript classes and methods in the Orchestrator JavaScript API.

You identify in <scripting-objects><object> elements the classes and methods to map to classes and methods in the JavaScript API.

Prerequisites

Set up the plug-in, and define <finder> elements and enumerations.

Procedure

1 Map a Java class from the plugged-in technology or from the plug-in implementation to a JavaScript class.

   The SolarSystemEventGenerator class defines the events that Orchestrator can invoke in the solar system application. The solar system vso.xml file maps the event generator class to a JavaScript class named _SolarSystemEventGenerator. By setting the strict attribute to true, Orchestrator can only call the methods from the SolarSystemEventGenerator class that are mapped in the vso.xml file. To allow scripting to instantiate a class you use the create attribute.

```
<scripting-objects>
    <object script-name="_SolarSystemEventGenerator"
    java-class="com.vmware.orchestrator.api.sample.solarsystem.SolarSystemEventGenerator"
    strict="true">
    <description>The entry point to generate events</description>
    [...]
    </object>
    [...]
</scripting-objects>
```

2 (Optional) If necessary, denote the JavaScript object as a singleton object.

   In the solar system example, SolarSystemEventGenerator is a singleton object. The plug-in adaptor can only create a single instance of the SolarSystemEventGenerator class. You can only call the methods of the SolarSystemEventGenerator JavaScript object and cannot instantiate the class in Orchestrator scripts.

```
<singleton script-name="SolarSystemEventGenerator"
    datasource="solar-datasource"/>
```

3 Map the methods in the Java class to methods in the Orchestrator JavaScript API in the `<object><methods>` element.

   The `SolarSystemEventGenerator` class defines a `generateFlareEvent()` method to generate solar flare events. The solar system `vso.xml` maps this method to a JavaScript method of the same name, and sets its parameters in the JavaScript method.

```
<methods>
    <method script-name="generateFlareEvent" java-name="generateFlareEvent">
  <description>Start a Solar Flare</description>
     <parameters>
   <parameter name="star" type="Star">The star which generates the event</parameter>
   <parameter name="magnitude" type="number">The magnitude of the flare</parameter>
  </parameters>
    </method>
</methods>
```

4 Map the attributes of a Java class to JavaScript attributes in `<object><attributes>` elements.

   The solar system `vso.xml` file maps the Java attributes of the `Star` object to attributes of the same name in the `Star` JavaScript class in the Orchestrator JavaScript API.

```
<object script-name="Star" java-class="com.vmware.solarsystem.Star"
 create="false" strict="true">
 [...]
 <attributes>
  <attribute script-name="id" java-name="id" return-type="string">
   The unique Id of the star</attribute>
  <attribute script-name="name" java-name="name" return-type="string">
   The name of the star</attribute>
  <attribute script-name="circumference" java-name="circumference" return-type="number">
   Circumference of the star</attribute>
  <attribute script-name="temperature" java-name="surfaceTemp" return-type="number">
   The temperature on the star's surface</attribute>
 </attributes>
 [...]
</object>
```

You mapped classes and their methods and attributes from the classes in the plugged-in technology and plug-in implementation to a JavaScript class and methods in the Orchestrator JavaScript API.

What to do next

Create or rebuild the DAR file for the plug-in.

Solar System JavaScript API Mappings

The `vso.xml` file for the solar system example maps objects, classes, methods, and attributes from the solar system application to scripting types, classes, methods, and attributes in the Orchestrator JavaScript API.

The following table lists the classes, methods, and attributes from the solar system application and plug-in implemention that the `vso.xml` file maps to the Orchestrator JavaScript API.

Table 3-6. Solar System JavaScript API Mappings

Scripting Class	Source Class	Attributes	Methods
`SolarSystemEventGenerator`	`SolarSystemEventGenerator`, defined by plug-in	None	`generateFlareEvent( star, magnitude)`
`SolarSystemTriggerGenerator`	`SolarSystemTriggerGenerator`, defined by plug-in	None	`createStarFlareTrigger( star, minMagnitude)`

Table 3-6. Solar System JavaScript API Mappings (Continued)

Scripting Class	Source Class	Attributes	Methods
`Star`	`Star`, defined by application	■ `id` ■ `name` ■ `circumference` ■ `temperature`	■ `addPlanet(planet)` ■ `removePlanet(planet)`
`Planet`	`Planet`, defined by application	■ `id` ■ `name` ■ `circumference` ■ `gravity` ■ `starId`	■ `addMoon(moon)` ■ `removeMoon(moon)`
`Moon`	`Moon`, defined by application	■ `id` ■ `name` ■ `volume` ■ `planetId`	None

Create the Plug-In DAR Archive

The final stage in the creation of a plug-in is to create the DAR archive that you import to Orchestrator.

IMPORTANT This is only necessary if you are building a skeleton project. The sample projects already include these directories and files.

The DAR archive is a standard ZIP file that you rename to `.dar`. The DAR archive contains all of the elements of the plug-in implementation and must adhere to a standard file and folder structure.

Prerequisites

- Implement the plug-in adapter and factory interfaces.
- Implement the configuration adapter interface and create the configuration Web application.
- Map the application to Orchestrator objects in the `vso.xml` file.

Procedure

1. Create a working directory in which to create the DAR archive.

   For example, create a directory named *plugin_name*.

2. Create a directory named `VSO-INF` at the root of the working directory.
3. Copy the `vso.xml` file to `VSO-INF`.
4. Create a directory named `lib` at the root of the working directory.
5. Add the JAR files containing the classes of the application to plug in and the classes of the plug-in adapter and factory implementations to `lib`.
6. (Optional) Create a directory named `webapps` at the root of the working directory.

   The `webapps` contains the WAR file of the configuration tab Web application.

7. Create a directory named `resources` at the root of the working directory.
8. (Optional) Create a directory named `images` in the `resources` directory.

   The `resources\images` directory can contain icons to represent the different objects of the plugged-in application in the Orchestrator **Inventory** view and selection dialog boxes.

9 (Optional) Create a directory named `packages` in the `resources` directory.

   The `resources\packages` directory can contain packages of workflows, actions, policies, Web views, and so on, that interact with the plugged-in application.

10 Create a ZIP archive that contains all of the preceding directories and files.

11 Rename the ZIP archive to *`plugin_name`*`.dar`.

You created the DAR archive that contains a plug-in, and imported it to Orchestrator.

What to do next

You can access the objects of the plugged-in application in the Orchestrator inventory to perform operations on them. You can also use the objects and methods that you mapped to the Orchestrator scripting API to create workflows, actions, policies, Web views, and so on, to interact with the objects through the plug-in.

Build the Solar System Application and Plug-In

If you adapt the solar system application or the solar system plug-in, you can build the DAR file by using the Apache Ant building tool.

The Orchestrator examples bundle contains the scripts and `build.xml` file that allow you to build the solar system DAR file and the JAR files that it contains. If you add new files to the solar system plug-in, you must update the `build.xml` file.

Prerequisites

Verify that you have Apache Ant 1.7.1 or later installed and configured on your system.

Procedure

1 Navigate to the folder that contains the solar system application and plug-in.

   `install-directory``\vco-samples-`*`version_number-build_number`*`\Plug-Ins\solarsystem`

2 Open the `maven-build.properties` file in a text editor and edit the `o11n.root.path` property to point to the root folder of the Orchestrator installation.

3 Open a terminal window and navigate to the *`install-directory`*`\vco-samples-`*`version_number-build_number`*`\Plug-Ins\solarsystem` folder.

4 Type the `ant` command in the terminal window.

You built the solar system DAR file to incorporate any modifications that you made to the application or to the plug-in.

What to do next

Install the solar system DAR file in the Orchestrator server.

Contents of the Solar System DAR File

The DAR file is a ZIP file that you rename to DAR. You can unzip the solar system DAR file to view the contents and file structure of the solar system plug-in.

The solar system example `o11nplugin-solarsystem.dar` file contains the following directories and files.

- `\lib`, that contains the following JAR archives:
  - `o11nplugin-solarsystem-core-1.0.0.jar`, that contains the classes of the plug-in adapter and factory implementations for the solar system application.
  - `o11nplugin-solarsystem-model-1.0.0.jar`, that contains the classes of the solar system application.

- `\resources`, that contains the following directories:
  - `\images`, that contains icons that represent the different objects of the solar system application on the Orchestrator **Inventory** tab.
  - `\packages`, that contains an Orchestrator package named `com.vmware.solarsystem.package`. The package contains workflows, policies, actions, and the Web view that allow Orchestrator to interact with the solar system application.
- `\VSO-INF\vso.xml`, the XML file that maps the solar system application to Orchestrator objects.
- `\webapps`, that contains the `o11nplugin-solarsystem-config.war` file for the Web application of the solar system configuration tab.

Install a Plug-In in the Orchestrator Server

After you create the plug-in DAR file, you must install it in the Orchestrator server. You install plug-ins in the Orchestrator configuration interface.

Prerequisites

Verify that you have a completed DAR file for a plug-in.

Procedure

1. Open the Orchestrator configuration interface in a Web browser and log in.

   `http://`*`orchestrator_server_DNS_name_or_IP_address`*`:8282`

2. Click **Plug-ins**.
3. Type the credentials for a user who is a member of the Orchestrator Administration group.

   When the Orchestrator server starts, the system uses these credentials to set up the plug-ins. The system checks the enabled plug-ins and performs any necessary internal installations such as package import, policy run, script launch, and so on.

4. Click the magnifying glass icon and select the DAR file to install.
5. Click **Open**.
6. Click **Upload and install**.

   The installed plug-in file is stored in the *`install_directory`*`\app-server\server\vmo\plugins` folder.

7. Click **Apply changes**.

   Depending on the plug-in, the configuration server might restart.

8. (Optional) If the plug-in adds a configuration tab to the configuration interface, click the tab for the plug-in.

   For example, if you install the solar system plug-in, click **Solar System**.

9. (Optional) Configure the plug-in in its configuration tab.

   For example, if you install the solar system plug-in, select your home planet and select whether Pluto is a planet or a dwarf planet.

10. Click **Apply changes**.
11. Restart the Orchestrator server.

You installed and configured a plug-in.

What to do next

Use Orchestrator to interact with the plugged-in technology.

Interact with the Solar System Application by Using Orchestrator

After you install a plug-in in the Orchestrator server, you can use the objects that it adds to the Orchestrator JavaScript API to create workflows, actions, policies, Web views, and so on. You use these items to interact with the plugged-in technology using Orchestrator.

The solar system plug-in includes the `com.vmware.samples.solarsystem` package that contains workflows, actions, a policy, and a Web view that implement the API objects that the solar system plug-in adds to the Orchestrator JavaScript API.

Procedure

1 View Plug-In Scripting Objects in the JavaScript API on page 80

   The objects of a plugged-in technology that you map to Orchestrator scripting objects appear in the Orchestrator JavaScript API.

2 Run Workflows on Plug-In Objects in the Inventory on page 81

   You can use the scripting objects that a plug-in adds to the Orchestrator JavaScript API to write workflows and actions to interact with the plugged-in technology.

3 Monitor Plug-In Events by Using Policies on page 82

   You can use policies to monitor events in a plugged-in technology and perform defined operations when the events occur.

4 Monitor Plug-In Events by Using Workflows on page 83

   Workflows can include a Wait Event element that suspends the workflow and waits for an event to occur in a plugged-in technology. Plug-ins can implement triggers and watchers to notify waiting workflows of the events that occur.

5 Access Plug-In Objects and Operations by Using a Web View on page 83

   With Web views, you can run workflows on objects from the Orchestrator inventory from a Web browser instead of from the Orchestrator client.

View Plug-In Scripting Objects in the JavaScript API

The objects of a plugged-in technology that you map to Orchestrator scripting objects appear in the Orchestrator JavaScript API.

The solar system `vso.xml` file maps objects from the solar system application and plug-in to classes, methods, attributes, and enumerations in the Orchestrator JavaScript API.

Prerequisites

- Install the solar system plug-in in the Orchestrator server.
- Start the Orchestrator client.

Procedure

1. In the Orchestrator client, select **Tools > API Explorer**.
2. Expand the **SolarSystem** node in the hierarchical list of scripting objects.

   You see the scripting types for the different types of celestial bodies, scripting classes for the `Star`, `Planet`, `Moon`, `SolarSystemEventGenerator`, and `SolarSystemTriggersManager` objects, and enumerations to define categories of stars and planets.

3. Expand a scripting class in the hierarchical list to see the scripting methods and attributes that it defines.

   The scripting methods and attributes are those that the `vso.xml` file maps for each object.

You can use the scripting objects from the plug-in to create workflows, actions, policies, and so on.

What to do next

Run a workflow on a solar system object in the **Inventory** view.

Run Workflows on Plug-In Objects in the Inventory

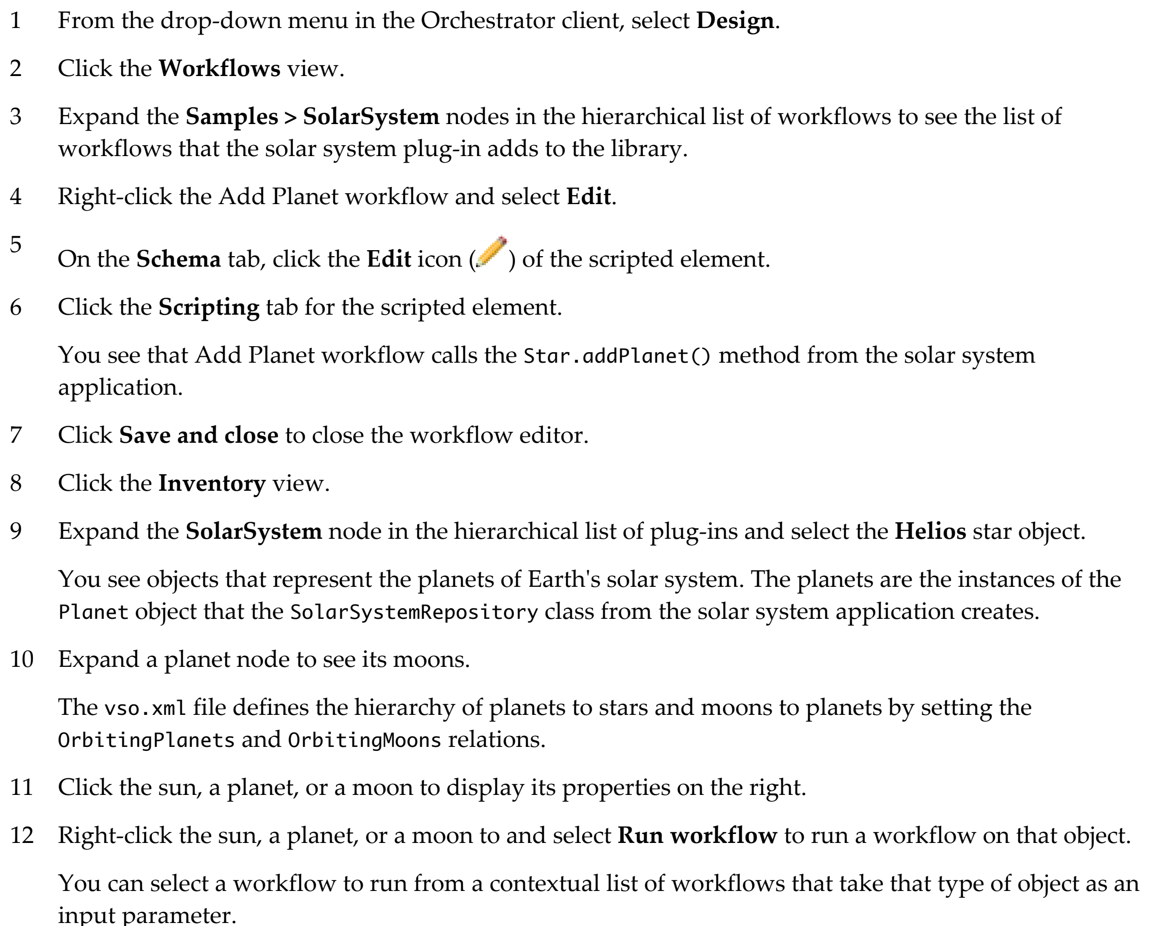
You can use the scripting objects that a plug-in adds to the Orchestrator JavaScript API to write workflows and actions to interact with the plugged-in technology.

The solar system plug-in includes a package of workflows that you can use to perform operations on the objects that the plug-in adds to the Orchestrator inventory.

Prerequisites

- Install the solar system plug-in in the Orchestrator server.
- Start the Orchestrator client.

Procedure

1. From the drop-down menu in the Orchestrator client, select **Design**.
2. Click the **Workflows** view.
3. Expand the **Samples > SolarSystem** nodes in the hierarchical list of workflows to see the list of workflows that the solar system plug-in adds to the library.
4. Right-click the Add Planet workflow and select **Edit**.
5. On the **Schema** tab, click the **Edit** icon () of the scripted element.
6. Click the **Scripting** tab for the scripted element.

   You see that Add Planet workflow calls the `Star.addPlanet()` method from the solar system application.
7. Click **Save and close** to close the workflow editor.
8. Click the **Inventory** view.
9. Expand the **SolarSystem** node in the hierarchical list of plug-ins and select the **Helios** star object.

   You see objects that represent the planets of Earth's solar system. The planets are the instances of the `Planet` object that the `SolarSystemRepository` class from the solar system application creates.
10. Expand a planet node to see its moons.

    The `vso.xml` file defines the hierarchy of planets to stars and moons to planets by setting the `OrbitingPlanets` and `OrbitingMoons` relations.
11. Click the sun, a planet, or a moon to display its properties on the right.
12. Right-click the sun, a planet, or a moon to and select **Run workflow** to run a workflow on that object.

    You can select a workflow to run from a contextual list of workflows that take that type of object as an input parameter.

You can run workflows on the solar system objects in the inventory. You can add a planet to the sun's orbit or generate or wait for a solar flare. You can modify the circumference of planets or split or destroy them.

Note You can use the Orchestrator workflow debugging tool to inspect the details of the workflow run. For information about debugging workflows, see *Developing with VMware vCenter Orchestrator*.

What to do next

Monitor events in the solar system application by setting a policy.

Monitor Plug-In Events by Using Policies

You can use policies to monitor events in a plugged-in technology and perform defined operations when the events occur.

The solar system plug-in includes a policy that monitors a star object for solar flares. When flares occur, the policy records the magnitude of the flares in the logs.

Prerequisites

- Install the solar system plug-in in the Orchestrator server.
- Start the Orchestrator client.

Procedure

1. From the drop-down menu in the Orchestrator client, select **Administer**.
2. Click the **Policy Templates** view.
3. Expand the **Samples > SolarSystem** nodes in the hierarchical list of policies.
4. Right-click the **Star** policy and select **Apply Policy**.
5. Add a policy description and select a `Star` object on which to apply the policy in the **Apply Policy** dialog box and click **Submit**.

   The **Star** policy opens in the **Policies** tab.
6. Click the **Star** policy and open the **Scripting** tab.

   In the scripting tab you see that the policy is monitoring a threshold named `Flare`.
7. Right-click the **Star** policy and select **Start policy**.
8. Click the **Inventory** tab.
9. Right-click the **Helios** star and select **Run workflow**.
10. Run the Generate Flare Event workflow, setting the magnitude of the flare to **`100`**.

    The Generate Flare Event workflow includes a script that calls the `SolarSystemEventGenerator.generateFlareEvent()` method. The gauge that the `SolarSystemEventGenerator` class implements pushes an event object named `Flare` to the Orchestrator policy engine.
11. Click the **Policies** tab.
12. Click the **Star** policy.
13. Click the **Logs** tab.

    The policy has recorded the magnitude of the solar flare event in the logs.

The `Star` policy implements the solar system scripting API to monitor star objects for solar flare events and records their magnitude. The policy keeps on running until you stop it. If you run the Generate Flare Event again, the policy continues to record the magnitudes of the flares in the logs.

What to do next

Monitor events on objects in the plugged-in technology by running workflows.

Monitor Plug-In Events by Using Workflows

Workflows can include a Wait Event element that suspends the workflow and waits for an event to occur in a plugged-in technology. Plug-ins can implement triggers and watchers to notify waiting workflows of the events that occur.

The solar system example includes a workflow that implements a Wait Event element to wait for solar flares. When a flare occurs, the waiting workflow resumes its run, records the magnitude of the flare in the logs, then ends.

Prerequisites

- Install the solar system plug-in in the Orchestrator server.
- Start the Orchestrator client.

Procedure

1. From the drop-down menu in the Orchestrator client, select **Design**.
2. Click the **Workflows** view.
3. Expand the **Samples > SolarSystem** nodes in the hierarchical list of workflows to see the list of workflows that the solar system plug-in adds to the library.
4. Right-click the Wait On Flare Event workflow, select **Start workflow**, and click **Submit**.

   The Wait On Flare Event workflow calls the `SolarSystemTriggerGenerator.createStarFlareTrigger()` method to create an event trigger.
5. Click the workflow token for this run of the Wait On Flare Event workflow.

   In the workflow schema, you can see that the workflow has suspended its run at the Waiting Event element. In the **Logs** tab you can see that the workflow is waiting for a flare of at least magnitude 10.
6. Click the **Inventory** view.
7. Expand the **SolarSystem** node in the hierarchical list of plug-ins.
8. Right-click **Helios** and select **Run workflow > Generate Flare Event**.

   Set the magnitude of the flare to a value greater than 10 when you run the workflow.
9. Click **Workflows**.
10. Click the workflow token for Wait On Flare Event workflow.

    The workflow is no longer waiting and has ended its run. In the **Logs** tab for this token you can see that the workflow has recorded a flare of at least magnitude 10.

The Wait On Flare Event workflow implements the solar system scripting API to create a plug-in trigger that waits for solar flares of a given magnitude. When a flare event occurs, the workflow ends, but you can create a loop in the workflow to record the event and wait for the next event.

What to do next

Access the solar system objects and workflows by using the solar system Web view.

Access Plug-In Objects and Operations by Using a Web View

With Web views, you can run workflows on objects from the Orchestrator inventory from a Web browser instead of from the Orchestrator client.

The solar system example includes a Web view that you can use to access the objects and workflows of the solar system plug-in from a Web browser.

Prerequisites

- Install the solar system plug-in in the Orchestrator server.
- Start the Orchestrator client.

Procedure

1. From the drop-down menu in the Orchestrator client, select **Administer**.
2. Click the **Web Views** view.
3. Right-click the **SolarSystem** Web view and select **Publish**.
4. Open a browser and go to `http://`*`orchestrator_server`*`:8280`.

   In the URL, *orchestrator_server* is the DNS name or IP address of the Orchestrator server, and 8280 is the default port number where Orchestrator publishes Web views.
5. On the Orchestrator home page, click **Web View List**.
6. Click **Solar System**.
7. Log in using your Orchestrator user name and password.
8. Click the buttons in the Web view to run workflows on the objects in the solar system application.

You can run workflows to add a planet to the Sun, modify, split, or remove planets from a Web browser. You can examine the structure and files of the solar system Web view in the source files of the solar system plug-in or exporting the Web view to a directory in **Web Views** in the Orchestrator client.

What to do next

You can adapt the classes of the solar system application and the plug-in implementation to experiment with plug-in development. You can use the solar system scripting API to develop more workflows that perform operations in the solar system application.

API Enhancements for Plug-In Development 4

Version 5.1 of Orchestrator introduces a number of new API features that you can use to simplify the plug-in development process. The list of new API features includes the use of annotations, Java-based configuration, Spring features, workflow and action generation, and SSL support.

This chapter includes the following topics:

- "Orchestrator Annotations API," on page 85
- "Orchestrator Spring-Based Plug-In API," on page 88
- "Orchestrator Workflow Generation API," on page 89
- "Orchestrator SSL Configuration API," on page 90

Orchestrator Annotations API

Annotations provide a way to define plug-in elements contained in the `vso.xml` file, without modifying the file directly. You can annotate the Java source files of the plug-in to define the finders and scripting objects to be included inside the `vso.xml` file.

You can use annotations and Java-based configuration to define all plug-in elements. See "Java-Based Configuration API for the Plug-In Definition File," on page 86.

Enable Annotation-Based Configuration

You can enable annotation-based configuration by adding the proper library dependencies to the build path of the plug-in.

The Orchestrator annotations are included inside the `o11n-plugin-sdk-tools.jar` file, but this library requires other libraries to run.

Procedure

1 Add the following libraries to the Ant *build.classpath* path variable.

```
<path id="build.classpath">
    ...
    <pathelement location="${maven.repo.local}/o11n-sdkapi.jar"/>
    <pathelement location="${maven.repo.local}/o11n-model.jar"/>
    <pathelement location="${maven.repo.local}/o11n-util.jar"/>
    <pathelement location="${maven.repo.local}/o11n-plugin-sdk-tools.jar"/>
    <pathelement location="${maven.repo.local}/o11n-plugin-sdk-plugen.jar"/>
    <pathelement location="${maven.repo.local}/commons-cli-1.2.jar"/>
    <pathelement location="${maven.repo.local}/commons-collections-3.2.1.jar"/>
    <pathelement location="${maven.repo.local}/commons-lang-2.6.jar" />
```

```
    <pathelement location="${maven.repo.local}/commons-logging-1.0.4.jar" />
    <pathelement location="${maven.repo.local}/spring-asm-3.1.0.RELEASE.jar"/>
    <pathelement location="${maven.repo.local}/spring-beans-3.1.0.RELEASE.jar"/>
    <pathelement location="${maven.repo.local}/spring-context-3.1.0.RELEASE.jar"/>
    <pathelement location="${maven.repo.local}/spring-core-3.1.0.RELEASE.jar"/>
    <pathelement location="${maven.repo.local}/spring-oxm-3.1.0.RELEASE.jar"/>
</path>
```

The `vso.xml` file generation process occurs at build time of the plug-in package, so the auxiliary libraries do not need to be packaged inside the plug-in.

2 Enable the `vso.xml` file generation.

   You can use Ant, to enable the generation of the `vso.xml` file by adding a new target that invokes the `VsoGenerator` class.

   The following is an example of an Ant target.

```
<target name="package" depends="compile,test" description="Package the application">
    <antcall target="generate-vso" />
    ...
</target>

<target name="generate-vso">
    <java fork="true" failonerror="yes"
classname="com.vmware.o11n.plugin.sdk.plugen.vso.VsoGenerator"
classpathref="build.classpath">
        <arg line="-name ${plugin.build.name}" />
        <arg line="-vsoDirectory ${maven.build.darDir}/VSO-INF" />
        <arg line="-moduleBuilder com.vmware.o11n.plugin.PowerShellModuleBuilder" />
    </java>
</target>
```

Annotating Objects

You can mark a domain class as an Orchestrator scripting object by using the `@VsoObject` annotation.

By default, the simple class name prefixed with your plug-in name will be used as a scripting object name. You can override this behavior by explicitly specifying the name attribute within the annotation.

You cannot export object properties as scripting attributes by using the `@VsoObject` annotation. To export a given property as a scripting attribute, you must annotate it with the `@VsoProperty` annotation. By default, the field name is used as a scripting attribute name. You can use `@VsoProperty` on a field or on a getter method level.

To generate the finder definition for a given scripting object, you must annotate your class with the `@VsoFinder` annotation.

Java-Based Configuration API for the Plug-In Definition File

Java-based configuration provides a way to define plug-in elements contained in the `vso.xml` file, without modifying the file directly. You can use a specific Java class to set the properties from the `vso.xml` file that are not directly related to finders and scripting objects.

You can use Java-based configuration and annotations to define all plug-in objects. See "Orchestrator Annotations API," on page 85.

Using Java-Based Configuration

You can use Java-based configuration for elements that are not supported by annotation-based configuration.

To connect fragments of the `vso.xml` file that are not connected to the annotated scripting objects, you must create your own `ModuleBuilder` class, which extends from the `com.vmware.o11n.plugin.sdk.module.ModuleBuilder` class, and to implement its `configure` method.

Example: Using the ModuleBuilder Class

The following code sample uses the `ModuleBuilder` class to generate elements in the `vso.xml` file.

```
...
public class CiscoModuleBuilder extends ModuleBuilder {

    private static final String UCSM_DATASOURCE = "ucsm-datasource";

    @Override
    public void configure() {
        module("UCSM")
            .withDescription("Cisco UCSM Plug-in.")
            .withImage("images/cisco_16x16.png")
            .basePackages("com.vmware.o11n.vmo.plugin.ucsm.model");

        configuration(CiscoUCSMConfigurationAdaptor.class, "images/cisco_16x16.png")
            .configurationWar("o11nplugin-ucsm-config.war").validatable();

        installation(InstallationMode.BUILD)
            .action(ActionType.INSTALL_PACKAGE,
                "packages/${artifactId}-package-${project.version}.package");

        inventory("System");

        finderDatasource(CiscoUCSMPluginAdaptor.class,
            UCSM_DATASOURCE).anonymousLogin(LoginMode.INTERNAL);
    }
}
```

You can use the annotation-based configuration method for enabling `vso.xml` file generation. See “Enable Annotation-Based Configuration,” on page 85.

Orchestrator Spring-Based Plug-In API

The Orchestrator Spring-based plug-in API offers boilerplate code that you can use to simplify plug-in development and add new features. You can implement additional features, such as scripting object lifecycle management, dependency injection, and basic resource management, by using the Spring-based plug-in API.

Spring-Based API Basic Configuration

You must extend some of the classes offered by the Spring-based API to start developing a plug-in based on the Spring API.

The Plug-In Adapter Implementation

```
public final class DemoPluginAdaptor extends AbstractSpringPluginAdaptor {
    private static final String DEFAULT_CONFIG = "com/vmware/o11n/plugin/demo/pluginConfig.xml";
    @Override
    protected ApplicationContext createApplicationContext(ApplicationContext defaultParent) {
        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext(new String[] { DEFAULT_CONFIG }, defaultParent);
        return applicationContext;
    }
}
```

The Plug-In Application Context Definition

You must define the following code inside the `pluginConfig.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <context:component-scan base-package="com.vmware.o11n.plugin.demo"
scoperesolver="com.vmware.o11n.plugin.sdk.spring.VsoAnnotationsScopeResolver">
    <context:include-filter type="annotation"
expression="ch.dunes.vso.sdk.annotation.VsoFinder"/>
    <context:include-filter type="annotation"
expression="ch.dunes.vso.sdk.annotation.VsoObject"/>
    </context:component-scan>
    <bean class="com.vmware.o11n.plugin.demo.DemoPluginFactory" id="pluginFactory" autowire-
candidate="false" scope="prototype" />
</beans>
```

The Plug-In Factory Implementation

```
public final class DemoPluginFactory extends AbstractSpringPluginFactory {
    @Override
    public Object find(InventoryRef ref) {
    }
    @Override
    public QueryResult findAll(String type, String query) {
    }
    @Override
    public List<?> findChildrenInRootRelation(String type, String relationName) {
    }
```

```
    @Override
    public List<?> findChildrenInRelation(InventoryRef parent, String relationName) {
    }
}
```

The plug-in factory is responsible to find objects. After implementing the plug-in factory, you do not need to implement the following methods.

- `public void registerEventPublisher(String type, String id, IPluginEventPublisher pluginEventPublisher)`
- `public void unregisterEventPublisher(String type, String id, IPluginEventPublisher pluginEventPublisher)`
- `public Object find(String type, String id)`
- `public List<?> findRelation(String type, String id, String relationName)`

Orchestrator Workflow Generation API

You can extend the Orchestrator functionality by creating workflows and actions based on external definitions. The plug-in SDK contains helper classes that allow basic workflow and action generation at runtime.

Generating Actions

You can generate actions by using the `ScriptModuleBuilder` class. The main attributes of an action are input parameters, return type, and script to be executed.

You must follow the steps for generating actions.

```
// Cretate instance of ScriptmoduleBuilder and set required action name
ScriptModuleBuilder builder = new ScriptModuleBuilder().setName(actionName);

// Set type of returned type (Optional)
builder .setResultType("string");

// Add input parameters, if any
builder.addParameter("SessionId", "string");

// Set script to be executed
builder.setScript("var a = 1");

// Persist generated action in vCO
builder.insert(categoryName, factory);
```

Generating Workflows

You can generate workflows by using the `WorkflowBuilderExt` class. The main attributes of a workflow are input and output parameters, attributes, and workflow item tasks. To have a complete workflow, you need to define links between workflow items, and optionally enhance the workflow presentation.

You must follow the steps for generating workflows.

```
// Cretate an instance of WorkflowBuilderExt
WorkflowBuilderExt wb = new WorkflowBuilderExt();

// Set generated workflow name
wb.setName(workflowName);
```

```
// Create workflow output parameter
wb.addInParameter("someParam", “string”);

// Create required workflow items and specify their location
wb.createEndItem(“endItem”, 50, 100);
ScriptingBoxItem item = wb.createScriptingBoxItem(“item1”, “var a =
someParam”).setLocation(50,50);
// In/Out parameters can be added, if needed
item.addInParameter(“someParam”, “someParam”, "string");
wb.bindItemInParameter(“item1”, “someParam”,“someParam”);

// Connect items to create real workflow
wb.connectItem(“item1”, “endItem”);

// Set workflow start item
wb.setRootItemName(“item1”);

// Persist workflow into vCO
wb.insertWorkflow(factory, targetFolder);
```

Orchestrator SSL Configuration API

SSL support provides a way to use the Orchestrator keystore to create secure connections based on SSL from plug-ins’ source code. You can use SSL to manage your own plug-in keystore and to implement your own classes to establish SSL connections between your plug-in and your configured hosts or services.

SSL Configuration Methods

You can configure SSL connections to use a specific connection method.

Plain Java URLConnection

If you want to use the Java standard `URLConnection` class, you have two options.

You can configure properties through the `HttpsURLConnection` class. The SDK provides the proper implementations for the standard Java interfaces `SSLSocketFactory` and `HostnameVerifier`. After configuring the properties, you can create underlying secure connections from any URL.

```
import java.net.URLConnection;
import javax.net.ssl.HttpsURLConnection;
import com.vmware.o11n.plugin.sdk.ssl.factory.PluginSSLSocketFactory;
import com.vmware.o11n.plugin.sdk.ssl.verifier.PluginHostnameVerifier;

...

// Initialization
HttpsURLConnection.setDefaultSSLSocketFactory(PluginSSLSocketFactory.getDefault());
// Optionally
HttpsURLConnection.setDefaultHostnameVerifier(new PluginHostnameVerifier());

...

URLConnection conn = new URL("https://...").openConnection();
...
```

You can create underlying secure connections directly from the `PluginSSLSocketFactory` class with the `PluginHostnameVerifier` class configured by default.

```
import java.net.URLConnection;
import com.vmware.o11n.plugin.sdk.ssl.factory.PluginSSLSocketFactory;

...

URLConnection conn = PluginSSLSocketFactory.getConnection("https://...");
...
```

Plain Java SSLSocketFactory

If you want to use the Java standard `SSLSocketFactory` class directly, the SDK provides the proper implementation to create underlying secure sockets.

```
import java.net.Socket;
import javax.net.ssl.SSLSocketFactory;
import com.vmware.o11n.plugin.sdk.ssl.factory.PluginSSLSocketFactory;

...

SSLSocketFactory factory = PluginSSLSocketFactory.getDefault();
Socket s = factory.createSocket(...);
...
```

Apache's HttpClient 3

If you want to use Apache HttpClient version 3, you must register the secure protocol that you want to use with your own implementation of their `ProtocolSocketFactory` interface. In the following examples, the SDK provides the Orchestrator implementation of that interface for you and two different ways to register the HTTPS protocol.

You can register the protocol manually.

```
import com.vmware.o11n.plugin.sdk.ssl.factory.HttpClient3PluginSSLSocketFactory;
import org.apache.commons.httpclient.protocol.Protocol;
import org.apache.commons.httpclient.protocol.ProtocolSocketFactory;

...

Protocol https = new Protocol("https", (ProtocolSocketFactory)
HttpClient3PluginSSLSocketFactory.getDefault(), 443);
Protocol.registerProtocol("https", https);
```

You can use `HttpClient3PluginSSLSocketFactory` to register the HTTPS protocol.

Note You should use the manual method to register other protocols or specific ports.

```
import com.vmware.o11n.plugin.sdk.ssl.factory.HttpClient3PluginSSLSocketFactory;

...

HttpClient3PluginSSLSocketFactory.registerHttpsProtocol();
```

Apache's HttpClient 4

If you want to use Apache HttpClient version 4, you must configure the `ClientConnectionManager` that you want to use, with the secure scheme that you want, and your own implementation of their `SSLSocketFactory` interface. In the following example, the SDK provides the Orchestrator implementation of that interface for you.

```
import com.vmware.o11n.plugin.sdk.ssl.factory.HttpClient4PluginSSLSocketFactory;
import org.apache.http.conn.scheme.Scheme;
import org.apache.http.conn.scheme.SchemeRegistry;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.impl.conn.SchemeRegistryFactory;
import org.apache.http.impl.conn.tsccm.ThreadSafeClientConnManager;

...

SchemeRegistry registry = SchemeRegistryFactory.createDefault();

Scheme https = new Scheme("https", 443, HttpClient4PluginSSLSocketFactory.getDefault());
registry.register(https);

ThreadSafeClientConnManager manager = new ThreadSafeClientConnManager(registry);
DefaultHttpClient client = new DefaultHttpClient(manager);
```

The HostValidator Helper Class

The `HostValidator` class can be used by plug-ins to retrieve, from an HTTPS URL, certificates used to establish a connection. The SDK provides the `HostValidator` helper class independently of the SSL configuration.

By using the `HostValidator` helper class, a plug-in can return information about certificates, that can be accepted through a user interaction. If the certificate information is accepted, the `HostValidator` class installs the certificate within the Orchestrator keystore.

You can use the features of the `HostValidator` helper class through the following methods.

- The constructor `HostValidator(url)`, for which the URL is the HTTPS URL.
- The method `getCertificateInfo()`, which returns a map with properties available for an HTTPS connection and its cerfiticates.
- The method `installCertificates()`, which installs certificates within the Orchestrator keystore.

Orchestrator Plug-In API Reference 5

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.

This chapter includes the following topics:

- "IAop Interface," on page 94
- "IConfigurationAdaptor Interface," on page 94
- "IDynamicFinder Interface," on page 95
- "IPluginAdaptor Interface," on page 95
- "IPluginEventPublisher Interface," on page 96
- "IPluginFactory Interface," on page 97
- "IPluginNotificationHandler Interface," on page 97
- "IPluginPublisher Interface," on page 98
- "WebConfigurationAdaptor Interface," on page 98
- "BaseAction Class," on page 99
- "ConfigurationError Class," on page 99
- "PluginLicense Class," on page 99
- "PluginTrigger Class," on page 100
- "PluginWatcher Class," on page 101
- "QueryResult Class," on page 101
- "SDKFinderProperty Class," on page 102
- "SDKHelper Class," on page 103
- "PluginExecutionException Class," on page 104
- "PluginLicenseException Class," on page 104
- "PluginOperationException Class," on page 104
- "ConfigurationError.Severity Enumeration," on page 105
- "ErrorLevel Enumeration," on page 105
- "HasChildrenResult Enumeration," on page 106

- "ScriptingAttribute Annotation Type," on page 107
- "ScriptingFunction Annotation Type," on page 107
- "ScriptingParameter Annotation Type," on page 108

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
`get(java.lang.String propertyName, java.lang.Object object, java.lang.Object sdkObject)`	`java.lang.Object`	Obtains a property from a given object in the plug-in.
`set(java.lang.String propertyName, java.lang.String propertyValue, java.lang.Object object)`	Void	Sets a property on a given object in the plug-in.

IConfigurationAdaptor Interface

The `IConfigurationAdaptor` interface allows you to add a tab in the Orchestrator configuration interface. You can use the tab to configure a plug-in.

You can extend the `IConfigurationAdaptor` interface to pass to the plug-in configuration information that is specific to your environment. The `SDKHelper` class defines further methods to pass configuration information from the configuration interface to the Orchestrator server.

The `IConfigurationAdaptor` interface is contained in the `ch.dunes.vso.sdk.conf` package.

The `IConfigurationAdaptor` interface defines the following methods.

Method	Returns	Description
`loadConfiguration(java.io.InputStream stream)`	Void	Loads or reloads the configuration. If the `stream` property is null, the plug-in loads its default configuration. Returns `java.io.IOException` if it encounters an error.
`saveConfiguration(java.io.OutputStream stream)`	Void	Saves the configuration details. If the `stream` property is null, the plug-in saves the configuration details in the default location when you click **Apply Changes** in the configuration interface. Returns `java.io.IOException` if it encounters an error.
`setPluginName(java.lang.String name)`	Void	Sets the plug-in name as it appears in the plug-in tab in the configuration interface.
`validateConfiguration()`	`ConfigurationError[]`	Validates the configuration if `validation="enabled"` is set.

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
`getIdAccessor(java.lang.String type)`	`java.lang.String`	Provides an OGNL expression to obtain an object ID programmatically.
`getProperties(java.lang.String type)`	`java.util.List<SDKFinderProperty>`	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 resonsible for session management. The `IPluginAdaptor` Interface defines the following methods.

Method	Returns	Description
`addWatcher(PluginWatcher watcher)`	Void	Adds a watcher to monitor for a specific event
`createPluginFactory(java.lang.String sessionID, java.lang.String username, java.lang.String password, IPluginNotificationHandler notificationHandler)`	`IPluginFactory`	Creates an `IPluginFactory` instance. The Orchestrator server uses the factory to obtain objects from the plugged-in technology by their ID, by their relation to other objects, and so on. The session ID allows you to identify a running session. For example, a user could log into two different Orchestrator clients and run two sessions simultaneously. Similarly, starting a workflow creates a session that is independent from the client in which the workflow started. A workflow continues to run even if you close the Orchestrator client.
`installLicenses(PluginLicense[] licenses)`	Void	Installs the license information for standard plug-ins that VMware provides
`registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)`	Void	Sets triggers and gauges on an element in the inventory
`removeWatcher(java.lang.String watcherId)`	Void	Removes a watcher
`setPluginName(java.lang.String pluginName)`	Void	Gets the plug-in name from the `vso.xml` file
`setPluginPublisher(IPluginPublisher pluginPublisher)`	Void	Sets the publisher of the plug-in

Method	Returns	Description
`uninstallPluginFactory(IPluginFactory plugin)`	Void	Uninstalls a plug-in factory.
`unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)`	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
`pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)`	Void	Publish a gauge for policies to monitor. Takes the following parameters: ■ `type`: Type of the object to monitor. ■ `id`: Identifier of the object to monitor. ■ `gaugeName`: Name for this gauge. ■ `deviceName`: Name for the type of attribute that the gauge monitors. ■ `gaugeValue`: Value for which the gauge monitors the object.
`pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)`	Void	Publish a trigger for policies to monitor. Takes the following parameters: ■ `type`: Type of the object to monitor. ■ `id`: Identifier of the object to monitor. ■ `triggerName`: Name for this trigger. ■ `additionalProperties`: 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
`executePluginCommand(java.lang.String cmd)`	Void	Use the plug-in to run a command. VMware recommends that you do not use this method.
`find(java.lang.String type, java.lang.String id)`	`java.lang.Object`	Use the plug-in to find an object. Identify the object by its ID and type.
`findAll(java.lang.String type, java.lang.String query)`	`QueryResult`	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, `findAll()` returns all objects of the specified type.
`findRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)`	`java.util.List`	Determines whether an object has children.
`hasChildrenInRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)`	`HasChildrenResult`	Finds all children related to a given parent by a certain relation.
`invalidate(java.lang.String type, java.lang.String id)`	Void	Invalidate objects by type and ID.
`void invalidateAll()`	Void	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
`getSessionID()`	`java.lang.String`	Returns the current session ID
`notifyElementDeleted(java.lang.String type, java.lang.String id)`	Void	Notifies the system that an object with the given type and ID has been deleted

Method	Returns	Description
`notifyElementInvalidate(java.lang.String type, java.lang.String id)`	Void	Notifies the system that an object's relations have changed. You can use the `notifyElementInvalidate()` 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.
`notifyElementUpdated(java.lang.String type, java.lang.String id)`	Void	Notifies the system that an object's attributes have been modified
`notifyMessage(ch.dunes.vso.sdk.api.ErrorLevel severity, java.lang.String type, java.lang.String id, java.lang.String message)`	Void	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
`pushWatcherEvent(java.lang.String id, java.util.Properties properties)`	Void	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
`getWebAppContext()`	String	Locates the WAR file of the Web application for the configuration tab. Provide the name and path to the WAR file from the `/webapps` directory in the DAR file as a string.
`setWebConfiguration(boolean webConfiguration)`	Boolean	Determine whether the contents of the configuration tab are defined by a Web application.

BaseAction Class

The `BaseAction` class is a helper class that you can use to create Orchestrator actions.

In the context of creating a plug-in configuration tab, the `BaseAction` class provides methods that you can implement to set up and run the configuration action that pushes configuration information to the Orchestrator server from the configuration interface.

The `BaseAction` class is contained in the `ch.dunes.vso.configuration.web.commons` package.

The `BaseAction` class defines the following methods:

Method	Returns	Description
`prepare()`	Void	Implement this method to instantiate the configuration adapter and load configuration information.
`execute()`	Void	Implement this method to push the configuration information to the configuration server.

ConfigurationError Class

The `ConfigurationError` class defines the error objects that the `IConfigurationAdaptor.validateConfiguration()` method returns the plug-in configuration contains errors.

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

The `ConfigurationError` class uses the `ConfigurationError.Severity` enumeration and defines the following fields:

- `public ConfigurationError.Severity severity`
- `public java.lang.String title`
- `public java.lang.String description`

Constructor

```
ConfigurationError(ConfigurationError.Severity severity, java.lang.String title, java.lang.String
description)
```

PluginLicense Class

The `PluginLicense` class obtains and sets any licensing information that a plug-in requires.

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

The `PluginLicense` class defines the following methods.

Method	Returns	Description
`getDescription()`	`java.lang.String`	Obtains the license description.
`getLicenseString()`	`java.lang.String`	Obtains the license key.
`getOwner()`	`java.lang.String`	Obtains the license owner.

Method	Returns	Description
setDescription(java.lang.String description)	Void	Sets the license description.
setLicenseString(java.lang.String licenseString)	Void	Sets the license key.
setOwner(java.lang.String owner)	Void	Obtains the license owner.

Constructor

PluginLicense()

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
`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

Method	Returns	Description
`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 `IDynanmicFinder.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

Method	Returns	Description
`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
`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)
```

SDKHelper Class

You can add a tab to the Orchestrator configuration interface to allow users to configure a plug-in. The `SDKHelper` class provides methods to obtain configuration information for a plug-in from the Orchestrator configuration interface.

The `SDKHelper` class is contained in the `ch.dunes.vso.sdk.helper` package.

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

The `SDKHelper` class defines the following methods.

Method	Returns	Description
`getConfigurationPathForPluginName(java.lang.String moduleName)`	`java.lang.String`	Obtains the path to the source files of the plug-in implementation.
`isPluginEnabled(java.lang.String pluginName)`	Boolean	Checks whether the plug-in is enabled or disabled.
`setPluginEnabled(java.lang.String pluginName, boolean flag)`	Void	Enables the plug-in.
`loadPropertiesForPluginName(java.lang.String moduleName)`	`java.util.Properties`	Loads a list of properties that users set in the configuration interface.
`savePropertiesForPluginName(java.util.Properties properties, java.lang.String moduleName)`	Void	Saves in the plug-in properties that the user sets in the configuration interface.
`getPluginInstallCredentials()`	`java.lang.String[]`	Obtains the credentials of the user who sets properties in the configuration interface.

Constructor

```
SDKHelper()
```

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`, `toStringfillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`

Constructor

```
PluginExecutionException(java.lang.String message)
```

PluginLicenseException Class

The `PluginLicenseException` class returns an error message if the plug-in encounters an exception when it installs a license for a plug-in.

```
public class PluginLicenseException
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`, `toStringfillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`

Constructor

```
PluginLicenseException(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)
```

ConfigurationError.Severity Enumeration

The `ConfigurationError` class uses the `ConfigurationError.Severity` enumeration to set the level of severity of configuration errors.

```
public static enum ConfigurationError.Severity
extends java.lang.Enum<ConfigurationError.Severity>
implements java.io.Serializable
```

The `ConfigurationError.Severity` enumeration defines the following constant values for error levels:

- `public static final ConfigurationError.Severity Info`
- `public static final ConfigurationError.Severity Warning`
- `public static final ConfigurationError.Severity Error`

The `ConfigurationError.Severity` enumeration defines the following methods.

Method	Returns	Description
`values()`	`public static ConfigurationError.Severity[]`	Returns an array containing the constants of this enumeration type, in the order that the plug-in declares them. You can use this method to iterate through the constants as follows: `for ( ConfigurationError.Severity c : ConfigurationError.Severity.values()) System.out.println(c);`
`valueOf(java.lang.String name)`	`java.lang.String name`	Returns the constant value of an enumeration with the specified name. The string must match an identifier that you use to declare an enumeration constant in this type. Extraneous whitespace characters are not permitted. The `name` parameter is the name of the enumeration constant to return.
`public int getValue()`	`int`	Returns the value of the error.

The `ConfigurationError.Severity` enumeration inherits the following methods from `class java.lang.Enum`:

`clone`, `compareTo`, `equals`, `finalize`, `getDeclaringClass`, `hashCode`, `name`, `ordinal`, `toString`, `valueOf`

ErrorLevel Enumeration

The ErrorLevel enumeration defines constant values for different levels of error that a plug-in encounters.

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

The `ErrorLevel` enumeration defines the following constant values for error levels:

- `public static final ErrorLevel Fatal`
- `public static final ErrorLevel Error`

- `public static final ErrorLevel Warning`
- `public static final ErrorLevel Info`
- `public static final ErrorLevel Debug`

The `ErrorLevel` enumeration defines the following methods:

Method	Returns	Description

`values()`	`static ErrorLevel[]`	Returns an array containing the constants of this enumeration type, in the order that plug-in declares them. You can use this method to iterate through the constants as follows:
		`for (ErrorLevel c : ErrorLevel.values())`
		`System.out.println(c);`

`valueOf(java.lang.String name)`	`java.lang.String`	Returns the constant value of an enumeration with the specified name. The string must match an identifier that you use to declare an enumeration constant in this type. Extraneous whitespace characters are not permitted. The `name` parameter is the name of the enumeration constant to return.
`getSeverity()`	`ErrorLevel`	Returns the `ErrorLevel` value of the error.

The `ErrorLevel` enumeration inherits the following methods from `class java.lang.Enum`:

`clone`, `compareTo`, `equals`, `finalize`, `getDeclaringClass`, `hashCode`, `name`, `ordinal`, `toString`, `valueOf`

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
getValue()	int	Returns one of the following values: **1** Parent has children **-1** Parent has no children **0** Unknown, or invalid parameter
valueOf(java.lang.String name)	static HasChildrenResult	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.
values()	static HasChildrenResult[]	Returns an array containing the constants of this enumeration type, in the order they are declared. This method can iterate over constants as follows: for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);

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 6

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.

This chapter includes the following topics:

- "module Element," on page 110
- "configuration Element," on page 111
- "description Element," on page 112
- "deprecated Element," on page 112
- "url Element," on page 112
- "installation Element," on page 113
- "action Element," on page 113
- "webview-components-library Element," on page 113
- "finder-datasources Element," on page 114
- "finder-datasource Element," on page 114
- "inventory Element," on page 115
- "finders Element," on page 115
- "finder Element," on page 116
- "properties Element," on page 117
- "property Element," on page 117
- "relations Element," on page 118
- "relation Element," on page 118
- "id Element," on page 118
- "inventory-children Element," on page 119
- "relation-link Element," on page 119

- "events Element," on page 119
- "trigger Element," on page 119
- "trigger-properties Element," on page 120
- "trigger-property Element," on page 120
- "gauge Element," on page 120
- "scripting-objects Element," on page 121
- "object Element," on page 121
- "constructors Element," on page 122
- "constructor Element," on page 122
- "Constructor parameters Element," on page 122
- "Constructor parameter Element," on page 122
- "attributes Element," on page 123
- "attribute Element," on page 123
- "methods Element," on page 124
- "method Element," on page 124
- "example Element," on page 125
- "code Element," on page 125
- "Method parameters Element," on page 125
- "Method parameter Element," on page 125
- "singleton Element," on page 126
- "enumerations Element," on page 126
- "enumeration Element," on page 126
- "entries Element," on page 127
- "entry Element," on page 127

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 `<finder>` 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.

Attributes	Value	Description
`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 6-1. Element Hierarchy

Parent Element	Child Elements
None	■ `<description>` ■ `<installation>` ■ `<configuration>` ■ `<webview-components-library>` ■ `<finder-datasources>` ■ `<inventory>` ■ `<finders>` ■ `<scripting-objects>` ■ `<enumerations>`

configuration Element

The `<configuration>` element allows you to add a tab in the Orchestrator configuration interface. You can use the tab to configure a plug-in.

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

Attributes	Value	Description
`icon`	Image file	Icon that represents the plug-in in the Orchestrator configuration interface. Mandatory attribute.
`adaptor-class`	Java class	Implementation of the `IConfigurationAdaptor` Java interface that defines the actions to perform in the configuration interface. Mandatory attribute.
`configuration-war`	WAR archive	Web application archive (`war` file) that contains the components of the Web application that implements the adapter class. The pages of the Web application appear in the configuration interface. Optional attribute.
`validation`	`enabled` or `disabled`	Validates the configuration against a function that you define in the adapter class. Optional attribute.

Table 6-2. Element Hierarchy

Parent Element	Child Element
<module>	None

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 6-3. Element Hierarchy

Parent Elements	Child Elements
■ <module> ■ <example> ■ <trigger> ■ <gauge> ■ <finder> ■ <constructor> ■ <method> ■ <object> ■ <enumeration>	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 6-4. Element Hierarchy

Parent Elements	Child Elements
■ <method> ■ <object>	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 6-5. Element Hierarchy

Parent Elements	Child Elements
■ <enumeration> ■ <object>	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`	Setting the `mode` value results in the following behavior when the Orchestrator server starts: ■ The action `always` runs ■ The action `never` runs ■ The action runs when the server detects a newer version of the plug-in Mandatory attribute.

Table 6-6. 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 6-7. Element Hierarchy

Parent Element	Child Elements
`<installation>`	None

webview-components-library Element

The `<webview-components-library>` element points to a JAR file containing custom Web view tapestry components that extend Web view capabilities.

The `<webview-components-library>` element is optional. The `<webview-components-library>` element has the following attributes.

Attributes	Value	Description
jar	String	A JAR file containing Web view components. Mandatory attribute.
specification-path	String	The path in the JAR file to the Tapestry component definition file, in the lib folder of the *.dar file. The path must begin with a forward slash (/). Mandatory attribute.

Table 6-8. Element Hierarchy

Parent Element	Child Elements
<module>	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.

Attributes	Value	Description
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.
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`	`<finder-datasource name>` 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 `IDynamicFinder` instance, to return the ID and properties of a finder programmatically, instead defining it in the `vso.xml` 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`	`<scripting-object type>` attribute	The `<scripting-object>` type, if any, to which to map this finder. Optional attribute.

Table 6-13. Element Hierarchy

Parent Element	Child Elements
`<finders>`	■ `<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	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. You set one or the other of bean-property or property-accessor, but not both. Optional attribute.
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
<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 6-21. Element Hierarchy

Parent Element	Child Elements
<finder>	■ <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 6-22. Element Hierarchy

Parent Element	Child Elements
<events>	■ <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 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," on page 21 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>	■ <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>	■ <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
<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 6-32. 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 exposes.

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.
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 parenthesis, 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.
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>	■ <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>	■ <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
<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 6-38.

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 6-39. 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 6-40. Element Hierarchy

Parent Element	Child Element
<object>	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
<module>	<enumeration>

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
<enumerations>	■ <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 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 7

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 on page 129

  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 on page 131

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

- Plug-In Implementation on page 134

  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 on page 138

  You can consider certain recommendations when developing the different components of your Orchestrator plug-ins.

- Documenting Plug-In User Interface Strings and APIs on page 140

  When you write user interface (UI) strings for vCO plug-ins and the related API documentation, it is best practice to 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.

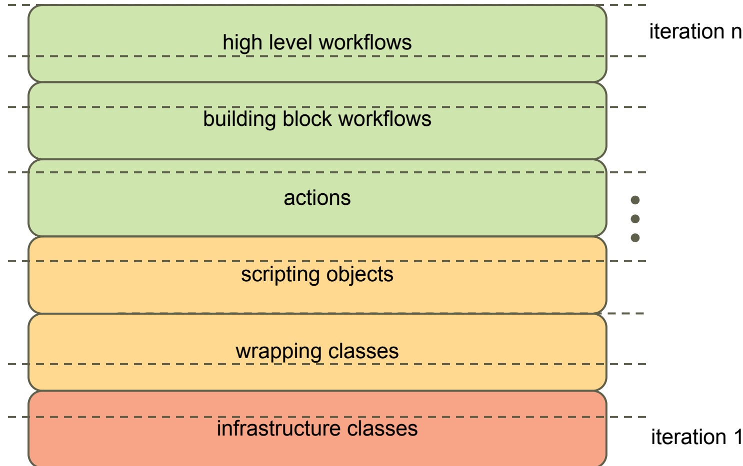
For information about plug-in layers, see "Structure of an Orchestrator Plug-In," on page 12.

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 7-1. 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.

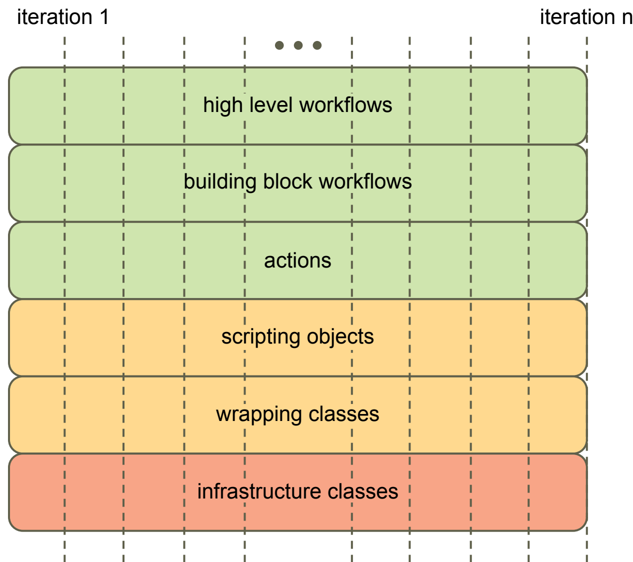
The bottom-up development process is considered good enough for small plug-ins, with reduced or non-existent set of wrapping classes, scripting objects, actions, or workflows.

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 7-2. 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.

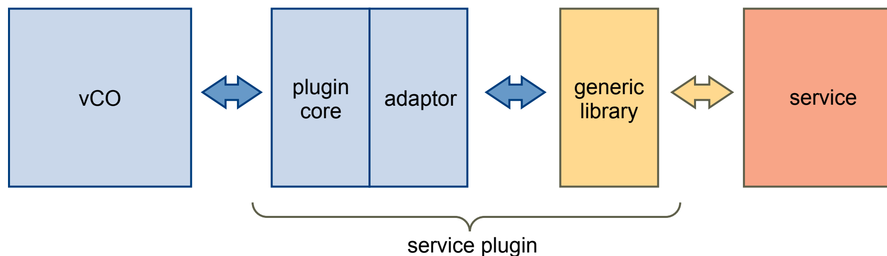
Types of Orchestrator Plug-Ins

Using plug-ins, you can integrate in Orchestrator general-purpose libraries or utilities like XML or SSH as well as entire systems such as vCloud. Depending on the technology that you integrate in 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 vCenter Orchestrator.

Figure 7-3. Archnitectire 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 Allows you to use any database within a workflow.

Mail plug-in Allows you to send emails within a workflow.

SSH plug-in Allows you to open SSH connections and run commands within a workflow.

XML plug-in Allows you to manage XML documents within a workflow.

Plug-ins for services have the following characteristics.

Complexity Plug-ins for services have from low to medium level 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 a basic set of classes that are the same 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 vCO inventory.

Plug-Ins for Systems

Plug-ins for systems connect the vCO 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 Allows you to manage vCenter Server instances using workflows.

vCloud Director plug-in Allows you to interact with a vCloud Director installation within a workflow.

Cisco UCSM plug-in Allows you to interact with Cisco entities within a workflow.

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

Complexity Plug-ins for systems have higher level of complexity, 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 the same functionality as that system 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 from 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 vCO. 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, vCenter Server plug-in provides such a tool.

- Plug-Ins for Object-Oriented Systems on page 133

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

- Plug-Ins for Resource-Oriented Systems on page 133

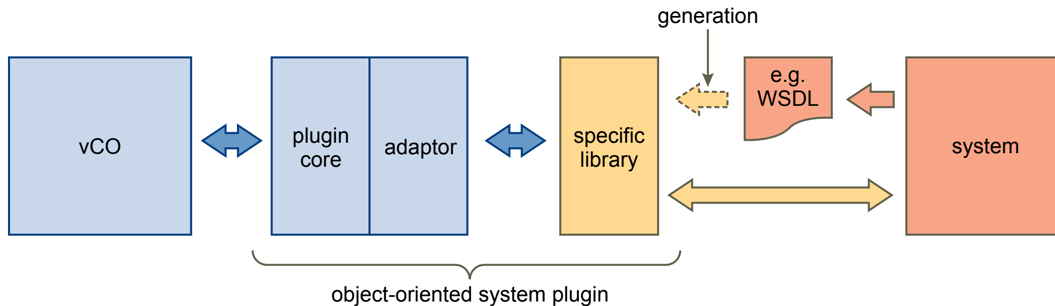
  Resource-oriented systems provide an interaction mechanism that is based on resources and simple operations that use HTTP methods.

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 7-4. Plug-Ins for Object-Oriented Systems



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

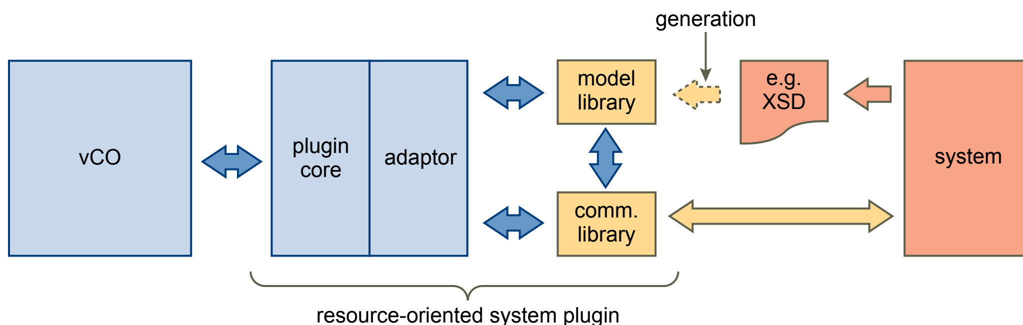
- If you use SOAP, you can use the WSDL file to generate a set of classes that combine the object model and the communication mechanism.
- This object model is almost everything that you have to expose inside vCenter Orchestrator.

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 7-5. 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, like 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 to interact 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 vCO. This could add some extra 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 on page 134

  You can apply a standard structure for the projects of your Orchestrator plug-ins.
- Project Internals on page 135

  You can apply certain approaches when implementing your plug-in, for example, cash objects, bring object in background, clone object, and so on. By applying such approaches, you can improve the performance of your plug-ins, avoid concurrency problems, and improve the responsiveness of the Orchestrator client.
- Workflow Internals on page 136

  You can implement a workflow to monitor long-time operations that your vCO plug-in performs.
- Workflows and Actions on page 136

  To ease the workflow development and usage, you can use certain good practices.
- Workflow Presentation on page 137

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

Table 7-1. Structure of a plug-in project

Module	Description
`/myAwesomePlugin-plugin`	Root of the plug-in project.
`/o11nplugin-myAwesomePlugin`	Module that composes the final plug-in DAR file.
`/o11nplugin-myAwesomePlugin-config`	Module that contains the plug-in configuration Web application. It generates a standard WAR file.
`/o11nplugin-myAwesomePlugin-core`	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.

Table 7-1. Structure of a plug-in project (Continued)

Module	Description
`/o11nplugin-myAwesomePlugin-model`	Module that contains all the classes that help you to 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`	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, cash objects, bring object in background, clone object, and so on. By applying such approaches, you can improve the performance of your plug-ins, avoid concurrency problems, and improve the responsiveness of the Orchestrator client.

Cache Objects if Possible

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 cash 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 will be 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 you 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 they provide the minimal information that you have to show in the inventory, like 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. If you have a cache and you always return the same instance of an object to every entity that requests it, then unwanted effects might occur. 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 will not be reverted. In such 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 you clone objects simultaneously. The biggest one is that the object that a using entity views might not be the last 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`, `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 could 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 as 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 could be the cause of strange *validation error* failures when a workflow is running. For example, between two nodes of a workflow, the `find` method will be 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 any runtime exception, probably you will get no more information than a *validation error* in the vCenter 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 vCO plug-in performs.

You should 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 have to 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 pooling 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 own complex workflows.

Create Higher-Level Workflows Based on Smaller Components

If you have to develop a complex workflow with lots of inputs and internal steps, you can split it in 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 of 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 7-2. Properties for Workflow Inputs

Properties	Usage
`Show in Inventory`	Use this property to help the user to run a workflow from the inventory view by clicking with the right button on it.
`Specify a root object to be shown in the chooser`	Use this property to help the user to choose inputs easily. If the root object can be refreshed in the presentation, or it is an attribute of an attribute of, or it 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.

Table 7-2. Properties for Workflow Inputs (Continued)

Properties	Usage
`Minimum string length`	Use this property to avoid empty strings from the testing tools.
`Custom validation`	Non-simple validations must be implemented with actions.

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

Recommendations for Orchestrator Plug-In Development

You can consider certain recommendations when developing the different components of your Orchestrator plug-ins.

Table 7-3. 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, or deletion of a scripting object, and for every other method that is available on the 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 break.
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 describes what the workflow does instead of how it works.
	Presentation property `mandatory input`	You need to 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 need to set the `Show in inventory` property to 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.

Table 7-3. Useful practices in plug-in implementation (Continued)

Component	Item	Description
	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 avoid breaking solutions that are built on previous versions of the workflow.
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 mass getting objects.
	Child discovery	If you need to retrieve object children separately, the retrieval process must be multithreaded and nonblocking on a single error.
	vCenter 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 vCO. 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 between objects. These are typically the properties that are present in the user interface.
Scripting object	Implementation	The `equals` method must be implemented to insure that == operation works on the same object as in some cases the object may 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 `Server.find`.
		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 collision occurs when you are orchestrating more than one server.

Table 7-3. Useful practices in plug-in implementation (Continued)

Component	Item	Description
	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 vCO server has a `Server.FindForId` 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 vCO may have policies triggered on various events. For example, a trigger or an event in the vCenter plug-in on the `Datacenter` object could be monitored by vCenter Orchestrator to determine when a new virtual machine is added, powered on, powered off, and so on.
	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 `moref` (managed object reference ID).
	Session manager	If you are connecting to a remote server that has a possibility for 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	Plug-in version should follow the 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 `no xyz method / property` on an object. Instead, methods should return `null` when no properties are available and be documented with details when these properties are not available.
	`vso.xml`	All objects, methods, and properties must be documented in `vso.xml`.

Documenting Plug-In User Interface Strings and APIs

When you write user interface (UI) strings for vCO plug-ins and the related API documentation, it is best practice to follow the accepted rules of style and format.

General Recommendations

- Use the official names for any of 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
virtual machine	VM

- End all complete sentences 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. The enclosing folders and packages should include a small description 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 by including the descriptions inside a `<![CDATA[insert your description here!]]>` tag.
- Use the standard Javadoc style for the Java source files.

Index

A

advices **136**
API
  action generation **89**
  annotate objects **86**
  annotation-based configuration **85**
  annotations **85**
  enhancements **85**
  generating actions **89**
  generating workflows **89**
  Java-based configuration **86**
  Java-based configuration usage **87**
  Spring-based basic configuration **88**
  Spring-based plug-in API **88**
  SSL **90**
  SSL configuration **90**
  SSL HostValidator helper class **92**
  workflow generation **89**
API documentation **140**

B

best practices **129**

H

HasChildrenResult Enumeration **106**

I

IConfigurationAdaptor interface **94**
IDynamicFinder interface **95**
IPluginAdaptor interface **15, 56, 57, 95**
IPluginEventPublisher interface **44, 96**
IPluginFactory **31**
IPluginFactory interface **16, 57, 97**
IPluginNotificationHandler **97**
IPluginPublisher interface **50, 98**

J

JavaScript API
  adding functions **55**
  adding objects **55**

N

new features **85**

P

plug-in adapter, creating **15**
plug-in API
  HasChildrenResult Enumeration **106**
  IDynamicFinder interface **95**
  IPluginAdaptor interface **95**
  IPluginEventPublisher interface **96**
  IPluginFactory interface **97**
  IPluginNotificationHandler **97**
  IPluginPublisher interface **98**
  PluginExecutionException **104**
  PluginLicenseException **104**
  PluginOperationException **104**
  PluginTrigger **100**
  PluginWatcher **101**
  QueryResult **101**
  ScriptingAttribute annotation **107**
  ScriptingFunction annotation **107**
  ScriptingParameter annotation **108**
  SDKFinderProperty class **102**
plug-in development
  best practices **129**
  bottom-up **129**
  top-down **130**
plug-in factory, creating **16**
plug-in implementation
  project internals **135**
  project structure **134**
  workflow internals **136**
  workflow presentation **137**
plug-in strings **140**
plug-in structure **12**
plug-in types
  plug-ins for services **131**
  plug-ins for systems **132**
plug-ins
  access from Web view **83**
  adapter **15, 26, 56, 60**
  add configuration tab **62, 63, 65, 67, 69**
  adding to JavaScript API **55**
  architecture **11**
  BaseAction class **65, 99**
  building approaches **129**
  components **13**
  configuration **111**

configuration action **61**
configuration adapter **61–63, 65**
configuration tab **103**
ConfigurationError class **99, 105**
ConfigurationError.Severity enumeration **105**
contents **19**
contents of DAR **78**
create DAR **77**
create event generator **41, 42**
create event publishers **43**
create scripting singleton **55**
create watchers **50, 52**
create workflow triggers **47, 48**
creating **25**
creating workflow triggers **46, 48**
DAR archive **22**
DAR file **77**
define finders **72**
enumerations **74**
ErrorLevel enumeration **105**
event handlers **17**
event listeners **33, 38**
event notifications **40**
example application **27**
expose external API **13**
factory **16, 26, 31–37**
find objects by identifier **34**
find objects by relation **36**
find objects by type **35**
find() method **34**
findAll() method **35**
finder objects **16, 17**
findRelation() method **36**
gauges **43, 44**
hasChildrenInRelation() method **36, 37**
IAop interface **94**
IConfigurationAdaptor interface **61–63, 65**
implementing notification handlers **39**
installation **79**
instantiate factory **57**
interact with plugged-in technology **80**
IPluginEventPublisher interface **41, 43, 44**
IPluginFactory interface **55**
IPluginNotificationHandler interface **38–40**
IPluginPublisher interface **50, 52**
JAR files **22**
listeners **17**
manage events **59**
mapping classes **75**
mapping methods **75**
monitor events **82, 83**
monitor object properties **48**
naming objects **21**
notification handling **33**
obtain configuration from user **65**
parts of a plug-in **11**
PluginLicense class **99**
PluginTrigger class **48**
PluginWatcher class **50, 52**
polices **82**
policies **17**
policy gauges **17**
policy triggers **17**
publish events **59**
publish watchers **52**
push events **44**
registering event listeners **39**
role of vso.xml file **14**
run workflows on objects **81**
SDKHelper class **61–63, 65, 103**
set up adapter **57**
set up factory **32**
solar system DAR file **78**
solar system finder mappings **74**
solar system JavaScript mappings **76**
solar system WAR file **69**
SolarSystemEventListener class **38**
structure **19**
Struts framework **65, 67**
triggers **43, 83**
using the solar system plug-in **80**
view scripting objects **80**
vso.xml **74**
vso.xml file **22, 70**
waiting workflows **83**
WAR file **22**
watchers **17, 50**
WebConfigurationAdaptor interface **98**
workflow triggers **17, 46, 47**

plug-ins for systems
object-oriented systems **133**
resource-oriented systems **133**
plug-ins, add configuration tab **61**
plug-ins, build solar system DAR **78**
plug-ins, CelestialBody.java **28**
plug-ins, errors **105**
plug-ins, ISolarSystemListener.java **29**
Plug-ins, Moon.java **29**
plug-ins, Planet.java **28**
plug-ins, scripting objects **17**
plug-ins, SolarSystemEventHandler.java **29**
plug-ins, SolarSystemRepository.java **30**

plug-ins, Star.java **28**
plug-ins, watchers **60**
PluginExecutionException **104**
PluginLicenseException **104**
PluginOperationException **104**
PluginTrigger **48, 100**
PluginTrigger class **46**
PluginWatcher **101**
PluginWatcher class **50**
plugs-in, PluginTrigger class **46**
policies **43, 44**

Q
QueryResult **101**

S
ScriptingAttribute annotation **107**
ScriptingFunction annotation **107**
ScriptingParameter annotation **108**
SDKFinderProperty class **102**
solar system application, components **27**
solar system plug-in
  components **30**
  set up **71**

V
vso.xml
  action element **113, 125**
  architecture **20**
  attribute element **123**
  attributes element **123**
  code element **125**
  configuration element **111**
  constructor element **122**
  constructor parameter element **122**
  constructors element **122**
  description element **112**
  entries element **127**
  entry element **127**
  enumeration element **126**
  enumerations element **126**
  events element **119**
  finder element **116**
  finder-datasource element **114**
  finder-datasources element **114**
  finders element **115**
  gauge element **120**
  id element **118**
  installation element **113**
  inventory element **115**
  inventory-children element **119**
  method element **124**
  method parameter element **125**
  method parameters element **125**
  methods element **124**
  object element **121**
  parameters element **122**
  properties element **117**
  property element **117**
  relation element **118**
  relation-link element **119**
  relations element **118**
  scripting-objects element **121**
  singleton element **126**
  trigger element **119**
  trigger-properties element **120**
  trigger-property element **120**
  url element **112**
  webview-components-library element **113**
vso.xml file
  definition **19**
  elements **109**
  module element **110**