vSphere Web Client Extensions Programming Guide

vSphere 5.5 Update1

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-001398-00

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

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

Contents

About This Book

The *vSphere Web Client Extensions Programming Guide* provides information about developing extensions to the vSphere Web Client in vSphere 5.5.

VMware provides many APIs and SDKs for different applications and goals. This book provides information about the vSphere Web Client extensibility framework for developers that are interested in extending the vSphere Web Client with custom functionality.

Revision History

This book is revised with each release of the product or when necessary. A revised version can contain minor or major changes. Table 1 summarizes the significant changes in each version of this book.

Table 1. Revision History

Revision Date	Description
03MAR2014	vSphere 5.5 U1 release adds HTML extensions.
03DEC2013	Add information about list view extension visibility.
20NOV2013	Correction and clarifications about how to register a plug-in.
19SEP2013	Release of the vSphere Web Client SDK for vSphere 5.5. New features include: ■ Action Sub-Menu Extensions ■ Action Prioritization

Intended Audience

This book is intended for anyone who needs to extend the vSphere Web Client with custom functionality in vSphere 5.5. Users typically include software developers who use Flex or ActionScript to create graphical user interface components that work with VMware vSphere.

VMware Technical Communications Glossary

VMware Technical Communications provides a glossary of terms that might be unfamiliar to you. For definitions of terms as they are used in VMware technical documentation go to http://www.vmware.com/support/pubs.

Document Feedback

VMware welcomes your suggestions for improving our documentation. Send your feedback to docfeedback@vmware.com.

Introduction to the vSphere Web Client 1

The VMware vSphere® Web Client is the primary method for system administrators and end users to interact with the virtual data center environment created by vSphere. vSphere manages a collection of objects that make up the virtual data center, including hosts, clusters, virtual machines, data storage, and networking resources.

The vSphere Web Client is a Web browser-based application that you can use to manage, monitor, and administer the objects that make up your virtualized data center. You can use the vSphere Web Client to observe and modify the vSphere environment in the following ways.

- Viewing health, status, and performance information on vSphere objects
- Issuing management and administration commands to vSphere objects
- Creating, configuring, provisioning, or deleting vSphere objects

You can extend vSphere in different ways to create a solution for your unique IT infrastructure. You can extend the vSphere Web Client with additional GUI features to support these new capabilities, with which you can manage and monitor your unique vSphere environment.

This chapter includes the following topics:

Understanding the vSphere Web Client Architecture

The vSphere Web Client architecture consists of a user interface layer and a service layer. Both layers reside on a Web application server, called the Virgo server. Each layer has a role in communicating with the vSphere environment, retrieving data, and presenting that data to the user's Web browser.

User Interface Layer

The user interface layer consists of an Adobe Flex application that is displayed in the user's Web browser. The Flex application contains all of the user interface elements with which the user interacts, such as menus, navigation elements, data portlets, and commands. The user can navigate through the various Flex elements in the user interface layer to view data on vSphere objects, send commands, and make changes to the vSphere environment.

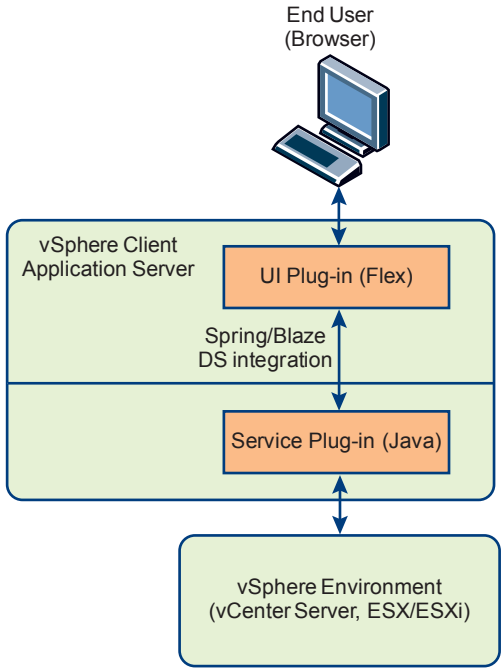
The exact configuration of the elements in the user interface layer depends on the unique properties of the vSphere environment and other factors such as the level of privileges each user has.

Service Layer

The service layer is a collection of Java services that run in a framework on the vSphere Web Client application server, called the Virgo server. These Java services communicate with vCenter Server and other parts of the vSphere environment, as well as other remote data sources. The Java services gather monitoring data on the virtual infrastructure, which is in turn displayed to the user by the Flex user interface layer. When the user performs an action from the Flex user interface, such as a management or administration command, the Java services perform that command on the virtual infrastructure.

The vSphere Web Client application server contains a Spring framework that manages communication between the user interface layer and the service layer.

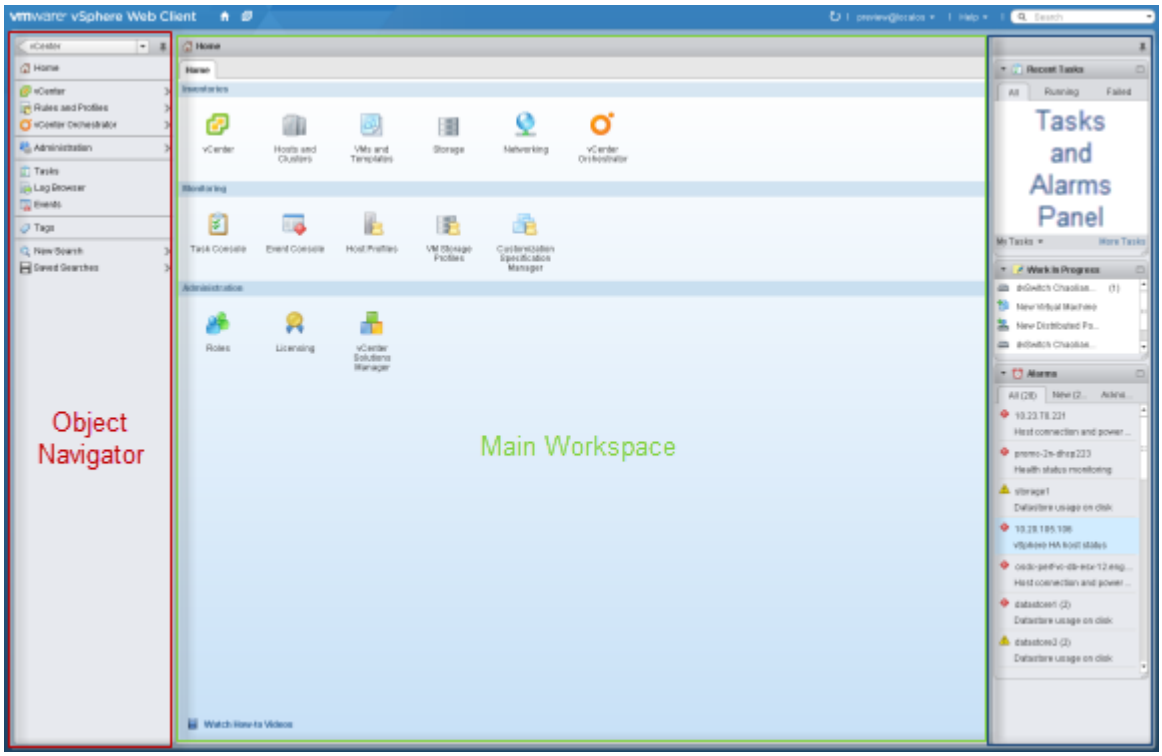
Figure 1-1. vSphere Web Client Architecture



Overview of the vSphere Web Client User Interface

The user interface layer of the vSphere Web Client contains all of the Flex objects, such as data views, toolbars, and navigation interfaces, that make up the vSphere Web Client graphical user interface.

The major parts of the vSphere Web Client user interface are the object navigator, the main workspace, and the tasks and alarms panel.

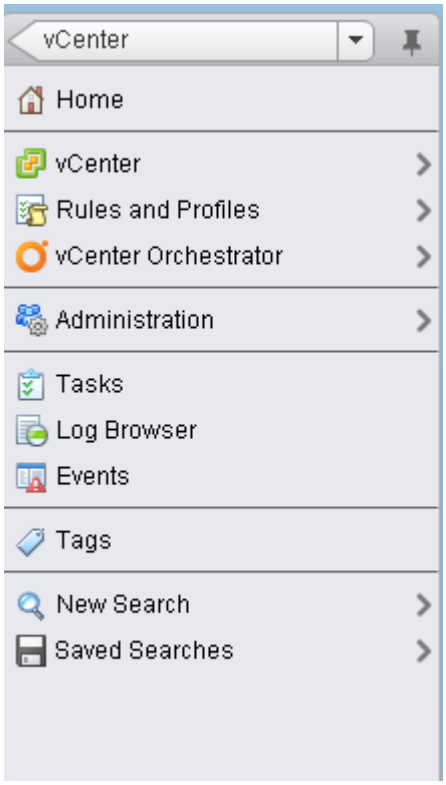


Object Navigator

The object navigator is the user's primary means of browsing the virtual infrastructure and accessing other solutions and data views in the vSphere Web Client. The user's selections in the object navigator drive the content of the vSphere Web Client main workspace.

Object Navigator Top Level

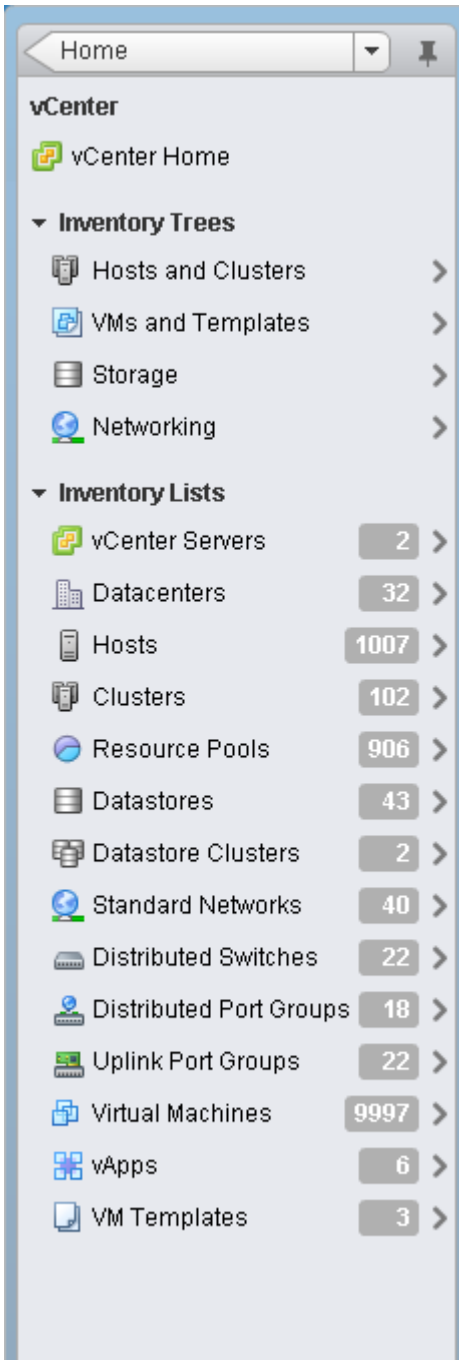
The top level of the object navigator contains links to the major features and solutions in the vSphere Web Client, including the vCenter, Rules and Profiles, and Administration applications. You can extend the object navigator top level with links to new solutions that you create, such as Global Views.



Object Navigator When Browsing the Virtual Infrastructure

When the user browses the virtual infrastructure, the object navigator is the user's primary means of interacting with the vSphere objects in the data center. In the object navigator vCenter level, vSphere objects are organized into inventory trees and inventory lists. Users can browse and search the objects in the inventories using the object navigator. When the user selects an object in the object navigator, information about that object appears in the main workspace.

When you extend the vSphere Web Client to support custom object types, you must extend the object navigator vCenter level with new inventory lists or custom object lists. You can also add links to other solutions to the vCenter level.



Main Workspace

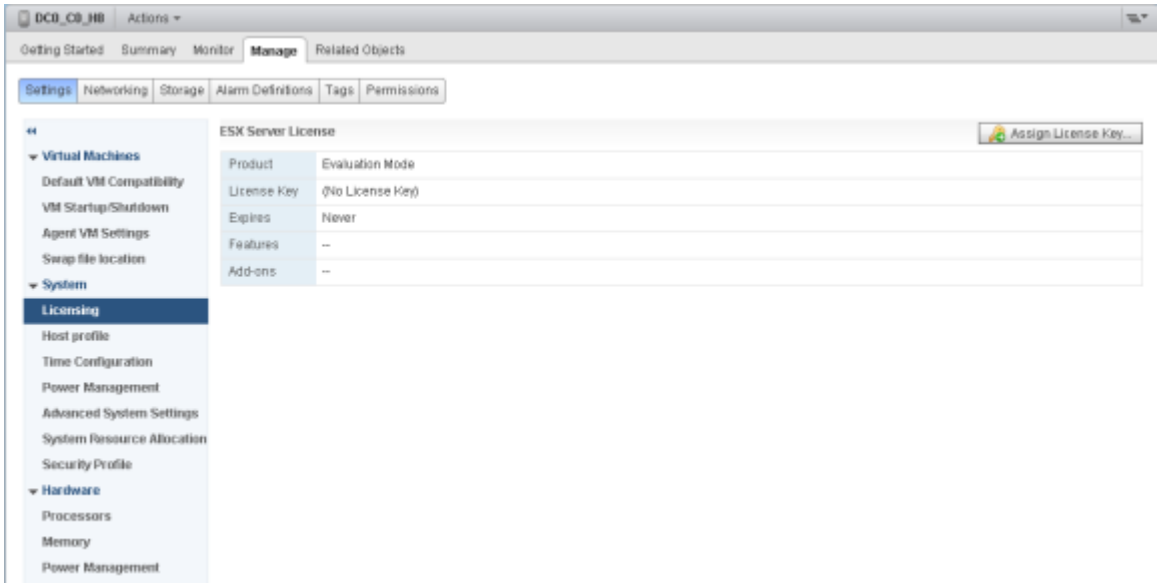
The main workspace is where the vSphere Web Client displays the home screen, solutions and applications, and information about the virtual infrastructure. The main workspace is the center of the vSphere Web Client graphical user interface and contains data views, navigation elements such as tabs and toolbars or context menus for user actions.

Home Screen

The home screen is the initial view shown in the main workspace when the user logs in to the vSphere Web Client. The home screen contains icon shortcuts to different solutions and inventories in the virtual infrastructure. You can extend the home screen by adding additional shortcuts.

Browsing the Virtual Infrastructure

When the user browses the virtual infrastructure using the object navigator, the main workspace displays an object workspace. An object workspace presents information about the selected vSphere object in a hierarchy of nested Flex data views, which are displayed as top-level tabbed screens. Any given vSphere object has associated **Getting Started**, **Summary**, **Monitor**, **Manage**, and **Related Objects** top-level tab screens. Some of these tabs contain second-level tabs and views within the tabs.



Some of the nested data views are contextual. For example, the **Monitor** tab always contains a second-level **Issues** tab, and might contain a second-level **Events** tab if any events are present for the selected object.

You can extend the object workspace for any given vSphere object type by adding second-level tabs or views to the existing hierarchy. The object workspace for each vSphere object contains these top-level tabs:

- **Getting Started**
- **Summary**
- **Monitor**
- **Manage**
- **Related Objects**

Getting Started Tab

The **Getting Started** tab shows a basic description of the vSphere object and some contextual information on how the object operates within the vSphere environment. The **Getting Started** tab might also provide links to common tasks for that object.

The **Getting Started** tab might not appear or might be disabled for a vSphere object.

Summary Tab

The **Summary** tab shows basic, high-level information about the selected object. The **Summary** tab might also show portlets with additional specific information about the object's features. The **Summary** tab quickly and clearly gives the user enough information to identify the specific object they have selected, and to see that object's role in the virtual infrastructure.

Monitor Tab

The **Monitor** tab shows current and historical information about how the selected object is performing. The **Monitor** tab shows alerts, issues, and other signals from the vSphere environment to which the user might respond. The **Monitor** tab generally contains data views that show information about the object's health, performance statistics, and event logs, and any issues and alarms that were raised.

Manage Tab

The **Manage** tab displays settings and tasks that determine how the selected object behaves. Using the **Manage** tab, users can perform operations on a vSphere object, such as provisioning or maintenance. Users can also change object settings and issue management commands from the **Manage** tab.

Related Objects Tab

The **Related Objects** tab shows lists of vSphere objects related to the currently selected object. For example, a Cluster object's **Related Objects** tab can contain a list of the Host objects in the cluster, as well as the related virtual machines, storage pods, networks, and other resources. Users can select a related object directly from the **Related Objects** tab, and view the workspace for that related object.

Global Views

The main workspace can display global views. A global view is a data view that is not a part of an object workspace for any vSphere object type. A global view is a free-form data view and need not follow a tab hierarchy as an object workspace does.

You can extend the vSphere Web Client with your own global views. Your global views can collect or summarize information from many different sources in the vSphere environment to create a dashboard or quick access screen, or to display information from outside the vSphere environment.

Workspaces for Custom Objects

If you add a custom object type to the vSphere environment, you can extend the main workspace to display an object workspace for that custom object. The vSphere Web Client SDK contains templates to help you create the standard object workspace tabs, such as **Getting Started**, **Summary**, **Monitor**, **Manage**, and **Related Objects**.

Administration Application

The Administration application allows users to change administrative settings and preferences for each service in the vSphere environment, as well as for the vSphere Web Client itself.

When the user selects the Administration application, the object navigator displays different categories of services such as Access, Licensing, Sign-On and Discovery, and Solutions. Information about specific services appears in the main workspace. The vSphere Web Client and additional plug-in modules appear as a service in the Administration navigation interface.

Understanding Extensibility in the vSphere Web Client

You extend the vSphere Web Client by creating plug-in modules. Each plug-in module extends either the user interface layer or the service layer of the vSphere Web Client. The user interface plug-in modules and service plug-in modules together form a complete solution to add new capabilities to the vSphere Web Client graphical user interface.

In general, you extend the vSphere Web Client for one of the following reasons.

- **You extended the vSphere environment in some way.** You can extend vSphere by adding a new type of object to the environment, or by adding more data to an existing object. If you extend vSphere in this way, you can extend the vSphere Web Client with new user interface elements that allow users to observe, monitor, and control these new objects.

- **You want to view existing vSphere data in a different way.** You can extend the vSphere Web Client without having added new objects or data to the vSphere environment. For example, you might want to collect existing vSphere data on a single screen or location in the user interface. Shortcuts, global views, and object navigator inventory lists are examples of extensions that you can use for these purposes. You can also create a new second-level tab, portlet, or other data view that displays existing vSphere data, such as performance data, as a custom graph or chart.

Extending the vSphere Web Client can involve creating both user interface plug-in modules and service plug-in modules.

User Interface Plug-In Modules

A user interface plug-in module adds one or more extensions to the vSphere Web Client user interface layer. Extensions to the user interface layer can include new data views, either in the virtual infrastructure or as global views. When you create a data view extension, you must also create the actual GUI objects in Adobe Flex or in HTML and package them in the plug-in module. These GUI objects rely on data from the vSphere Web Client service layer. You can use the libraries included with the vSphere Web Client SDK to enable communication between your GUI objects and the service layer.

Other user interface extensions can include new workspaces for custom objects, shortcuts added to the object navigator or home screen, new relations between vSphere objects, and new actions associated with vSphere objects.

For a complete discussion of the types of extensions you can add to the vSphere Web Client user interface, see Chapter 2, “vSphere Web Client User Interface Layer,” on page 21.

Adding Java Services

You can add new Java services to the vSphere Web Client service layer. The Java services you add can perform any of the functions of a typical Java Web service. In general, however, Java services you add to the vSphere Web Client service layer are used to retrieve data from the vSphere environment for display in the user interface layer, or to make changes to the vSphere environment in response to actions in the user interface layer.

Getting Data from the vSphere Environment

Service plug-in modules that gather data from the vSphere environment usually extend the native services on the vSphere Web Client application server, such as the Data Service. You can create standalone custom Java services for data gathering, but a best practice is to extend the built-in services in the vSphere Web Client SDK. Extensions to the built-in services in the vSphere Web Client SDK are often simple wrappers around existing Java services that you create.

In general, you must extend the vSphere Web Client Data Service if your extension solution meets any of the following criteria.

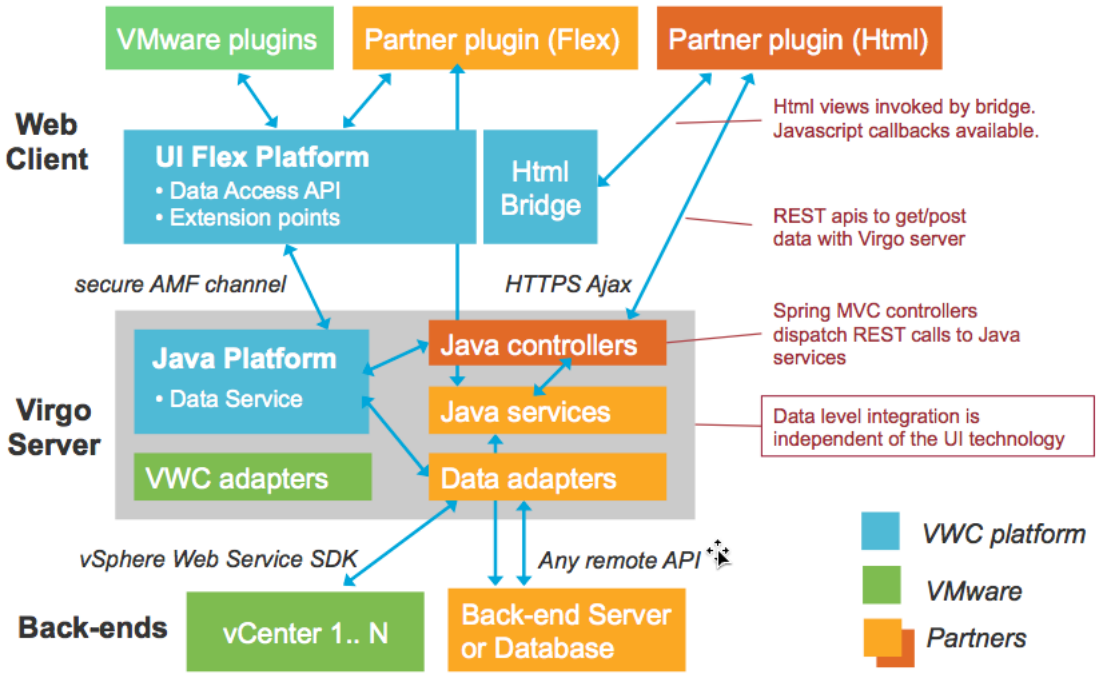
- **Your extension provides new data about existing vSphere objects.** If your extension provides a GUI element to display data that the vSphere Web Client services do not already provide, you must extend the Data Service to provide this data.
- **You want to add a new type of object to the vSphere environment.** If you are adding a new type of object to the vSphere environment, you can extend the Data Service to provide data for objects of the new type.

The service extensions you create can access data from any source, either inside or outside of the vSphere environment. For example, you can create an extension to the Data Service that retrieves data from an external Web server, rather than from vCenter Server.

Making Changes to the vSphere Environment

Service plug-in modules that make changes to the vSphere environment are standalone Java services that you create. These services are used when the user starts an action in the vSphere Web Client user interface. If you create an action extension, you must also create the Java service that performs the action operation on the vSphere environment as a service plug-in module.

Figure 1-2. Web Client Extensions Service Architecture



Plug-In Modules in a Complete Solution

The plug-in modules you must create for your solution depend on the features your solution provides for the user. Most extension solutions include at least one user interface plug-in module, to add a new feature to the vSphere Web Client user interface. Depending on what types of extensions your user interface plug-in module contains, you might also need to add a service to the service layer to support your user interface extensions.

Some common extension solutions for the vSphere Web Client are:

- Global views
- Adding data views to existing virtual infrastructure object workspaces
- Adding actions to existing vSphere objects
- Adding a Custom Object Type to the vSphere Environment

Global Views

A global view is a custom data view that appears in the vSphere Web Client main workspace. You can use a global view to display any data from inside or outside the vSphere environment. Typical uses for global views include dashboard applications that display information on vSphere objects, or complex monitoring or management applications that are separate from the virtual infrastructure.

To add a global view to the vSphere Web Client, you must create a user interface plug-in module that contains the following extensions.

- A global view extension. The global view extension specifies the objects that appear in the main workspace when the user selects the global view.
- An extension to the object navigator or home screen. This extension serves as the link or shortcut through which the user can access the global view.

If your global view requires data from the vSphere environment that the vSphere Web Client Data Service does not already provide, you must also create a Data Service Adapter to extend the Data Service.

Adding Data Views to Existing Virtual Infrastructure Object Workspaces

You can add data view extensions to the object workspace for any existing object in the vSphere Web Client. The vSphere Web Client provides extension points to let you add custom elements to the existing **Getting Started**, **Summary**, **Monitor**, **Manage**, and **Related Objects** tabs for each type of vSphere object, such as a host, virtual machine, or cluster. These data view extensions are displayed as second-level tabs or tab views in the object workspace hierarchy.

To add a data view to an existing object workspace for a vSphere object, you must create a user interface plug-in module that contains a data view extension. The data view extension specifies the visual component that appears in the main workspace in the object workspace hierarchy.

If your data view extension requires data from the vSphere environment that the vSphere Web Client Data Service does not already provide, you must also create a Data Service Adapter to extend the Data Service.

Adding Actions to Existing vSphere Objects

You can create actions associated with existing vSphere objects, such as hosts, clusters, or virtual machines. When you create an action extension, your actions are added to the vSphere Web Client Actions Framework. The Actions Framework displays the available actions for a given object to the user with toolbars and context menus in the vSphere Web Client main workspace.

To add an action to an existing vSphere object, you must create a user interface plug-in module that contains an action extension. The action extension specifies the details of the action, including when the action is available and how the action is handled.

You must also add a Java service to the vSphere Web Client service layer. The service layer code that handles the action execution must import the new Java service. The Java service is responsible for performing the action operation on the vSphere environment.

Adding a Custom Object Type to the vSphere Environment

You can add new object types to the vSphere environment, and extend the vSphere Web Client to display information about the new objects. The vSphere Web Client SDK includes templates that you can use to create the standard object workspace, including the **Getting Started**, **Summary**, **Monitor**, **Manage**, and **Related Objects** tabs, for your custom object type.

To add support for a custom object type to the vSphere Web Client, you must create a user interface plug-in module that contains the following extensions.

- A templated object workspace extension. The object workspace extension creates the standard tab hierarchy for the new custom object, including the **Getting Started**, **Summary**, **Monitor**, **Manage**, and **Related Objects** tabs.
- A data view extension for each tab in your object workspace.
- An action extension for each action associated with the new object type.
- An object navigator extension that adds a new Inventory List. The Inventory List is a shortcut in the object navigator virtual infrastructure level. The shortcut represents an aggregation of all of the instances of your object type in the vSphere environment.
- A Related Objects extension that describes the relationship between your custom object and other vSphere objects in the virtual infrastructure.

In addition, you must create the following service plug-in modules for your custom object.

- A Data Service Adapter to extend the vSphere Web Client Data Service. The Data Service Adapter gives the Data Service the capability to retrieve information about the new object type.
- A Java service for any actions associated with your new object type. The Java service performs the action operations on your custom objects in the vSphere environment.

Plug-In Packages

The plug-in modules you create for your extension solution, for the user interface layer and the service layer, are deployed to vSphere in plug-in packages. A plug-in package bundles together one or more plug-in modules. Each plug-in package represents a complete solution for the vSphere Web Client. The package contains at least one user interface plug-in module, that adds new elements to the vSphere Web Client user interface, and might contain one or more Java services that are required to provide the new elements with data.

You deploy plug-in packages to a vCenter server by registering with the server's `ExtensionManager` API. When the vSphere Web Client connects to a vCenter server, the vSphere Web Client downloads and deploys any plug-in packages that are currently registered with the vCenter server.

Requirements for Developing vSphere Web Client Extension Solutions

To create plug-in modules for the vSphere Web Client, your development environment must include the following items.

- A development environment capable of developing Web applications using ActionScript and MXML or Javascript. A best practice is to use the Spring Tools Suite and the FlexBuilder plug-in for the Eclipse IDE. The vSphere Web Client SDK includes tools and plug-ins for the Spring Tools Suite to aid you in creating vSphere Web Client plug-in modules.
- A text editor for creating XML files. While you can use any text editor to create the required XML, a best practice is to use an XML editor such as Altova XML Spy to ensure that you create well-formed XML files.
- A development environment capable of developing Java-based Web applications. A best practice is to use the Spring Tools Suite. The vSphere Web Client contains tools and plug-ins for the Spring Tools Suite to aid you in creating services compatible with the vSphere Web Client Virgo server framework.
- A build of the vSphere Web Client on a Web server, running the vSphere Web Client Virgo server framework.
- A vCenter Server or vCenter Server Virtual Appliance, to which the vSphere Web Client can connect.

NOTE You can build out the virtualization stack you need to run vCenter Server and the Virgo server using virtual machines, instead of providing dedicated hardware.

Flex Run-Time Version Dependencies

To achieve a richer user interface, the vSphere Web Client uses both Flex 3 (`halo`) and Flex 4 (`spark`) themes. These dependencies are managed implicitly when you build your plug-in using the Eclipse IDE with the FlexBuilder plug-in.

If you use Maven or other tools to build your plug-in, you must specify the dependencies on the `halo` and `spark` component libraries. Even if your plug-in does not use `halo` components directly, parts of the VMware framework do use `halo` components. If you omit a dependency, your project might compile but not render the UI correctly at run time.

The `flexmojos` build artifact for Maven establishes a dependency for the `spark` component library, but not for the `halo` component library. You must add an explicit `halo` dependency to the build.

To specify the `halo` dependency using Maven, add the following lines to your `ui/pom.xml` file:

```
<dependency>
   <groupId>com.adobe.flex.framework</groupId>
   <artifactId>halo</artifactId>
   <version>${flex.sdk.version}</version>
   <type>swc</type>
   <classifier>theme</classifier>
   <scope>theme</scope>
</dependency>
```

If you have a different build tool, use appropriate means to configure a `halo` dependency or build action.

Naming Convention for vSphere Objects in the vSphere Web Client SDK

The vSphere Web Client user interface uses user-friendly names for the default types of vSphere-managed objects. However, the service APIs, extension points, and libraries in the vSphere Web Client SDK refer to vSphere objects using the naming convention established by the vSphere API. Table 1-1 contains a mapping of names in the vSphere Web Client user interface to the object type names used in the vSphere Web Client SDK.

Table 1-1. vSphere Entity Names in the UI and SDK Code

vSphere Entity Name in Web Client User Interface	Entity Name in SDK Code
Cluster	`ClusterComputeResource`
Datacenter	`Datacenter`
Datastore	`Datastore`
Distributed Port Group	`DistributedVirtualPortgroup`
Distributed Switch	`DistributedVirtualSwitch`
Folder	`Folder`
Host	`HostSystem`
Standard Network	`Network`
Resource Pool	`ResourcePool`
Storage Pod	`StoragePod`
vApp	`VirtualApp`
Virtual Machine	`VirtualMachine`

vSphere Web Client User Interface Layer 2

The user interface layer represents the front end of the vSphere Web Client, and contains the GUI elements that users view and click. The user interface is based on a modular Flex application, hosted on the vSphere Web Client application server, that runs in the user's Web browser. The user interface layer contains every visual component of the application, including data views, portlets, and navigation controls.

You can add new features or elements to the Flex application by creating user interface plug-in modules. A user interface plug-in module contains one or more extensions, which add Flex or HTML GUI elements to the vSphere Web Client user interface.

This chapter contains the following topics:

Understanding the User Interface Layer

The user interface layer consists of a Flex application, called the container application, that serves as a framework for a set of modular GUI elements. The container application supplies the structural components of the vSphere Web Client user interface, such as major navigation controls, window frames, and menus. The container application manages and renders each of the GUI elements that it contains. Together, the collected GUI elements and the container application make up the vSphere Web Client user interface.

Each GUI element inside the container application is a self-contained object that communicates directly with the vSphere environment. Each element can retrieve data for display, or send commands or make changes to the virtual infrastructure. Each GUI element implements a version of the MVC architecture to manage communication with the vSphere environment.

The container application uses a metadata framework to determine where to place each GUI element in the interface, and how that element looks. Each GUI element has a metadata definition that you create using an XML schema that is defined in the vSphere Web Client SDK. The container application uses the values in the metadata definition to render the element at the appropriate place in the interface.

User Interface Plug-In Modules

A user interface plug-in module is a standard Web Application ARchive (WAR) bundle. The WAR bundle contains all of the resources and classes required for each GUI element the module adds to the vSphere Web Client interface. In a user interface plug-in module, the GUI elements you add to the interface are called extensions.

In the SWF file that contains the Flex classes that make up the UI components of the plug-in module extensions, the main class must be an MXML class that extends the `mx.modules.Module` class.

In the root folder of the WAR bundle, you must create a manifest file called `plugin.xml`. The `plugin.xml` manifest file uses metadata to define the plug-in module's extensions and resources.

Creating the Plug-In Module Manifest

The plug-in module manifest file, `plugin.xml`, is a metadata file that the vSphere Web Client container application uses to integrate the extensions in the plug-in module with the rest of the interface. The `plugin.xml` file fulfills the following functions.

- The manifest defines each individual extension in the user interface plug-in module.
- The manifest specifies the SWF file containing the Flex or ActionScript classes you have created for the module extensions.
- If the user interface plug-in module contains an interface element defined in HTML, the manifest specifies the HTML file.
- The manifest specifies the location of any included runtime resources in the module, such as localization data.
- If the user interface plug-in module hosts any extension points, the manifest declares those extension points.

XML Elements in the Manifest File

The metadata in the manifest file follows a specific XML schema. The major XML elements of a user interface plug-in module manifest include the `<plugin>` element, the `<resources>` element, and one or more `<extension>` elements. The vSphere Web Client SDK contains several examples of user interface plug-in modules with complete `plugin.xml` manifest files.

<plugin> Element

The `<plugin>` element is the root element of any plug-in module manifest file. All other elements are contained within the `<plugin>` element. The attributes of the `<plugin>` element contain information about the entire plug-in module.

- **id**. The unique identifier that you choose for the plug-in module.
- **moduleUri**. A Uniform Resource Identifier (URI) for the SWF file in the plug-in module. The SWF file contains the Flex or ActionScript classes used by the extensions in the plug-in module. The URI should be specified relative to the root directory of the plug-in module WAR bundle.
- **securityPolicyUri**. A URI for a standard cross-domain security policy file. This optional element is used when the plug-in module is hosted remotely and the domain of the plug-in module URL is different from the domain vSphere Web Client application server. For more information on specifying a cross-domain security policy file, go to http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.
- **defaultBundle**. The name of the default resource bundle for the plug-in module. The bundle name must be unique to your plug-in module to avoid name clashing issues with other plug-in modules. Resources, such as localization data and icons, can be dynamically loaded at runtime. See "Specifying Dynamic Resources" on page 23.

The following XML fragment shows how the `<plugin>` element might appear in the plug-in module manifest file.

```
<plugin id="com.acme.sampleplugin"
    moduleUri="SampleModule.swf" defaultBundle="com_acme_sampleplugin">
    ...
    <!-- additional plugin data -->
    ...
</plugin>
```

<resources> Element

The <resources> element is used to specify the location of the plug-in module's runtime resources, such as localization data. In general, resources are bundled in separate SWF files from the SWF file that contains the plug-in module Flex classes.

The vSphere Web Client supports the same set of locales as vCenter Server. The default locale is the locale set by the user's Web browser. If the user's Web browser is not set to a locale that the vSphere Web Client supports, the vSphere Web Client defaults to the English (United States) locale.

A best practice is to set the locale attribute in the <resource> element to the value {locale}, rather than hard-coding a specific locale, in your plugin.xml manifest. Using the {locale} value instructs the vSphere Web Client to use the locale that the user's Web browser specifies at runtime.

For the vSphere Web Client to properly import the resource locale at runtime, resource bundles for all supported locales must be included in the plug-in module. Currently supported locales include English (United States) (en_US), French (fr_FR), German (de_DE), Japanese (ja_JP), Korean (ko_KR), and Mandarin Chinese (zh_CN) locales. For testing purposes, you can make copies of the default locale for any languages for which you do not have resources available. The sample plug-in modules included with the vSphere Web Client SDK demonstrate this method.

The following XML fragment shows how the <resources> element might appear in the plug-in module manifest file.

```
<resources>
    <resource locale={locale}>
        <module uri="locales/sample-plugin-{locale}.swf"/>
    </resource>
</resources>
```

In the preceding example, the {locale} placeholder corresponds to a particular locale, such as en_US for English. Your plug-in module must contain a separate .SWF resource bundle for each of the supported locales, using the relative path in the preceding example.

Specifying Dynamic Resources

Your extension definitions can use the dynamic resource loader to load resources or localization data at runtime. When you have specified a <resources> bundle in the manifest file, the <extension> elements in the manifest file can specify that bundle by reference. Resource specifications in the <extension> elements should use the expression #{bundle:key} when specifying resources, where bundle indicates the name of the resource bundle, and key indicates the resource to use.

For example, in an <extension> element in the plugin.xml file, you can specify an icon resource to be loaded dynamically at runtime by using the following expression.

```
<icon>#{MyResourceBundle:MyIconImage}</icon>
```

Specifying a dynamic resources allows you to avoid using a static value for the icon resource. The bundle name must specify a resource bundle that you made available in the <resources> element of plugin.xml.

You can also omit the bundle name from a resource expression:

```
<icon>#{MyIconImage}</icon>
```

Omitting the bundle name causes the vSphere Web Client to use the bundle name that the defaultBundle attribute in the <plugin> element specifies.

<extension> Element

The plug-in module manifest file must define each extension that the plug-in module adds to the vSphere Web Client. Each extension is defined using the <extension> element. The <extension> element contains information about each feature, and its exact composition varies depending on the kind of user interface element your plug-in module is adding. See "Defining Extensions" on page 26.

Extensions Ordering

You can use the <precedingExtension> element to specify the order in which the vSphere Web Client renders the extensions in your plug-in module.

Within each <extension> element, you can specify a <precedingExtension> element that contains the ID of another extension that is to be rendered before the current extension. Setting the value of <precedingExtension> to NULL causes that extension to be rendered first.

If no <precedingExtension> value is specified, the extensions are rendered in the order they appear in the plugin.xml module manifest file. If multiple extensions specify the same value for <precedingExtension>, they are rendered after that specified extension in the order in which they appear in the manifest.

The following XML fragment shows how the <precedingExtension> element might appear in the extension definitions in the plug-in module manifest file.

```
<extension id = "com.MyPluginPackage.MyPlugin.PerformanceView">
    <extendedPoint>vsphere.core.vm.views</extendedPoint>
    <precedingExtension>NULL</precedingExtension>
    ...(extension data)...
</extension>

<extension id = "UtilityView">
    <extendedPoint>vsphere.core.vm.views</extendedPoint>
    <precedingExtension>PerformanceView</precedingExtension>
    ...(extension data)...
</extension>
```

The <precedingExtension> elements in the example ensures that the PerformanceView extension is rendered first, followed by the UtilityView extension.

Types of vSphere Web Client Extensions

You can create several types of extensions to the vSphere Web Client user interface. Each type of extension has different requirements for its metadata definition, and some extensions can require you to provide visual components to serve as new GUI elements.

To build a complete extension solution for the vSphere Web Client, you might need to include multiple types of extensions in your user interface plug-in module. For a more complete discussion of the individual extensions required for common solutions, see "Plug-In Modules in a Complete Solution" on page 17.

You can create these types of extensions to the vSphere Web Client user interface:

- Global view extension
- New data views
- Object navigator extensions
- Action extensions
- Relation extensions
- Home screen shortcuts
- Object list view extensions

Adding a Global View

A global view extension is a free-form, general-purpose data view that appears in the vSphere Web Client main workspace. Unlike the object workspaces for objects in the virtual infrastructure, a global view extension does not need to follow any particular tab structure. The structure of your global view can be simple or complex, depending on how you create the components for the extension. For more information on creating components for data view extensions, see Chapter 5, "Architecting Data Views," on page 43.

Users can access global views through pointers in either the object navigator or the home screen. For users to access your global view extension, you must also create extensions that add a pointer to the object navigator, home screen, or both.

See Chapter 3, "Adding a Global View Extension," on page 33.

Adding Data Views to Virtual Infrastructure Objects

When the user selects a vSphere object in the object navigator, that object's workspace appears in the vSphere Web Client main workspace. Each object workspace contains a hierarchy of nested data views in a series of top-level tabs and second-level tabs. See "Browsing the Virtual Infrastructure" on page 14.

You can add new data views to the tabs in any object workspace. The vSphere Web Client provides extension points for specific locations in each object workspace. For more information on extension points, see "Defining Extensions" on page 26.

When you create a data view extension for an object workspace, you must also provide the Flex and HTML classes that appear as the new GUI element in the interface. As a best practice, create these classes by using the same architecture, libraries, and services as the existing data views in the vSphere Web Client. See Chapter 5, "Architecting Data Views," on page 43.

You can also add entirely new object workspaces, with the default structure and tab screens, to the interface. In general you create object workspace extensions to support new object types that you add to the vSphere environment.

See Chapter 4, "Adding to vCenter Object Workspaces," on page 37.

Enhancing the Object Navigator

The object navigator is the main navigation interface in the vSphere Web Client. The object navigator control always appears on the left side of the vSphere Web Client screen.

The object navigator control's top level contains pointers to the major solutions and applications in the vSphere Web Client, such as the vCenter browser and the Administration application. You can extend the top level of the object navigator with a pointer to your own solution or global view.

The user browses the virtual infrastructure by using the object navigator vCenter page. The vCenter page shows the objects in the vSphere environment in inventory trees and inventory lists. You can extend this level of the object navigator with a new inventory list. In general, you add a new inventory list to represent a new or custom type of object in the environment. You can also create your own categories of lists in the object navigator vCenter page.

You create most object navigator extensions by defining the extensions using metadata, in the `plugin.xml` manifest file. Most object navigator extensions do not require that you create new visual components.

See Chapter 6, "Extending the Object Navigator," on page 59.

Adding Actions

Actions represent the commands and requests that the users can send to make changes to the virtual infrastructure in the vSphere environment. All actions in the vSphere Web Client are governed by the Actions Framework. The Actions Framework determines which actions appear in the vSphere Web Client user interface, in toolbars and context menus. You can add actions to the vSphere Web Client by creating an extension to the Actions Framework.

When you create an action extension, you must also provide the ActionScript classes that are called when the user performs the action. You must also extend the vSphere Web Client service layer with a Java service. The Java service performs the action operation in the vSphere environment.

See Chapter 7, "Creating Action Extensions," on page 67.

Creating a Relation Between vSphere Objects

vSphere uses a graph of related objects to model the objects that make up the virtual infrastructure. When the user selects an object in the GUI, the vSphere Web Client can provide a list of any objects that are related to the currently selected object. For example, a user can access information about the virtual machines related to a given host object. These lists appear in the object navigator control and in the **Related Objects** tab in the selected object's workspace.

You can create extensions that define new relationships between vSphere objects, or relationships between a new type of vSphere object that you created and the existing objects in the vSphere environment. You create relation extensions by using only the metadata definition. You do not need to create a Flex or HTML class.

See Chapter 8, "Creating a New Relation Between vSphere Objects," on page 83.

Adding a Home Screen Shortcut

You can create extensions that add a shortcut to the vSphere Web Client home screen. Home screen shortcuts can provide the user with a pointer to any data view in the vSphere Web Client, including global views, specific inventory lists, or a top level solution application in the vSphere Web Client.

You create Home screen shortcut extensions by using only the metadata definition. You do not need to create a Flex or HTML class.

See Chapter 9, "Creating Home Screen Shortcuts," on page 87.

Extending Object List Views

Each object workspace in the vSphere Web Client contains a list view that displays information about each individual object of that type in the virtual infrastructure. You can create a new list view extension for your own custom object type, or extend the existing list views for any type of vSphere object. When you extend an existing vSphere object list view, you can add new data columns to appear in the list.

See Chapter 10, "Extending vSphere Object List Views," on page 89.

Defining Extensions

The metadata extension definition for a user interface extension specifies where the extension appears in the vSphere Web Client GUI, and provides information about how the extension's appearance, including which Flex classes contain the actual GUI code.

The metadata that you must create is different depending on what kind of extension you are adding to the vSphere Web Client, but all extension definitions share certain basic characteristics.

Extension Point

The extension point represents the specific location in the vSphere Web Client user interface where the extension appears. The vSphere Web Client SDK publishes a set of extension points that correspond to specific locations in the user interface. See Appendix A, "List of Extension Points," on page 113.

Each extension definition must specify one of the provided extension points. The vSphere Web Client SDK provides several extension points for adding new data view extensions to the Virtual Infrastructure object workspaces. Each of these points corresponds to an existing top-level tab or second-level tab for a specific type of vSphere object. The SDK also provides extension points for other extensible GUI features, including the object navigator, Actions Framework, Related Objects specifications, and home screen shortcuts.

You can nest extensions. An extension can define its own extension points, creating a multilevel structure of extensions. The Virtual Infrastructure object workspaces use a nested structure of extensions. For example, top-level tab screen extensions in an object workspace define extension points for second-level tab extensions, and second-level tab extensions define extension points for nested views or portlets.

Extension Object

The extension object is a data object that describes the properties of the extension. You must provide a specific type of extension object for each extension point. Rather than instantiating the actual extension object, your extension definition must use MXML or HTML to provide a description of the required object.

The extension objects associated with each extension point correspond to specific ActionScript classes in the vSphere Web Client SDK. The properties of the ActionScript class are used in the data object to specify things like labels and icons for the extension. In the cases of data views, actions, and certain extensions to the object navigator, you also use the extension object properties to specify the Flex or HTML classes that you created for those extensions.

Filtering Metadata

You can include filtering metadata in your extension definition. You can use filtering metadata to control when the user has access to a given extension. For example, you can filter an extension to appear only for certain types of objects, or when an object's property has a specific value. Filtering metadata is optional and not required for every extension definition.

Extension Definition XML Schema

In the `plugin.xml` manifest file, each extension is defined in its own `<extension>` element. The principal characteristics of the extension are described in their own elements, the `<extendedPoint>` element for the extension point, the `<object>` element for the extension object, and the `<metadata>` element for filtering metadata.

The following example shows an example extension definition, starting with the `<extension>` element. The extension that the XML definition describes adds a data view to the object workspace for a Virtual Machine object. In the example, the data view is a section view for the **Summary** tab in the object workspace for Virtual Machine objects. This example definition would appear in a plug-in module `plugin.xml` manifest file where extensions are defined.

Example 2-1. Example Extension Definition in plugin.xml Manifest File

```
<!-- Add a summary Section View to the VirtualMachine Summary tab by
   extending the extension point "vsphere.core.vm.summarySectionViews". -->

<extension id="com.vmware.samples.viewspropertiesui.vm.summarySectionView">
    <extendedPoint>vsphere.core.vm.summarySectionViews</extendedPoint>
    <object>
        <name>Sample Summary Section View</name>
        <componentClass
            className="com.vmware.samples.viewspropertiesui.views.VmSampleSummarySectionView"/>
    </object>
</extension>
```

In the example, the `<extension>` element contains an `id` attribute, which is a unique identifier that you create. Other extensions, such as extensions to the object navigator, can reference this extension using the `id` attribute.

Specifying the Extension Point

The `<extendedPoint>` XML element contains the name of the extension point. The vSphere Web Client renders the extension at the extension point when the new plug-in module is loaded. In Example 2-1, the extension point is `vsphere.core.vm.summarySectionViews`.

Describing the Extension Object

The `<object>` element describes the required extension data object for the target extension point. The `<object>` element uses standard XML syntax to describe a data object of the ActionScript class type that the extension point requires and sets values for the properties in that object. Each type of extension requires a specific extension data object. See Appendix A, "List of Extension Points," on page 113.

NOTE In general, you do not explicitly specify the extension data object type. The `<object>` element is assumed to be an object of the type that the extension point requires. The `<object>` element has an optional `type` attribute that you can use to explicitly specify a type. You only need to explicitly specify a type if your extension definition provides a subclass of the extension point's required class, or if the extension point requires an interface rather than a class.

In Example 2-1, the extension point `vsphere.core.vm.summarySectionViews` requires a class of type `com.vmware.ui.views.ViewSpec`. The `<object>` element describes how the `ViewSpec` data object looks.

To find the required extension object type for any given extension point, see Appendix A, "List of Extension Points," on page 113.

Objects of type `com.vmware.ui.views.ViewSpec` have a `<name>` property and a `<componentClass>` property. The `<name>` property contains the name that the vSphere Web Client displays for the new data view extension. In Example 2-1, this name is "Sample Monitor View Title." You can also specify a dynamic resource string for any `<name>` property.

The `<componentClass>` property specifies the Flex class that implements the new data view. If you are implementing an HTML component, you specify one of the `htmlbridge` Flex classes supplied by the SDK. Otherwise, you must create the Flex component. In Example 2-1, this class is `com.vmware.samples.viewpropertiesui.views.VmSampleSummarySectionView`, which appears under the **Summary** tab for virtual machine objects.

You can use the ActionScript API reference included with the vSphere Web Client SDK to obtain detailed information on the extension data object that you must create for each type of extension.

Providing Filtering Metadata

The `<metadata>` element contains filtering data that determines when the extension is available to the user. See "Filtering Extensions" on page 28.

Filtering Extensions

In your extension definition, you can use filtering metadata to control when the extension appears in the vSphere Web Client GUI. You can filter extensions based on the selected object type, on the value of any property associated with the selected object, or on the user's privilege level. You set the filter type and the specific filter values by using the appropriate XML elements inside your extension definition's `<metadata>` element.

Filtering Based on Selected Object Type

You can filter your extension to only appear when the user selects one or more specific types of vSphere objects. You specify the types of objects for which the extension is valid by creating an `<objectType>` element in the `<metadata>` element in the extension definition. The extension appears only when the user selects an object whose type matches the value of the `<objectType>` element.

Example 2-2 shows an example extension definition with filtering based on the selected object type. In the example, the extension actions appear only when the user has selected a Virtual Machine object.

Example 2-2. Extension Filtered by Entity Type

```
<extension id="com.vmware.samples.actions.vmActionSet">
      <extendedPoint>vise.actions.sets</extendedPoint>
      <object>
         <actions>
            <com.vmware.actionsfw.ActionSpec>
               <uid>com.vmware.samples.actions.myVmAction1</uid>
               <label>#{action1.label}</label>
               <command className="com.vmware.samples.actions.VmActionCommand"/>
            </com.vmware.actionsfw.ActionSpec>
          </actions>
      </object>
      <metadata>
         <!-- This filters the action to be visible only on VMs -->
```

```
        <objectType>VirtualMachine</objectType>
      </metadata>
</extension>
```

You can use any vSphere or custom object type name as the value for the <objectType> element. To specify multiple object types, include two or more object type names in the <objectType> element, separated by commas. You can also use the * symbol to specify all object types. See "Naming Convention for vSphere Objects in the vSphere Web Client SDK" on page 20.

Filtering Based on the Value of a Property of the Selected Object

You can filter your extension to appear or not appear depending on the value of a property of the selected object. You must use the property value filter in conjunction with the object type filter.

You create the property value filter by describing one or more property value comparisons to be made on the selected object by using the <propertyConditions> element. You include the <propertyConditions> element in the <metadata> element in the extension definition. You can define a single comparison, or define multiple comparisons and conjoin those comparisons together.

In the <propertyConditions> element, you must describe a data object of type com.vmware.data.query.CompositeConstraint using MXML syntax. Using the CompositeConstraint data object, you specify the names of the object properties used in the filter, the desired value for each object property, and the comparison operator. You can also specify a conjoiner if your CompositeConstraint data object has multiple comparisons.

Defining Comparison Conditions Using ComparisonInfo

In CompositeConstraint data object, you describe each property value comparison using the <nestedConstraints> element. The <nestedConstraints> element contains an array of data objects of type com.vmware.data.query.PropertyConstraint. Each PropertyConstraint data object represents one comparison between a given object property and a value you specify.

When you create a PropertyConstraint data object, you specify the object property to compare by using the <propertyName> element, the value to compare against using the <comparableValue> element, and the comparison operator using the <comparator> element.

The value of the <propertyName> element must match the name of the object property to compare. You can set the value of the <comparableValue> element depending on the type of property you are comparing, but the value must be a primitive type. You can use a string value, an integer value, or a Boolean value of true or false in the <comparableValue> element.

You use the <comparator> element to choose how the property is compared against the value you specify in the filter. You can use values of EQUALS, NOT_EQUALS, GREATER, SMALLER, CONTAINS, EQUALS_ANY_OF or CONTAINS_ANY_OF. If you use the CONTAINS operator, you must provide an array of values. If you use the operators EQUALS_ANY_OF or CONTAINS_ANY_OF, you must provide a string containing multiple values in the <comparableValue> element, each separated by a comma.

Conjoining Multiple Comparisons

If your CompositeConstraint data object defines multiple comparisons, you can choose how those comparisons are conjoined by using the <conjoiner> element. You can use a value of AND or OR in the <conjoiner> element.

Example Property Value Filter

Example 2-3, on page 30, shows an example extension definition for an action extension. The extension is filtered based on the value of the property isRootFolder, and whether the selected object has child objects of a particular type. In the example, the extension appears only when the value of the isRootFolder property is true, and the selected object contains child objects that are Virtual Machines.

Example 2-3. Extension Filtered by Entity Property Value

```
<extension id="com.vmware.samples.actions.vmActionSet">
      <extendedPoint>vise.actions.sets</extendedPoint>
      <object>
         <actions>
            <com.vmware.actionsfw.ActionSpec>
               <uid>com.vmware.samples.actions.myVmAction1</uid>
               <label>#{action1.label}</label>
               <command className="com.vmware.samples.actions.VmActionCommand"/>
            </com.vmware.actionsfw.ActionSpec>
         </actions>
      </object>
       <metadata>
         <objectType>Folder</objectType>
         <propertyConditions>
            <com.vmware.data.query.CompositeConstraint>
               <nestedConstraints>
                 <com.vmware.data.query.PropertyConstraint>
                     <propertyName>isRootFolder</propertyName>
                     <comparator>EQUALS</comparator>
                     <comparableValue>
                        <String>false</String>
                     </comparableValue>
                  </com.vmware.data.query.PropertyConstraint>
                  <com.vmware.data.query.PropertyConstraint>
                     <propertyName>childType</propertyName>
                     <comparator>CONTAINS</comparator>
                     <comparableValue>
                        <String>VirtualMachine</String>
                     </comparableValue>
                  </com.vmware.data.query.PropertyConstraint>
               </nestedConstraints>
               <conjoiner>AND</conjoiner>
            </com.vmware.data.query.CompositeConstraint>
         </propertyConditions>
      </metadata>
</extension>
```

Filtering Based on User Privilege Level

You can filter your extension to only appear for users that have specific privileges. You can base your filter on global privilege settings in the vSphere Web Client, such as settings or licenses. For example, you can use a filter to make your extension available only to users that have global privileges to change settings.

You can also filter your extension based on privileges related to specific types of vSphere objects. For example, you can use a filter to make your extension available only to users who have privileges to create or delete `Datastore` objects.

You specify the privilege for which the object is valid by creating a `<privilege>` element inside the `<metadata>` element in the extension definition. The extension appears only for users whose privileges include the value specified by the `<privilege>` element. You can specify multiple privilege values in the `<privilege>` element, separated by commas. If you specify multiple privileges, the user must have all of the specified privilege values for the extension to appear.

Example 2-4, on page 31, shows an example extension definition with filtering based on the user's privileges. In the example, the extension appears only when the user's privileges include `Global.Licenses`.

Example 2-4. Extension Filtered by User Privileges

```
<extension id="vsphere.core.hosts.sampleMonitorView">
    <extendedPoint>vsphere.core.hosts.monitorViews</extendedPoint>
    <object>
        <name>Sample Monitor View Title</name>
        <componentClass className="com.vmware.vsphere.client.sampleplugin.SampleObjectView"/>
    </object>
    <metadata>
        <privilege>Global.Licenses</privilege>
    </metadata>
</extension>
```

You can filter your extension by using any standard vCenter privilege.

Using Templates to Define Extensions

You can use extension templates to define extensions in your user interface plug-in module. The vSphere Web Client SDK provides XML templates that you can include in the `plugin.xml` file. The templates use variable values that you provide to automatically create extension definitions.

The vSphere Web Client SDK provides templates for a number of extension types, including the data views in a vSphere object workspace and for inventory list extensions in the object navigator. You can find examples of templates among the sample material for each extension type. See Chapter 4, “Adding to vCenter Object Workspaces,” on page 37, and Chapter 6, “Extending the Object Navigator,” on page 59.

Adding a Global View Extension 3

In the vSphere Web Client, a global view extension is a way to create a free-form data view. You can use a global view extension to create your own custom solution for the vSphere Web Client user interface. A global view extension can have nearly any function, including aggregating data about different types of vSphere objects onto a single screen, or displaying data from sources outside the vSphere environment.

A global view can be a simple single-level data view that uses the entire vSphere Web Client main workspace, or a complex nested view with its own internal navigation structure and organization. Creating a global view extension has a few restrictions.

Global views are displayed in the vSphere Web Client main workspace, but exist outside of the virtual infrastructure hierarchy. The user selects a global view directly, either through a pointer in the object navigator or a shortcut on the vSphere Web Client home screen.

To create a global view extension, you must define the extension using XML in the plug-in module manifest file, and create the Flex or HTML code that appears in the main workspace.

This chapter contains the following topics:

Use Cases for Creating a Global View Extension

Global view extensions can be used to create dashboard-style data views or console-style applications.

Dashboards

You can use a global view extension to create a dashboard-style data view. A dashboard aggregates data from different sources in the vSphere environment together in one unified data view. For example, you can create a dashboard that provides status information about all custom company-branded objects in the vSphere environment.

Consoles

You can use a global view extension to create a console-style application that is displayed in the vSphere Web Client main content area. The vSphere Web Client Tasks Console and Events Console are examples of this kind of global view.

Defining a Global View Extension

The vSphere Web Client provides an extension point for all global view extensions, named `vise.global.views`. The extension definition must reference the `vise.global.views` extension point, and provide an extension object of type `com.vmware.ui.views.GlobalViewSpec`.

Example 3-1 shows an example extension definition for a global view extension. In the example, the `<extendedPoint>` element specifies the global view extension point. The `<object>` element defines a data object of type `com.vmware.ui.views.GlobalViewSpec`.

Example 3-1. Example Global View Extension

```
<extension id="mySolution.myPlugin.myConsoleApp">
    <extendedPoint>vise.global.views</extendedPoint>
    <object>
        <name>My Console App</name>
        <componentClass className="com.mySolution.myPlugin.MyConsoleApp"/>
        <applyDefaultChrome>true</applyDefaultChrome>
    </object>
</extension>
```

Table 3-1 lists the properties that you provide for the `com.vmware.ui.views.GlobalViewSpec` extension data object, using the `<object>` element in your extension definition.

Table 3-1. GlobalViewSpec Extension Object Properties

Property	Type	Description
`<name>`	string	String that appears as the global view title in the main workspace. In Example 3-1, the value is hard coded as `"My Console App"`. You can also use a dynamic resource string from your plug-in module.
`<componentClass>`	string	Flex class that implements the global view. This Flex class appears in the vSphere Web Client main workspace. If you implement the view in HTML, specify the `HtmlView` container class. You must set the value of the `<componentClass>` element `className` attribute to the fully qualified name of the Flex view class. In Example 3-1, the class is `com.mySolution.myPlugin.MyConsoleApp`.
`<applyDefaultChrome>`	boolean	A Boolean value. If you set this property to `true`, the global view appears with the default header, footer, title, and icon graphics provided by the vSphere Web Client. This property is optional and defaults to `true` if omitted.

Creating Global View Classes

Each global view extension is an independent Flex or HTML element that must communicate with the vSphere environment to retrieve data, or send commands, that the view requires. The vSphere Web Client SDK includes an MVC framework and a Flex library that you can use when creating global view Flex classes. Both Flex and HTML classes can use the Data Access Manager library to retrieve data from the vSphere environment and to send commands to the virtual infrastructure.

You can create a global view class using any pattern or framework, but using the vSphere Web Client SDK MVC framework and Data Access Manager is a best practice for Flex classes. See Chapter 5, “Architecting Data Views,” on page 43.

Making a Global View Accessible to Users

To make your global view extension accessible to users, you must create a pointer to the global view somewhere in the vSphere Web Client user interface. You can create a pointer as either an object navigator extension or a home screen shortcut extension. If you create a pointer in the object navigator, you can add the pointer to either the top level Solutions category, or to the virtual infrastructure level.

See Chapter 6, “Extending the Object Navigator,” on page 59, and Chapter 9, “Creating Home Screen Shortcuts,” on page 87.

Adding to vCenter Object Workspaces

4

The vSphere Web Client displays a standard object workspace for each type of vSphere object. The object workspace is a collection of data views with a tabbed navigation structure. The workspace for a given vSphere object appears in the vSphere Web Client main workspace when the user selects an object when browsing the virtual infrastructure.

Each vSphere object type has a **Summary**, **Monitor**, **Manage**, and **Related Objects** top-level tab view, and may contain additional sub-views within each tab. You can add extensions to create your own sub-views within any of the top-level tab views, or within specific sub-views. You can also create new object workspaces with the default top-level tab and sub-view structure.

This chapter contains the following topics:

Use Cases for Adding to an Object Workspace

You can either add a new data view to the existing object workspace for any type of vSphere object, or you can create an object workspace for a custom vSphere object.

In general, you add a data view extension to an existing object workspace to convey additional information about a vSphere object that is not included in an object's standard workspace. You might need to create an extension to the vSphere Web Client service layer, such as a Data Service adapter, to provide the data for your data view extension. See Chapter 11, "vSphere Web Client Service Layer," on page 93.

The vSphere Web Client SDK provides a set of extension points for each type of object in the vSphere environment, such as a virtual machine or host. Each extension point corresponds to a specific view in that object's workspace, to which you can add your own subordinate view extension.

When you create an object workspace, you use the XML extension templates included with the vSphere Web Client SDK. The XML extension templates create all of the standard top-level tabs and subordinate views in a standard object workspace, as well as defining a set of extension points for each. You then create data view extensions that reference the extension points that the template created to complete your object workspace.

Adding to an Existing Object Workspace

The vSphere Web Client SDK provides a unique extension point for each of the default tabs and subordinate views for every type of object in the vSphere environment. When you define a data view extension that references one of these extension points, your extension appears as a subordinate view inside that tab or view.

Extension points use the following naming convention:

```
vsphere.core.${objectType}.${view}
```

The `${objectType}` value denotes the type of vSphere object for which to extend the object workspace. The `${view}` value specifies the exact view to which to add the extension, such as a top-level tab or specific subordinate view.

For example, if you define an extension that specifies the `vsphere.core.vm.manageViews` extension point, your extension appears as a subordinate view on the **Manage** tab in the object workspace for virtual machine objects.

For a complete list of virtual infrastructure object workspace extension points, see Appendix A, "List of Extension Points," on page 113.

Types of Data Views

A data view extension appears in a slightly different fashion depending on which extension point that you specify in the extension definition. Data views can appear as the following structures.

- **Second-level tabs.** A data view extension appears as a second-level tab if you specify an extension point for a top-level tab, such as **Summary**, **Monitor**, or **Manage** in an object workspace.
- **Second-level tab views.** A data view extension appears as a view within an existing second-level tab if you specify one of the preexisting second-level tab extension points, such as the **Performance** view inside the **Monitor** tab in an object workspace.
- **Portlets.** A data view extension appears as a portlet if you specify the **Summary** tab extension point in an object workspace.

When you design the user interface for your data view extension, keep in mind the extension point where the extension appears. The extension point data view type affects the amount of available screen space and the layout of your data view.

Defining a Data View Extension

An extension to an existing object workspace must specify the target extension point in the extension definition, and provide a data object of type `com.vmware.ui.views.ViewSpec`.

Example 4-1 shows an example extension definition for an extension that adds a data view to the workspace for Host objects. In the example, the `<extendedPoint>` element references the extension point for the **Summary** tab, `vsphere.core.host.summarySectionViews`. The extension appears as a portlet under the **Summary** tab for Host objects. The `<object>` element defines a data object of type `com.vmware.ui.views.ViewSpec`.

Example 4-1. Example Portlet Extension for Host Summary Tab

```
<extension id="mySolution.myPlugin.MySummaryPortlet">
    <extendedPoint>vsphere.core.host.summarySectionViews</extendedPoint>
    <object>
        <name>My Summary Portlet</name>
        <componentClass className="com.mySolution.myPlugin.MyPortletView"/>
    </object>
</extension>
```

Table 4-1, on page 39, lists the properties that you provide for the `com.vmware.ui.views.ViewSpec` extension data object, using the `<object>` element in your extension definition.

Table 4-1. ViewSpec Extension Object Properties

Property	Type	Description
<name>	string	String value that appears as the view title where appropriate in the user interface, such as on the second-level tab button or in the portlet title bar. The value of the <name> property can be a hard-coded string or a dynamic resource included in your plug-in module. In Example 4-1, the string is hard coded as "My Summary Portlet".
<componentClass>	string	Flex class that you created for your data view extension. This Flex class appears at the user interface location corresponding to the extension point you specify. If you implement the view in HTML, specify the HtmlView container class. You must set the value of the <componentClass> element className attribute to the fully qualified name of the Flex view class. In Example 4-1, the class is com.mySolution.myPlugin.MyPortletView.

Creating an Object Workspace for a Custom Object

If you added a new type of object to the vSphere environment, you can create a complete workspace for that object in the vSphere Web Client by using an XML template. You instantiate the XML template in the plug-in module's `plugin.xml` manifest file.

The vSphere Web Client SDK includes a standard object view template for an object workspace. The object view template uses variable values that you provide to create extension definitions and extension points for the object workspace tabs and subordinate views, including **Summary**, **Monitor**, **Manage**, and **Related Objects**.

Using the Object View Template to Create an Object Workspace

To create an instance of the standard object view template, you use the `<templateInstance>` element in your plug-in module's `plugin.xml` file. Inside the `<templateInstance>` element, you define the variables that describe the specific instance of the template.

Example 4-2 shows an example of how to instantiate the object view template for a new object called a Rack. The `<templateInstance>` shown in the example cam appear anywhere inside the root `<plugin>` element in the `plugin.xml` file, but a best practice is to include the template definition with any other extension definitions.

Example 4-2. Instantiating the Standard Template in plugin.xml

```
<templateInstance id="com.vmware.samples.rack.viewTemplateInstance">
      <templateId>vsphere.core.inventory.objectViewTemplate</templateId>
      <variable name="namespace" value="com.vmware.samples.rack"/>
      <variable name="objectType" value="samples:Rack"/>
</templateInstance>
```

In Example 4-2, the `<templateInstance>` element contains an `id` attribute, which is a unique identifier of your choice.

The `<templateId>` element contains the identifier of the template you want to create. To use the standard object view template included with the vSphere Web Client SDK, this identifier must be `vsphere.core.inventory.objectViewTemplate`. To use the `objectViewTemplate`, you must define a `namespace` variable and an `objectType` variable.

Namespace

The `namespace` variable sets the naming convention for the object workspace extension points. When the vSphere Web Client creates the extension points for the object workspace, it prepends the value of the `namespace` variable onto the default extension point names for each data view.

In Example 4-2, the `namespace` variable has a value of `com.vmware.samples.rack`. The vSphere Web Client uses the value to create an extension point for each data view included in the standard object workspace. The extension point for the **Monitor** tab for the Rack object workspace is named `com.vmware.samples.rack.monitorViews`. The other extension points in the Rack object workspace are named using the same convention.

The following list shows the extension points that you create when you instantiate the standard object view template.

namespace`.gettingStartedViews`
namespace`.summaryViews`
namespace`.monitorViews`
namespace`.monitor.issuesViews`
namespace`.monitor.performanceViews`
namespace`.monitor.performance.overviewViews`
namespace`.monitor.performance.advancedViews`
namespace`.monitor.tasksViews`
namespace`.monitor.eventsViews`
namespace`.manageViews`
namespace`.manage.settingsViews`
namespace`.manage.alarmDefinitionsViews`
namespace`.manage.tagsViews`
namespace`.manage.permissionsViews`
namespace`.relatedViews`
namespace`.list.columns`

Object Type

You use the `objectType` variable to associate the object workspace with your custom entity type. Your custom entity type name should include a namespace prefix, such as your company name, to avoid clashing with other object type names in the vSphere environment. The vSphere Web Client displays the object workspace when the user selects an object of the specified type. In Example 4-2, the `objectType` variable has the value `samples:Rack`. The object workspace created by the template appears in the vSphere Web Client when the user selects a Rack object.

Creating the Data Views Within the Template

After you create an object workspace using the standard template, you can create data views at the resulting extension points in the same way that you do for other extension points in the Virtual Infrastructure. See "Defining a Data View Extension" on page 38.

Top-level tabs, second-level tabs, and views created using the standard object view template do not appear in the vSphere Web Client main workspace unless you define a data view extension at that extension point. For example, if you do not define a data view extension at the `com.vmware.samples.rack.monitor.performanceViews` extension point, the **Performance Views** second-level tab in the **Monitor** tab does not appear for Rack objects.

Using the Summary Tab Template

The vSphere Web Client SDK contains a template that you can use to create a standard **Summary** tab view for your custom object. The standard **Summary** tab view contains a Summary header view displayed at the top of the main workspace, and an extension point for any number of Portlet views below the header. The extension point that the template creates for Portlet views is named *namespace*`.summarySectionViews`.

To use the standard Summary tab template, you must create an instance of the template in your plug-in module `plugin.xml` file, using the `<templateInstance>` element. The `<templateInstance>` element can appear anywhere in the root `<plugin>` element, but after the definition for the standard object workspace template.

Example 4-3, on page 41, shows an example of how to instantiate the Summary tab template for a custom object called a Rack.

Example 4-3. Instantiating the Summary Tab Template in plugin.xml

```
<!-- Object Workspace Template for Rack -->
<templateInstance id="com.vmware.samples.rack.viewTemplateInstance">
      <templateId>vsphere.core.inventory.objectViewTemplate</templateId>
      <variable name="namespace" value="com.vmware.samples.rack"/>
      <variable name="objectType" value="samples:Rack"/>
</templateInstance>

<!-- Summary Tab Template for Rack -->
<templateInstance id="com.vmware.samples.chassis.summaryViewTemplateInstance">
    <templateId>vsphere.core.inventory.summaryViewTemplate</templateId>
    <variable name="namespace" value="com.vmware.samples.rack"/>
    <variable name="summaryHeaderView" value="com.vmware.samples.rack.views.RackSummaryView"/>
</templateInstance>
```

In Example 4-3, the `<templateInstance>` element contains an `id` attribute, which is a unique identifier of your choice.

The `<templateId>` element contains the identifier of the template to create. To use the standard Summary tab template included with the vSphere Web Client SDK, this identifier must be `vsphere.core.inventory.summaryViewTemplate`. To use the `summaryViewTemplate`, you must define a `namespace` variable and a `summaryHeaderView` variable.

Namespace

The value you supply for the `namespace` variable must match the `namespace` for the **Summary** tab's parent object workspace. The **Summary** tab created by the Summary tab template uses the extension point *`namespace`*`.summaryViews`, as defined by the object workspace template.

The Summary tab template also creates the extension point *`namespace`*`.summarySectionViews`. You can use the *`namespace`*`.summarySectionViews` extension point to add portlet data views to your custom object **Summary** tab.

In Example 4-3, the `namespace` variable has a value of `com.vmware.samples.rack`, matching the `namespace` variable in the Rack object workspace template.

Summary Header View

You use the `summaryHeaderView` variable to specify the data view that appears at the top of the **Summary** tab. The value you supply for the `summaryHeaderView` variable must be the fully qualified class name for an MXML data view class in your plug-in module. If you omit the `summaryHeaderView` variable, no header appears in the **Summary** tab, and the main workspace displays the portlet data views you create at the *`namespace`*`.summarySectionViews` extension point, if any.

Adding Portlets to the Summary Tab

After you create a **Summary** tab for your custom object workspace by using the Summary tab template, you can add portlet data views to the **Summary** tab. You create the portlet data views at the *`namespace`*`.summarySectionViews` extension point that the template creates, in the same way that you do for any other extension point in the Virtual Infrastructure. See "Defining a Data View Extension" on page 38.

Creating Data View Classes

Each data view extension in an object workspace is an independent Flex or HTML element that must communicate with the vSphere environment to retrieve data, or send commands, that the view requires. The vSphere Web Client SDK includes an MVC framework and a Flex library that you can use when creating data view Flex classes. Both Flex and HTML classes can use the Data Access Manager library to retrieve data from the vSphere environment and to send commands to the virtual infrastructure.

You can create a data view class using any pattern or framework you want, but using the vSphere Web Client SDK MVC framework and Data Access Manager is a best practice for Flex classes. See Chapter 5, "Architecting Data Views," on page 43.

Architecting Data Views

5

When you create a data view extension to the vSphere Web Client user interface layer, you must provide user interface classes for the GUI elements that appear in the main workspace area of the interface. Data view extensions that require you to provide classes include global view extensions, extensions to existing object workspaces in the virtual infrastructure browser, and custom object workspaces that you create.

Flex classes for GUI elements in the vSphere Web Client are implemented using the MVC software architecture pattern. In the MVC pattern, the business logic and graphics components of a GUI element are separated into different classes. The Flex classes in vSphere Web Client data views use an internal MVC framework called Frinje to create the associations between the classes used in the MVC pattern.

The vSphere Web Client SDK provides an event-driven API, based on the Frinje framework, called the Data Access Manager. Data view classes can use the Data Access Manager library to query data from sources inside or outside of the vSphere environment.

You can use any framework to build vSphere Web Client data view extensions in Flex, but using the Frinje framework is a best practice. By using the Frinje framework and implementing a compatible MVC architecture, your extensions can use the built-in Data Access Manager library provided with the vSphere Web Client SDK to retrieve data and send commands to the virtual infrastructure.

The MVC architecture pattern applies only to vSphere Web Client extensions that add data views to the GUI. Other extensions, such as shortcuts for the home screen or object navigator, are defined only in metadata and do not require you to create MVC code.

This chapter contains the following topics:

About the MVC Architecture

The Model-View-Controller architecture separates the domain logic of the software from the user interface elements. This decoupling of components allows each part of the software to be developed, tested, and maintained independently.

In the MVC architecture, the model component manages the business logic of the software, including calculations, data access, and communications with external data sources. The view component manages only the graphics-related functions of the user interface. The controller component contains the control logic that governs communication between the model component and the view component.

The model, view, and controller components can be implemented as a single class or as a collection of different classes.

The Frinje Framework MVC in the vSphere Web Client

The vSphere Web Client uses the Frinje framework to implement the concepts of the MVC architecture. Frinje uses Flex to build on the MVC architecture in several ways by including additional features.

Understanding how Frinje implements MVC is essential to reading and making use of the sample data view extensions code written in Flex and included with the vSphere Web Client SDK and the associated tutorials. The vSphere Web Client SDK contains a plug-in for the Eclipse development environment that can create Flex and Java projects for a vSphere Web Client plug-in module.

The Frinje framework uses metadata annotations to describe the relationships between the different classes that make up the MVC model. Frinje does not require you to subclass specific foundational classes to use the functionality of the Frinje framework. This results in source code with no direct dependency on the Frinje framework, so the same class definitions can be compiled into any Flex environment.

View Components in Frinje

Frinje implements the View component of the MVC architecture as two separate classes. The View component's graphic elements are implemented in an MXML view class, and the View component's logic elements are implemented in an ActionScript mediator class. The Frinje framework refers to the MXML class as a view class, and the ActionScript class as a mediator class.

Overview of View and Mediator Classes

The view class and the mediator class are associated using metadata tags. The MXML view class uses the `[DefaultMediator]` tag to specify the associated mediator class. Example 5-1 shows a sample MXML view class with an associated mediator. The name of the associated mediator class in the example is `com.acme.myPlugin.views.MyPluginMediator`.

Example 5-1. Sample Frinje View Class

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Box xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Metadata>
        [DefaultMediator("com.acme.myPlugin.views.MyPluginMediator")]
    </mx:Metadata>

        <mx:Label text="Hello World!"/>
        <mx:Button id="showSelected" label="Show Selected"/>

</mx:Box>
```

The ActionScript mediator class uses the `[View]` tag to specify the associated view class. Example 5-2 shows a sample ActionScript mediator class with functions to set and retrieve the associated view. The name of the associated view class in the example is `MyPluginView`.

Example 5-2. Sample Frinje Mediator Class

```
public class MyPluginMediator extends EventDispatcher
{
    private var _view:MyPluginView;

    [View]
    /** The view associated with this mediator. */
    public function set view(value:MyPluginView):void {
        _view = value;
    }
    public function get view():MyPluginView {
        return _view;
    }
    ...
```

```
}
```

The view class contains the graphic elements that the user sees and clicks, such as buttons, text fields, and menus. The mediator class contains the logic that drives those elements, for example the actions taken when a user clicks a particular button. A mediator class interacts with the other elements in the Frinje framework by dispatching events in response to user actions. To gain this functionality, each mediator class must extend the `EventDispatcher` base class.

Creating View Components with the vSphere Web Client SDK

When you create data view or global view extensions to the vSphere Web Client, you must provide the UI view that appears in the main workspace, and the mediator class that contains the interaction logic for that view.

The view class is typically an MXML class that encapsulates the visual Flex components for the data view. The view class specifies the data view mediator using the `[DefaultMediator]` annotation, and contains little business logic, if any.

The mediator class in a vSphere Web Client data view contains the logic used to populate the view components, make data requests, and handle data responses. The vSphere Web Client SDK provides libraries and interfaces that your mediator class can use to obtain the object that the user has currently selected in the vSphere Web Client, and to request data from the vSphere environment.

Obtaining the Currently Selected vSphere Object

Some data views, such as those you add to the object workspace for a specific vSphere object type, need to track which object the user has selected using the object navigator. For example, if you add a data view to the object workspace for Virtual Machine objects, that data view must track which Virtual Machine the user has selected to request the appropriate data for display.

Your mediator class can use the Frinje framework to obtain the object that the user currently has selected in the object navigator. To make use of this functionality, your mediator class must implement the `com.vmware.ui.IContextObjectHolder` interface.

When your mediator class implements the `IContextObjectHolder` interface, you must provide `get` and `set` methods for the `contextObject` property. You do not need to set the `contextObject` property directly. The Frinje framework calls the mediator class `set contextObject()` method when the view becomes active. The currently selected vSphere object is passed to `set contextObject()` as the parameter `value`.

A best practice is to place your data view's initial data requests and other initialization code in the `set contextObject()` method. That way, you can initialize the view when the framework passes the currently selected object as the view becomes active.

Example 5-3, on page 46, shows an example mediator class. The example mediator contains the logic for a data view for a custom object type called Chassis. The example mediator implements the `IContextObjectHolder` interface, and uses `set contextObject()` to obtain the currently selected Chassis object.

Example 5-3. Example Mediator Class Implementing IContextObjectHolder

```
public class ChassisSummaryViewMediator extends EventDispatcher implements IContextObjectHolder {
    private var _view:ChassisSummaryView;
    private var _contextObject:IResourceReference;

    public function set contextObject(value:Object):void {
        _contextObject = IResourceReference(value);
        if (_contextObject == null) {
            clearData();
            return;
        }
        // Initialize the view's data once the current context is known.
        requestData();
    }

    [Bindable]
    public function get contextObject():Object {
        return _contextObject;
    }
    ...
}
```

In the example, the framework invokes the `set contextObject()` method, which in turn sends an initial data request, or clears the interface if no object is selected.

Requesting Data from the vSphere Environment

To obtain the data needed to populate the view, your mediator class must communicate with data sources. These data sources can be in vSphere, or they can be external Web sources. Your mediator class can use the Data Access Manager Flex library in the vSphere Web Client SDK to request data from either type of data source. To use Data Access Manager, your mediator must use the DAM API, which is based on Frinje events. See "About the Frinje Event-Driven APIs" on page 48.

Your mediator class can also communicate with a data source by importing a Java service that you create. If you create a Java service and add it to the vSphere Web Client Virgo server framework, you can use an ActionScript proxy class to access the service from your mediator class. See "Model Components in Frinje" on page 46.

Model Components in Frinje

Frinje implements the Model component of the MVC architecture as an Actionscript proxy class and associated data objects. The proxy class handles communication with Java services that run on the vSphere Web Client Virgo server framework, as part of the vSphere Web Client service layer.

For most data view extensions that you create, you do not need to implement the proxy class yourself. The vSphere Web Client SDK provides a Flex library called the Data Access Manager, which handles all communications tasks with the Java services running in the vSphere Web Client service layer. For more information about the Data Access Manager library, see "Using the Data Access Manager Library" on page 51.

You only need to implement the proxy class if you created your own custom Java service and added that service to the vSphere Web Client service layer. Any proxy class you create must extend the `com.vmware.flexutil.proxies.BaseProxy` class in the vSphere Web Client SDK. Example 5-4, on page 47, shows a sample proxy class that calls a simple "echo" service in the vSphere Web Client service layer.

Example 5-4. Sample Frinje Proxy Class

```
public class EchoServiceProxy extends BaseProxy {
   private static const SERVICE_NAME:String = "EchoService";

   // channelUri uses the Web-ContextPath define in MANIFEST.MF (globalview-ui)
   private static const CHANNEL_URI:String =
         "/" + GlobalviewModule.contextPath + "/messagebroker/amfsecure";

   /**
    * Create a EchoServiceProxy with a secure channel.
    */
   public function EchoServiceProxy() {
      super(SERVICE_NAME, CHANNEL_URI);
   }

   /**
    * Call the "echo" method of the EchoService java service.
    *
    * @param message    Single argument to the echo method
    * @param callback   Callback in the form <code>function(result:Object,
    *       error:Error, callContext:Object)</code>
    * @param context    Optional context object passed back with the result
    */
   public function echo(message:String, callback:Function = null, context:Object = null):void {
      callService("echo", [message], callback, context);
   }
```

Controller Components in Frinje

Frinje implements the Controller component of the MVC architecture using the Frinje event bus and command classes.

Frinje Event Bus

The Frinje event bus is a global event bus that extends and improves the capabilities of the native Flex event model. Rather than requiring each component class to dispatch and receive events directly to and from one another, the Frinje event bus lets you use metadata annotations to specify the events relevant to your classes. The Frinje event bus routes events for your application to classes that are registered to subscribe to those events.

The view, mediator, command, and proxy classes in the Frinje framework work together with the Frinje event bus to drive the vSphere Web Client user interface. Classes that dispatch events, typically views and mediators, use metadata annotations to specify the events they generate. Likewise, classes that handle events, typically command classes, use metadata annotations to specify the events they can receive. The Frinje event bus is responsible for routing the events and instantiating the receiving class at runtime.

By using the Frinje event bus, event-dispatching classes, such as views or mediators, and event-handling classes, such as command classes, can be developed independently without those classes explicitly referencing one another.

Command Classes

In the Frinje framework, a command class is responsible for handling events generated by changes in the state of the user interface. When the user clicks a button, for example, the View containing that button dispatches an event on the Frinje event bus. A command object is instantiated to handle that event using the appropriate business logic.

Events can be dispatched from any part of the Frinje framework, but are commonly generated from view or mediator classes in response to user actions. For example, if the user clicks an action for which you previously created an action extension, the vSphere Web Client generates a Frinje event using the appropriate action ID. As part of your action extension, you must provide a command class with a method annotated to handle the generated Frinje event. See "Organizing Your Actions in the User Interface" on page 75.

About the Frinje Event-Driven APIs

Some libraries and frameworks in the vSphere Web Client SDK, specifically the Data Access Manager (DAM) and the Actions Framework, make use of the Frinje framework. These features provide APIs in the vSphere Web Client SDK that are based on Frinje events. The DAM, for example, provides specific events that are sent over the Frinje event bus, and you use the DAM by sending and receiving these events.

A mediator class can use the Data Access Manager library by dispatching the request events in the DAM API, and handling the response events. The classes do not need to explicitly include the libraries or subclass any specific class. They need only the proper Frinje metadata annotations to make use of the Frinje event bus.

Frinje Metadata Annotations

Your classes and methods make use of the Frinje framework by including metadata annotations with the class or method declaration. You can use the following metadata annotations when constructing your classes. Each annotation has certain required attributes you must include.

- `[DefaultMediator]`
- `[Event]`
- `[Model]`
- `[RequestHandler]`
- `[ResponseHandler]`
- `[EventHandler]`
- `[View]`

IMPORTANT String arguments in metadata annotations are not validated at compile time. Review your metadata annotations carefully if you encounter errors or your plug-in modules do not function correctly. In addition, it is a best practice to use the fully qualified class name for any class arguments included in your metadata annotations.

DefaultMediator

You use the `[DefaultMediator]` tag to declare the mediator class associated with a particular view component. You typically use this tag in your view MXML class, to specify the ActionScript class that contains the UI logic. The `[DefaultMediator]` tag has one argument, which must be the fully qualified class name for the mediator class.

```
[DefaultMediator("fully_qualified_class_name")]
```

To use the `[DefaultMediator]` tag in an MXML class, you must enclose the tag in `<mx:Metadata>` elements. Example 5-5 shows an example `[DefaultMediator]` tag.

Example 5-5. Example [DefaultMediator] Metadata Tag in MXML view

```
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Metadata>
        [DefaultMediator("com.vmware.samples.chassisui.views.ChassisSummaryViewMediator")]
    <mx:Metadata>
...
```

In the example, the `[DefaultMediator]` tag declares the `ChassisSummaryViewMediator` class as the mediator class for the view described in MXML.

Event

You use the `[Event]` tag to define any events that your class generates. You must insert the `[Event]` tag before the ActionScript class definition for any class that dispatches events.

The [Event] tag is a native Flex metadata tag that contains the arguments name and type. You use the name argument to specify the name of the event that your class can dispatch, and the type argument to specify the data type for the event object.

```
[Event(name="event_name", type="event_type")]
```

In the vSphere Web Client SDK, you typically use the [Event] tag to specify events from the SDK's Data Access Manager library. Your classes can dispatch these events to request data from the vSphere environment. See "Using the Data Access Manager Library" on page 51.

Example 5-6 shows an example mediator class annotated to dispatch a Data Access Manager event. The event class is included in the com.vmware.data.query library in the vSphere Web Client SDK.

Example 5-6. Example [Event] Metadata Tag in Mediator Class

```
// Declares the data request event sent from this UI class.
[Event(name="{com.vmware.data.query.events.DataByModelRequest.REQUEST_ID}",
    type="com.vmware.data.query.events.DataByModelRequest")]
public class ChassisSummaryViewMediator extends EventDispatcher implements IContextObjectHolder {
...

private function requestData():void {
    // Dispatch an event to fetch the _contextObject data from the server along the specified
                model.
    dispatchEvent(DataByModelRequest.newInstance(_contextObject, ChassisSummaryDetails));
}
...
```

In the example, the name attribute in the [Event] tag has the value {com.vmware.data.query.events.DataByModelRequest.Request_ID}. The REQUEST_ID corresponds to a specific event identifier defined in the DataByModelRequest class, rather than a hard-coded event name.

Model

You use the [Model] tag to annotate a data model class. Data model classes are used to specify information being retrieved through the vSphere Web Client SDK Data Access Manager library. See "Using the Data Access Manager Library" on page 51.

RequestHandler

You use the [RequestHandler] tag to annotate a method to handle a particular action in the vSphere Web Client. Typically, you create a command class for your action and annotate a method in that command class with the [RequestHandler] tag.

The [RequestHandler] tag has one parameter, which is the UID for the action that the method handles. The action UID in the [RequestHandler] tag must match the action UID that you specified in the action's extension definition. See Chapter 7, "Creating Action Extensions," on page 67.

```
[RequestHandler("action_uid")]
```

ResponseHandler

You use the [ResponseHandler] tag to annotate a method to handle a specific type of event generated by the Data Access Manager library in response to a data request.

The method you annotate with the [ResponseHandler] tag listens for specific, named events that are dispatched from its parent component class. When writing a UI component class, such as a mediator, you typically annotate the class with the [Event] tag to specify the events named events that the class can generate. You can then annotate specific methods within that class with [ResponseHandler] to handle each event.

The [ResponseHandler] tag has a single argument, which you use to specify the event name that the method expects.

```
[ResponseHandler(name="response_event_name")]
```

The method you annotate with `[ResponseHandler]` must accept the parameters `request` and `result`, representing the type of data request and the result of the data request, respectively. The `request` type must match that of the dispatched event. The method you annotate with `[ResponseHandler]` can also accept the optional parameters `error` and `resultInfo`.

Example 5-7 shows an example method annotated to handle a Data Access Manager data response event. The event class is included in the `com.vmware.data.query` library in the vSphere Web Client SDK.

Example 5-7. Example [ResponseHandler] Metadata Tag in Event Handler Method

```
[ResponseHandler(name="{com.vmware.data.query.events.DataByModelRequest.RESPONSE_ID}")]
public function onData(request:DataByModelRequest, result:ChassisSummaryDetails):void {
    _view.chassisDetails = result;
}
```

In the example, the `onData` function is annotated to receive the event `com.vmware.data.query.events.DataByModelRequest.Response_ID`. The event must be generated from the `onData` function's parent class, and the parent class must be annotated with an `[Event]` tag that specifies that the class dispatches the `com.vmware.data.query.events.DataByModelRequest.RESPONSE_ID` event.

EventHandler

You use the `[EventHandler]` tag to annotate a method to handle a general, application-wide notification event. An example of such an event is the `DataRefreshInvocationEvent` that is generated when the user clicks the global refresh button in the vSphere Web Client UI.

The `[EventHandler]` tag has one argument, which specifies the name of the event for which the method is listening.

```
[EventHandler(name="event_name")]
```

The method you annotate with `[EventHandler]` must accept an `event` parameter. The `event` parameter contains the generated event.

You can annotate a method with `[EventHandler]` to handle data responses from the Data Access Manager. However, a method annotated with `[ResponseHandler]` is more suited to handling data requests. The Frinje framework extracts the request type and result data automatically for `[ResponseHandler]` methods.

Example 5-8 shows an example method annotated to handle the global `DataRefreshInvocationEvent`.

Example 5-8. Example [EventHandler] Metadata Tag in Event Handler Method

```
[EventHandler(name="{com.vmware.core.events.DataRefreshInvocationEvent.EVENT_ID}")]
public function onGlobalRefreshRequest(event:DataRefreshInvocationEvent):void {
    requestData();
}
```

View

You use the `[View]` tag to inject a view component object into the view's associated mediator class. When you use the `[View]` tag, you can reference the view component with a generic view variable, and the Frinje framework will associate the mediator with the specific view at creation time. The associated view must be annotated with the `[DefaultMediator]` tag for the framework to make the association. See "DefaultMediator" on page 48.

Example 5-9, on page 50, shows an example of a mediator class that declares a generic variable for the view class, and uses the `[View]` tag to inject the view component in the mediator `get` and `set` methods for the view.

Example 5-9. Example [View] Metadata Tag in Mediator Class

```
/**
 * The mediator for the ChassisSummaryView view.
 */
public class ChassisSummaryViewMediator extends EventDispatcher implements IContextObjectHolder {
```

```
    private var _view:ChassisSummaryView;

    [View]
    public function get view():ChassisSummaryView {
        return _view;
    }

    public function set view(value:ChassisSummaryView):void {
        _view = value;
    }
...
```

Using the Data Access Manager Library

When you create your data view Flex classes using the MVC architecture pattern and Frinje framework, you can use the Data Access Manager library included in the vSphere Web Client SDK. The DAM library is a Flex library that communicates with the vSphere Web Client Data Service. Using DAM and the Data Service is the principal way to retrieve information about objects in the vSphere environment.

In general, the mediator class in a data view extension uses the DAM library to request data from the vSphere environment. The mediator class then updates the extension's view class to reflect the new information. The DAM also includes a data refresh mechanism that you can use to handle data changes implicitly. You can include a data update specification with your data request, so that DAM can assign updated data to GUI elements in your view class without the need for extra code to do so.

Data view extensions implemented in HTML communicate with the Data Service indirectly. To use the Data Service library from an HTML extension, you must install an adapter process on the Virgo application server. For more information about HTML bridge adapters, see "Creating Action Extensions" on page 67.

Data Access Manager Workflow

The DAM API consists of Frinje events, such as requests for data or responses to data requests. To use DAM, your Flex classes must dispatch the request events and listen for the associated responses. You indicate which events your classes can receive by using Frinje metadata annotations.

To use the Data Access Manager library, you must build your extension Flex classes as follows.

1 Create an annotated data model class in your plug-in module. The data model class must extend the `DataObject` base class. The data model class describes the information that the extension's mediator class must retrieve using the DAM. The mediator references the data model class when dispatching data request events or receiving data response events.

   To query for a single property of a given vSphere object, you do not need to create a data model class, and can proceed to Step 2.

2 In your extension's mediator class, create and dispatch one or more data request events. Data request events include both a reference to the target vSphere object, and a data model class as described in Step 1. Optionally, the data request can also include a data update specification that will cause the DAM to send notifications whenever the values of the requested properties change.

   For data requests events that retrieve a single property, you specify the property name using a `string` value and do not need to include a data model class.

3 The DAM sends a data response event, which contains a `DataObject` with the values of the requested properties, as specified in the data model class in Step 1. You must annotate a method in your extension's mediator class to receive the data response event.

4 In your data response event handler method, update your extension's user interface elements using the data in the data response event.

Creating the Data Model Class

You create a data model class in your extension to describe the information needed from the DAM. When you create a data model class, you specify the type of vSphere or custom object to which the request pertains, and the specific properties of that object that you need to retrieve. For example, you can create a data model class to request information on a Virtual Machine object, and include the `name` and `powerState` properties in the data request.

Example 5-10 shows an example data model class.

Example 5-10. Data Model Class

```
package com.myExtension.model {
    import com.vmware.core.model.DataObject

    // data model class annotation and declaration
    [Model(type="VirtualMachine")]
    public class MyVmData extends DataObject {

        //properties to retrieve, specified by annotation
        [Model(property="name")]
        public var myVmName:String;

        [Model(property="runtime.powerState")]
        public var myVmPowerState:String;
    }
}
```

Class Declaration

The data model class must extend the `DataObject` class and use the `[Model]` annotation tag. Typically, you create the data model class as a separate class or package in your plug-in module. This package must import the package `com.vmware.core.model.DataObject` from the vSphere Web Client SDK.

The `type` tag in the `[Model]` annotation is optional. If you specify the `type` tag in the annotation, the data model class can retrieve properties only for objects of the specified type. You can create a data model class to request common properties for multiple object types, or you can use the `type` tag in the `[Model]` annotation to make the data model class specific to a particular vSphere object type.

In Example 5-10, the class `MyVmData` has a `[Model]` annotation and extends `DataObject`. In the example, the object type `VirtualMachine` is specified, indicating that the data model retrieves information on Virtual Machine objects.

Declaring the Properties to Retrieve

In the data model class, you must specify a class property for each data property to retrieve through DAM. When you declare each property, you must include a `[Model]` annotation with a `property` tag. The `[Model]` annotation on a property differs from the `[Model]` annotation on the class.

The `[Model]` annotation on a property associates the property with a data property belonging to a vSphere object. The `property` tag of the annotation specifies the name of the property to retrieve from the vSphere object. When you use the data model class in a data request, DAM retrieves the property specified in the annotation from the vSphere environment, and assigns the value to the associated class property.

When you use the data model class in a data request, the DAM returns values only for class properties that have the `[Model]` annotation. Class properties without the annotation are ignored.

In Example 5-10, the class `MyVmData` includes the `myVmName` and `myVmPowerState` annotated properties. Each property includes an annotation that specifies to the DAM the vSphere property to associate with the class property.

Retrieving Properties for Related Objects

You can specify and annotate class properties to retrieve data on related objects, such as the host associated with a given virtual machine object. You use the `relation` tag in the `[Model]` annotation for the class property to specify that the property requested from the DAM is for a related object.

Example 5-11 shows a data model class with a property annotated for the DAM to retrieve data from a related object. In the example, the class property `myVmHostName` is annotated for the DAM to retrieve the `name` property of the vSphere host object related to the target virtual machine object.

Example 5-11. Data Model Class with a Related Object Property Annotation

```
[Model(type="VirtualMachine")]
public class MyVmData extends DataObject {
    [Model(relation="runtime.host", property="name")]
    public var myVmHostName:String;
}
```

You can retrieve properties over multi-hop relationships by specifying each hop separated by a comma. Example 5-12 shows a property annotated to retrieve data over a multi-hop relationship. In the example, the `myDatacenterName` property is annotated to retrieve the `name` property for a datacenter object associated with the host for a given virtual machine.

Example 5-12. Data Model Class with a Multi-hop Relationship Annotation

```
[Model(type="VirtualMachine")]
public class MyVmData extends DataObject{
    [Model(relation="runtime.host,cluster", property="name")]
    public var myDatacenterName:String;
}
```

You can also retrieve properties for an object relationship with multiple cardinality, where the relationship is one-to-many. For example, one such relationship is that of a virtual machine object to multiple related network objects. When you retrieve a related property of this type, the associated class property in the data model class must be of type `Array` or `ArrayCollection`.

Nested Data Models

The annotated properties in a data model class can be single properties, or they can be nested data model classes. To specify a class property as a nested data model class, as opposed to a single property, you omit the `property` tag from the `[Model]` annotation when declaring the class property.

Omitting the `property` tag implicitly declares that the class member variable is a data model of the same object type as the parent data model. If the `relation` tag is specified, however, the class property is assumed to be a data model for the object type specified in the `relation` tag.

Example 5-13 shows a data model class with two class properties that are nested data models. The first property, `myVmHardwareData`, is another data model for Virtual Machine objects, the same as the parent data model. The second property, `myHostDetails`, is a data model for Host objects.

Example 5-13. Nested Data Model Classes

```
[Model(type="VirtualMachine")]
public class MyVmData extends DataObject{
    [Model]
    public var myVmHardwareData:VmHardwareData; // VmHardwareData is a VM data model
    [Model(relation="runtime.host")]
    public var myHostDetails:HostData; // HostData is a Host data model
}
```

You can use a nested data model to create an array of data model objects. Example 5-14 shows a data model class that uses a nested data model to create an array of objects based on the Host data model.

Example 5-14. Nested Data Model Used to Create Array

```
[Model(relation="runtime.host")]
public var HostList:Array; // array of Host objects using the HostData model
```

IMPORTANT If the related object for which you want to retrieve data is a user-defined custom object, you must use a different syntax in your `[Model(relation="...")]` annotation. Your `[Model]` annotation must explicitly specify both the custom object type, and the nested data model that you use to define that type. Example 5-15 shows a data model class that uses a nested data model for a user-defined custom Chassis object.

Example 5-15. Related Data Model for Custom Object Type

```
[Model(relation="Chassis as comexample:Chassis", nestedModel="com.example.ChassisModelData"]
public var myChassisDetails:ChassisModelData; // Chassis object using the ChassisModelData
               model
```

In the example, the `[Model(relation="...")]` annotation specifies the exact object type for `Chassis as comexample:Chassis`. The annotation also includes an explicit `nestedModel` attribute that specifies the fully-qualified class name for the Chassis data model. Use this syntax for any custom object type that you include in a data model class.

Sending Data Requests and Handling Data Responses

Your extension's mediator class can use the DAM API to create requests for data. Your mediator class creates data requests by dispatching one or more data request Frinje events specified in the API. The DAM responds to each data request by generating a data response event. You can annotate a method in your mediator class to handle the data response event and update the associated view class with the new information.

Data Request Events

The DAM API contains several different kinds of data request events. Requests can retrieve a single property for a single object in the vSphere environment, or requests can retrieve multiple properties for an object by using a data model class to specify those properties. Requests can be targeted to a specific vSphere object, or you can specify a constraint to request data on all objects that match your defined criteria.

Your mediator requests data from DAM by dispatching data request events. To dispatch the events in the DAM API, your mediator class declaration must include an `[Event]` annotation for each type of event it dispatches, and the class must extend the base SDK class `EventDispatcher`. See Example 5-16, on page 56. In the class methods, you use the `dispatchEvent()` method to dispatch each data request event.

For data requests that retrieve a single property, you must specify the name of the property to retrieve using a `String` type.

Data Response Events

The DAM can return either a response with data about a single vSphere object, or a response with data about multiple vSphere objects. The DAM always returns one of these two responses, depending on the type of request. For requests sent using a data model class, the DAM returns a data response with an object of the data model class type, or an array of such objects for multiple objects. For requests sent for a single property, the DAM returns the property value in a generic `DataObject` wrapper, or an array of generic `DataObjects` for multiple objects.

Your mediator class can listen for, and handle, data response events by annotating a class method with the `[ResponseHandler]` annotation. You specify the type of event for which the method listens in the `[ResponseHandler]` annotation. See Example 5-16, on page 56.

Request-to-Response Mapping

Table 5-1 contains the types of data request events, the resulting data response events, and the associated result types included in the DAM API.

Table 5-1. Data Request and Data Response Events

Request Event	Result Type	Description
`DataByModelRequest`	Instance of specified data model class	Get multiple properties for a given vSphere or custom object, specified by a data model class.
`PropertyRequest`	Appropriate subtype of `DataObject` with the property value in the `value` field	Get a single property for a given vSphere or custom object. The property name is specified using a String value.
`DataByConstraintRequest`	`ArrayCollection` of instances of specified data model class	Get properties specified by data model for vSphere or custom objects that match the given constraint.
`DataByQuerySpecRequest`	`ArrayCollection` of `ObjectDataObject` (property-value map)	Get data specified by query specification.

Examples of Data Request and Data Response Events

Example 5-16, on page 56, shows an example mediator class that interfaces with the DAM by dispatching data request events and handling data response events. The class `MyDataViewMediator` extends the `EventDispatcher` base class, and the declaration has an `[Event]` annotation for each of the data request events that the class can dispatch.

Each of the methods that the class uses to handle data response events contains a `[ResponseHandler]` annotation with the method declaration.

Example 5-16. Example Mediator Class Using DAM Interface

```
[Event(name="{com.vmware.data.query.events.DataByModelRequest.REQUEST_ID}",
               type="com.vmware.data.query.events.DataByModelRequest")]
[Event(name="{com.vmware.data.query.events.PropertyRequest.REQUEST_ID}",
               type="com.vmware.data.query.events.PropertyRequest")]
[Event(name="{com.vmware.data.query.events.DataByConstraintRequest.REQUEST_ID}",
               type="com.vmware.data.query.events.DataByConstraintRequest")]

public class MyDataViewMediator extends EventDispatcher {

    // Requests
    private function requestData(event:Event): void {

        // data-model request event
        var VmDataRequest:DataByModelRequest = DataByModelRequest.newInstance(vmRef, MyVmData);
        // vmRef is the target object reference; MyVmData is the data model class
        dispatchEvent(VmDataRequest);

        // single-property request event
        var VmGuestInfoRequest:PropertyRequest = PropertyRequest.newInstance(vmRef, "guest");
        // vmRef is the target object reference; single property specified as String
        dispatchEvent(VmGuestInfoRequest);

        // request by constraint
        var VmListRequest:DataByConstraintRequest = DataByConstraintRequest.newInstance(
                       QuerySpecUtil.createConstraintForRelationship(hostRef, "vm"), MyVmData);
        dispatchEvent(vmListRequest);
    }

    // Responses
    [ResponseHandler(name="{com.vmware.data.query.events.DataByModelRequest.RESPONSE_ID}")]
    public function onMyVmDataRetrieved(request:DataByModelRequest, result:MyVmData):void {
    // use MyVmData result
    }
    [ResponseHandler(name="{com.vmware.data.query.events.PropertyRequest.RESPONSE_ID}")]
    public function onVmGuestInfoRetrieved(request:PropertyRequest, result:StringDataObject):void
                   {
    // use string result from result.value
    }
    [ResponseHandler(name="{com.vmware.data.query.events.DataByConstraintRequest.RESPONSE_ID}")]
    public function onVmListRetrieved(request:DataByConstraintRequest,
                   result:ArrayCollection):void {
    // result is an ArrayCollection of MyVmData objects
    }
}
```

In the example, the `MyDataViewMediator` class dispatches a `DataByModelRequest` event, a `PropertyRequest` event, and a `DataByConstraintRequest` event in the `requestData` method.

The `DataByConstraintRequest` specifies the constraint using the `createConstraintForRelationship` method. In the example, the constraint specifies the virtual machine objects related to the host object specified by `hostRef`, which represent all virtual machines related to the given host.

Error Checking and Troubleshooting Your Data Requests

Data response events contain an additional `error` property that you can use to obtain more information about the results of a failed data request. The `error` property is available as an optional parameter to the response handler method. Example 5-17, on page 57, shows a code excerpt that demonstrates the use of this property.

Example 5-17. Example Mediator Class Using `error` Property

```
[Event(name="{com.vmware.data.query.events.DataByConstraintRequest.REQUEST_ID}",
               type="com.vmware.data.query.events.DataByConstraintRequest")]

public class MyDataViewMediator extends EventDispatcher {
    // Skip request method; show only response method.
    [ResponseHandler(name="{com.vmware.data.query.events.DataByConstraintRequest.RESPONSE_ID}")]
    public function onDataRetrieved(request:DataByConstraintRequest,
                                    result:ArrayCollection,
                                    error:Error):void {
        if (error != null) {
            _logger.debug("onDataRetrieved error: " + error.message);
            return;
        }
        // Valid result has been received.
        _view.datastoreItems = result;
    }
}
```

For more information, see the ActionScript API reference included with the vSphere Web Client SDK.

Data Refresh and Data Update Specifications

You can use the DAM's data refresh mechanism to subscribe your component to receive updated data from the vSphere environment. You can bind the requested data directly to elements in the view class, causing those elements to be updated whenever the requested data changes. To perform this binding, you must include a data update specification when you dispatch your data request event.

A data update specification is included in the SDK base class `DataRequestInfo`. You must create an instance of `DataRequestInfo` and set the `DataUpdateSpec` property, and then pass the `DataRequestInfo` object when you create your data request. See Example 5-18, on page 58.

The DAM data refresh mechanism has different modes of operation. You specify the mode of operation with your choice of constructor for the `DataUpdateSpec` object that you pass to your `DataRequestInfo` object.

- `implicit`. The DAM refreshes the data when the user performs an operation on the relevant object, or when the user clicks on the vSphere Web Client global **Refresh** button.
- `explicit`. The DAM refreshes the data only if explicitly requested, such as when the user clicks the vSphere Web Client global **Refresh** button.

To create an implicit binding, you use the `newImplicitInstance()` method to construct the `DataUpdateSpec` object. To create an explicit binding, you use the `newExplicitInstance()` method.

Example 5-18, on page 58, shows an example mediator class that includes the `DataRequestInfo` and `DataUpdateSpec` objects to create an automatic binding in the data request. In the example, the `DataUpdateSpec` is constructed using the `newImplicitInstance()` method, resulting in an implicit binding.

Example 5-18. Mediator Class Using DAM Binding Specification for Data Refresh

```
import com.vmware.data.query.events.DataByModelRequest;
import com.vmware.data.query.DataUpdateSpec;
import com.vmware.data.query.events.DataRequestInfo;

[Event(name="{com.vmware.data.query.events.DataByModelRequest.REQUEST_ID}",
               type="com.vmware.data.query.events.DataByModelRequest")]

public class MyVmViewMediator extends EventDispatcher {
    private function onStart():void {
        var requestInfo:DataRequestInfo =
            new DataRequestinfo(DataUpdateSpec.newImplicitInstance());
            // create a DataRequestInfo with an implicit-mode DataUpdateSpec

        var myRequest:DataByModelRequest = DataByModelRequest.newInstance(
            vmRef,
            MyVmData, // data model
            requestInfo); // include the DataRequestInfo object when constructing the request
        dispatchEvent(myRequest);
    }

    [ResponseHandler(name="{com.vmware.data.query.events.DataByModelRequest.RESPONSE_ID}")]
    public function onDataRetrieved(request:DataByModelRequest, result:MyVmData):void {
        _view.vmData = result;
    }
}
```

In the example, the function `onStart` dispatches the initial `DataByModelRequest.` The initial request includes a `DataRequestInfo` object with an implicit `DataUpdateSpec`. The function `onDataRetrieved` is annotated to handle the response, and updates a UI control with the returned data. Because of the data update specification, the DAM calls `onDataRetrieved` when the data changes.

Extending the Object Navigator 6

The object navigator is the user's primary navigation interface for reaching solutions and applications in the vSphere Web Client, and for finding and focusing on objects in the virtual infrastructure. The object navigator control contains a collection of nodes, which can be simple pointers to global views and other solutions that can appear in the vSphere Web Client main workspace. Nodes can also be expandable collections for vSphere object types.

The object navigator home page appears when the user logs in to the vSphere Web Client, and contains a pointer node for the home screen, an entry point to your environment's virtual infrastructure through vCenter, and pointer nodes for other major solutions in the vSphere Web Client. The object navigator Administration page appears when the user clicks the Administration application, and contains categories and pointer nodes specific to that application. The object navigator vCenter page appears when the user browses the objects in the virtual infrastructure. On the vCenter page, the object navigator displays the available objects in the vSphere environment in collections, such as inventory tree nodes and inventory list nodes.

You can extend the object navigator by creating new nodes and categories on each page. You can add both pointer nodes and object collection nodes to the object navigator control.

This chapter contains the following topics:

Use Cases for Extending the Object Navigator

Extensions to the object navigator provide access to other extensions or areas of interest in the vSphere Web Client, such as global views and object workspaces. You create different object navigator extensions depending on what access you want to give the user.

Global Views

You can add a pointer node to the object navigator that causes a global view extension to appear in the vSphere Web Client main workspace. Global views are generally consoles, dashboards, applications. A best practice is to create pointer node extensions on the object navigator home page only for major applications and solutions. You can create pointers to other global views and dashboards on the vSphere Web Client Home screen. See Chapter 9, "Creating Home Screen Shortcuts," on page 87.

Object Collections

The object navigator vCenter page organizes vSphere objects using object collection nodes called inventory lists. Each inventory list node is an aggregate collection of all objects of a particular type in the vSphere environment. The Virtual Machine inventory list, for example, is an object collection node that contains all Virtual Machines in the vSphere environment.

Entity collection nodes are expandable lists. When the user clicks an object collection node, a list of every object in the collection appears in the object navigator control. The user can focus on any specific object in the collection to view that object's workspace and data views.

You can create an extension to add a new object collection node to the object navigator. A best practice is to add your object collection node to the object navigator vCenter page, and to create a new category for the object collection node.

Object collection nodes rely on relation extensions to display information about that object type's relationship to other objects in the vSphere environment. See Chapter 8, "Creating a New Relation Between vSphere Objects," on page 83.

Defining an Object Navigator Extension

All object navigator extensions must reference a common extension point, named `vise.navigator.nodespecs`. object navigator extensions must specify this extension point in the extension definition, and provide a data object of type `com.vmware.ui.objectnavigator.model.ObjectNavigatorNodeSpec.`

An object navigator extension can add a category to the object navigator control, it can add a simple pointer node, or it can add an object collection node. You determine which type of node you add to the object navigator by which properties you include in the `ObjectNavigatorNodeSpec` data object, and how you set those properties. Some properties of the `ObjectNavigatorNodeSpec` data object are optional and not used for certain node types.

Specifying the Object Navigator Page and Category

When you add any type of extension to the object navigator, you specify where the extension appears by using the `<parentUid>` property in the extension definition. Your extension can appear beneath one of the predefined categories in the object navigator, or under a new category that you create with an extension.

Table 6-1 lists the pre-defined categories in the object navigator and the corresponding `id` values to which you must set the `<parentUid>` property.

Table 6-1. Object Navigator Categories and IDs

Category Type	Category ID
Object Navigator Home Page > Solution Nodes	vsphere.core.navigator.solutionsCategory
Object Navigator vCenter Page	vsphere.core.navigator.virtualInfrastructure
Object Navigator vCenter Page > Inventory Lists Category	vsphere.core.navigator.viInventoryLists
Object Navigator Administration Page	vsphere.core.navigator.administration

Adding a Category to the Object Navigator

Your extension can add a category, rather than a node, to the object navigator. Table 6-2, on page 61, lists the properties of the `ObjectNavigatorNodeSpec` data object that you must set to add a category to the object navigator.

Table 6-2. ObjectNavigatorNodeSpec Extension Object Properties for Category Extension

Property	Type	Description
<title>	string	A string that appears as the text label for the category in the object navigator control. The string can be hard coded, or it can be a dynamic resource string included in your plug-in module.
<parentUid>	string	A string value that determines where the extension appears in the object navigator control. The value of <parentUid> property must match the extension id attribute of the parent category. The extension appears in the category that you specify in the <parentUid> property. See "Specifying the Object Navigator Page and Category" on page 60.

Example 6-1 shows an example extension definition for an object navigator extension that adds a new category. In the example, the category is named "Admin Sample", and the extension category is added to the object navigator **Administration** page.

Example 6-1. Example Category Extension to Object Navigator

```
<extension id="mySolution.myPlugin.myAdminSampleCategory">
    <extendedPoint>vise.navigator.nodespecs</extendedPoint>
    <object>
        <title>Admin Sample</title>
        <parentUid>vsphere.core.navigator.administration</parentUid>
    </object>
</extension>
```

Adding a Pointer Node to the Object Navigator

Your extension can add a pointer node to the object navigator. A pointer node extension causes a specific application, object workspace, or global view to appear in the vSphere Web Client main workspace when the user clicks the pointer node.

Table 6-3 lists the properties of the `ObjectNavigatorNodeSpec` data object that you must set to add a pointer node to the object navigator.

Table 6-3. ObjectNavigatorNodeSpec Extension Object Properties for Pointer Node Extension

Property	Type	Description
<title>	string	A string that appears as the text label for the pointer node in the object navigator control. The string can be hard coded, or it can be a dynamic resource string included in your plug-in module.
<parentUid>	string	A string value that determines where the extension appears in the object navigator control. The value of <parentUid> property must match the extension id attribute of the parent category. The extension appears in the category that you specify in the <parentUid> property. See "Specifying the Object Navigator Page and Category" on page 60.
<navigationTargetUid>	string	A string value that specifies the global view, application, or object workspace that appears in the main workspace when the user clicks the pointer node. The value of the <navigationTargetUid> property must match the extension id attribute for the target view.
<icon>	resource	The icon that appears for the pointer in the object navigator control. The value of the <icon> property typically references a dynamic resource in your plug-in module.

Example 6-2, on page 62, shows an example extension definition for an object navigator extension that adds a new pointer node. In the example, the pointer node has the name label "Sample Dashboard" and causes the extension `mySolution.myPlugin.myDashboardApp` to appear in the main workspace. The pointer node appears in the object navigator control in the top-level Solutions category.

Example 6-2. Example Pointer Extension to Object Navigator

```
<extension id="mySolution.myPlugin.myDashboardPointer">
    <extendedPoint>vise.navigator.nodespecs</extendedPoint>
    <object>
        <title>Sample Dashboard</title>
        <parentUid>vsphere.core.navigator.solutionsCategory</parentUid>
        <navigationTargetuid>mySolution.myPlugin.myDashboardApp</navigationTargetUid>
        <icon>#{samplePluginImages:sample}</icon>
    </object>
</extension>
```

Adding an Object Collection Node to the Object Navigator

Your extension can add a new object collection node to the object navigator. You typically add an object collection node to the object navigator if you have introduced a new type of object to the vSphere environment, and you want to provide direct access to the instances of that object.

An object collection node extension represents an aggregation of object instances, such as data centers or hosts. When the user clicks an object collection node, the node can expand to show a list of all instances of that object in the vSphere environment.

When you create an object collection node, part of your extension definition must reference a relation extension. The vSphere Web Client uses data from the relation extension to populate the expandable list of object instances. See Chapter 8, "Creating a New Relation Between vSphere Objects," on page 83.

Table 6-4 lists the properties of the `ObjectNavigatorNodeSpec` data object that you must set to add an object collection node to the object navigator.

Table 6-4. ObjectNavigatorNodeSpec Object Properties for Object Collection Node Extension

Property	Type	Description
`<title>`	string	A string that appears as the text label for the pointer node in the object navigator control. The string can be hard coded, or it can be a dynamic resource string included in your plug-in module.
`<parentUid>`	string	A string value that determines where the extension appears in the object navigator control. The value of `<parentUid>` property must match the extension `id` attribute of the parent category. The extension appears inside the category that you specify in the `<parentUid>` property. See "Specifying the Object Navigator Page and Category" on page 60. Typically, most object collection nodes are found in the Inventory Lists category of the object navigator vCenter page. A best practice is to either add your object collection node to the Inventory Lists category, or to create a new category extension.
`<navigationTargetUid>`	string	A string value that specifies the global view, application, or object workspace that appears in the main workspace when the user clicks the pointer node. The value of the `<navigationTargetUid>` property must match the extension `id` attribute for the target view. For an object collection node, a best practice is to set this value to an object collection data view. You can create an object collection data view by using the standard template when you create an object workspace for your custom object type. See Chapter 4, "Adding to vCenter Object Workspaces," on page 37.
`<icon>`	resource	The icon that appears for the pointer in the object navigator control. The value of the `<icon>` property typically references a dynamic resource in your plug-in module.

Table 6-4. ObjectNavigatorNodeSpec Object Properties for Object Collection Node Extension (Continued)

Property	Type	Description
<isFocusable>	boolean	A Boolean value. If you set this value to true, when the user clicks the collection node, the object navigator control will slide to display a new page with an expandable list of every object in the collection.
<nodeObjectType>	string	A string that indicates the type of object collection the node represents. The value you specify in the <nodeObjectType> property must match the extension id of an ObjectRelationSetSpec that you create in a Relation extension for your object.

Example 6-3 shows an example extension definition for an object navigator extension that adds an object collection node for a custom object type called a Rack. In the example, the Rack object collection node appears under the Inventory Lists category in the object navigator vCenter page.

The Rack object collection node extension references an object workspace collection data view with the identifier mySolution.myPlugin.Rack.objectCollectionView, and a relation extension with the identifier RackNodeCollection.

Example 6-3. Example Entity Collection Node Extension

```
<extension id="mySolution.myPlugin.myRackCollectionNode">
    <extendedPoint>vise.navigator.nodespecs</extendedPoint>
    <object>
        <title>Rack</title>
        <icon>#{myPluginImages:Rack}</icon>
        <parentUid>vsphere.core.navigator.viInventoryLists</parentUid>
        <navigationTargetUid>mySolution.myPlugin.Rack.objectCollectionView</navigationTargetUid>
        <isFocusable>true</isFocusable>
        <nodeObjectType>RackNodeCollection</nodeObjectType>
    </object>
</extension>
```

Using a Template to Define an Object Collection Node

You can use an XML template included in the vSphere Web Client SDK to quickly create an object collection node extension for a new object type. The object collection template generates an extension definition for an object collection node that contains an expandable sliding view in the object navigator for all objects in the collection. The variables in the template reference the necessary relation and object workspace extensions that the object collection node needs to function.

Example 6-4 shows an example of how to instantiate the object collection template for a new object type called a Chassis.

Example 6-4. Example Usage of Object Collection Template

```
<templateInstance id="com.vmware.samples.lists.allChassis">
       <templateId>vsphere.core.inventorylist.objectCollectionTemplate</templateId>
       <variable name="namespace" value="com.vmware.samples.chassisCollection"/>
       <variable name="title" value="#{chassisLabel}"/>
       <variable name="icon" value="#{chassis}"/>
       <variable name="objectType" value="samples:Chassis"/>
       <variable name="listViewId" value="com.vmware.samples.chassis.list"/>
       <variable name="parentUid" value="com.vmware.samples.chassisAppCategory"/>
</templateInstance>
```

In the example, the <variable> elements correspond to the properties of the ObjectNavigatorNodeSpec data object. To avoid name clashes with other extensions, such as object view templates, the namespace variable must be unique.

The template instance ID must be unique for every object collection you define.

Adding Custom Icons and Labels to an Object Collection Node

You can customize any object collection node that you create, by adding a new icon and label. To add an icon or label representation to an object collection node, you must create a separate extension called an object representation extension. An object representation extension can specify one or more icons and labels to display for a given object type, and the conditions under which each icon and label are displayed. For example, you can specify three different icons for a single object type, and have each icon appear under different conditions.

Defining an Object Representation Extension

Object representation extensions must reference a common extension point, named `vise.inventory.representationspecs`. Object representation extensions must specify this extension point in the extension definition, and provide a data object of type `com.vmware.ui.objectrepresentation.model.ObjectRepresentationSpec.`

An object representation extension specifies the target object type and an array of objects of type `com.vmware.ui.objectrepresentation.model.IconLabelSpec`. Each `IconLabelSpec` object in the array represents one icon and label set for the object, and can define the conditions for when that icon and label set appears.

Example 6-5 shows an example extension definition for an object representation extension. In the example, the extension adds one new icon and label set for a custom object called a Rack.

Example 6-5. Object Representation Extension for Rack Object

```
<extension id="com.vmware.samples.rack.iconLabelSpecCollection">
      <extendedPoint>vise.inventory.representationspecs</extendedPoint>
      <object>
         <objectType>samples:Rack</objectType>
         <specCollection>
            <com.vmware.ui.objectrepresentation.model.IconLabelSpec>
               <iconId>#{rack-empty}</iconId>
               <labelId>#{empty}</labelId>
               <conditionalProperties>
                  <String>!chassis</String>
               </conditionalProperties>
            </com.vmware.ui.objectrepresentation.model.IconLabelSpec>
         </specCollection>
      </object>
</extension>
```

Table 6-5 lists the properties of the `ObjectRepresentationSpec` data object that you must set.

Table 6-5. ObjectRepresentationSpec Extension Object Properties

Property	Type	Description
`<objectType>`	string	The object type for which the icon and label sets apply. In Example 6-5, the object type is `samples:Rack`.
`<specCollection>`	array	An array of objects of type `com.vmware.ui.objectrepresentation.model.IconLabelSpec`. Each `IconLabelSpec` object defines a single icon and label pair, along with the conditions under which that icon and label set appear.

Defining an Individual Icon and Label Set

Each `IconLabelSpec` object contains an icon ID, a label ID, and one or more conditional properties. The icon ID and label ID values specify the icon and label resources from your plug-in resource .SWF file.

Conditional properties govern when the icon and label appear in the object navigator. The `<conditionalProperties>` property of the `IconLabelSpec` object contains an array of properties to be evaluated as simple Boolean properties. You can use the negation operator (!) to evaluate against the reverse of the boolean value.

In Example 6-5, the property `chassis` is evaluated as a simple Boolean, and the (!) operator is used. The icon and label set in the example is only displayed for Rack objects without an associated Chassis.

You can also use a more advanced `<conditions>` property in an `IconLabelSpec` object to describe more complex display conditions for your icon and label set. The `<conditions>` property functions identically to property query constraints in the vSphere Web Client Data Service. See "Handling Constraints" on page 98

Creating Action Extensions 7

In the vSphere Web Client, actions represent commands that the user can issue to manage, administer, or otherwise manipulate the objects in the vSphere environment. Each action in the vSphere Web Client is associated with one or more specific vSphere object types. For example, the user might perform an action to change the power state of a selected Virtual Machine object, or to cause a Host object to enter or exit maintenance mode.

You can extend the vSphere Web Client by adding new actions. You can add actions to existing vSphere objects, or create actions associated with a new type of vSphere object.

When you add an action extension to the vSphere Web Client user interface layer, you must also extend the vSphere Web Client service layer with a new Java service. The Java service is responsible for performing the action operation on the target vSphere object.

This chapter contains the following topics:

Use Cases for Adding an Action Extension

You can extend the vSphere Web Client by adding actions associated with an existing type of vSphere object, or with a new type of vSphere object. You might add actions to an existing object type if you have created a custom version of that vSphere object, such as a custom host.

In addition to creating the action extension in the user interface layer, you must also add a Java service to the vSphere Web Client service layer. This Java service is used to perform the action operation on the target vSphere object.

About the Actions Framework and Action Sets

The Actions Framework governs all available actions in the vSphere Web Client. Each action in the Actions Framework is associated with one or more specific types of objects in the vSphere environment. Actions associated with virtual machines, for example, are available only when the user has selected a virtual machine object. Available actions are displayed in the actions drop-down menu at the top of the main workspace, or in a context menu when the user right-clicks on an object in the object navigator.

All actions in the Actions Framework are organized into groups called action sets. When you create action extensions to the vSphere Web Client, you must define one or more new action sets in the Actions Framework.

Action Controllers in Flex Extensions

In the plug-in module that contains your action extension, you must create a Flex command class. The Flex command class contains a handler method for each action. When the user invokes your action, the vSphere Web Client generates a Frinje event, which is then routed to the appropriate handler method in your command class.

Action Controllers for HTML Extensions

In the plug-in module that contains your action extension, you must create a Java actions controller. The controller runs on the Virgo application server and acts as a dispatcher for commands. The HTML UI component sends commands to the actions controller using a REST API, and the controller routes the commands to services that implement the actions.

Defining an Action Set

An extension that adds one or more actions to the vSphere Web Client must define an action set. You add each action set extension to a specific extension point in the vSphere Web Client user interface layer, named `vise.actions.sets`.

Your extension definition must define an action set and the individual actions within that action set. An action set is a data object of type `com.vmware.actionsfw.ActionSetSpec`. The `ActionSetSpec` object contains an `<actions>` property, which is an array of action data objects. You specify each individual action in the set inside the `<actions>` property, using a separate data object for each.

You associate each action set extension with a particular type of vSphere object. A best practice is to use the vSphere Web Client extension filtering mechanism to ensure that the actions are only visible when the user selects the relevant type of vSphere object. See "Filtering Extensions" on page 28.

IMPORTANT If you omit the `<metadata>` element for extension filtering in your action set extension definition, your action is shown for all vSphere objects. Use the `<metadata>` element to ensure that your actions appear only for the correct type of vSphere or custom objects.

Defining Individual Actions in an Action Set

Each action data object in a set is a data object of type `com.vmware.actionsfw.ActionSpec`. You must create an `ActionSpec` data object for every action to add to the action set.

You create each `ActionSpec` data object using a `<com.vmware.actionsfw.ActionSpec>` XML element. In this element, you set the properties for the action. Table 7-1 lists the properties that you set for each `ActionSpec` object.

Table 7-1. ActionSpec Object Properties

Property	Type	Description
`<uid>`	string	Unique identifier of the action. Using namespace style is a best practice, for example com.mySolution.myPlugin.chassis.chassisPowerAction.
`<label>`	string	String containing the text label for the action, suitable for a button or context menu in the vSphere Web Client user interface.
`<description>`	string	String containing a short description of the action, suitable for a tooltip.
`<icon>`	resource	Icon class or dynamic resource for the action. This property is optional.
`<acceptsMultipleTargets>`	boolean	Boolean value. If you set this value to `true`, the action can be made available when the user selects multiple vSphere objects. This property is optional.

Table 7-1. ActionSpec Object Properties

Property	Type	Description
<conditionalProperty>	string	Name of a Boolean property on the target vSphere or custom object. You can use the value of the property specified to define whether the action is available on the target object. For example, you can specify a Boolean property on a Virtual Machine object, and make your action available only when that property is true. This property is optional.
<command>	string	Command class for a Flex-based action. The command class contains a method annotated to handle the event that the vSphere Web Client generates when the user invokes your action. The handler method must then perform the actual action operation on the vSphere environment. See "Using Flex Command Classes To Handle Actions" on page 72.

HTML-based extensions do not use the <command> property. Instead they contain a <delegate> object.

Defining the <delegate> Object in HTML-Based Action Extensions

HTML-based extensions use delegated actions instead of the command classes used by Flex-based extensions. The <delegate> object requires a <className> property and an <object> element that contains only an embedded <root> object. Example 7-2, "Example Action Set for an HTML-Based Extension," on page 70 shows the structure of the <delegate> object for an HTML-based extension.

The <className> property must specify the HtmlActionDelegate class for HTML-based extensions:

```
<className>com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate</className>
```

The <root> object specifies the UI dialog box used to initiate the HTML-based action.

Table 7-2 lists the properties that you set in the <root> object.

Table 7-2. Root Object Properties for HTML Action Extension

Property	Type	Description
<actionURL>	string	Identifies the HTML resource to be displayed. Can be an absolute HTTPS URL or a bundle context path. If it is a bundle context path, a relative URL, it must end with the .html extension to enable session authentication. For absolute URLs, the framework does not use session authentication.
<dialogTitle>	string	Title string for dialog box. Must be present, or the action is treated as headless. Can be localized.
<dialogSize>	string	Width and height of the dialog box, in pixels, separated by comma.

Example Action Extension Definitions

You define the client part of your action extension using ActionSpec objects in your plugin.xml file. The properties you use to define an ActionSpec object depend on the type of extension.

Flex-based action extensions specify a command class, implemented in ActionScript. HTML-based extensions specify instead a delegate class, provided by the framework.

There are two types of HTML-based action extensions. One type, known as a *UI* action, displays a modal dialog box for user input or confirmation before submitting a service request. The other type, known as a *headless* action, initiates a request to a service without additional user input. An extension definition for a UI action specifies the size and title of the dialog box, while a headless action definition omits the dialog box properties.

Flex-Based Action Extension Definition

Example 7-1 shows an example extension definition for a Flex-based action set extension. In the example, the extension adds a set of two actions to the vSphere Web Client Actions Framework. The actions are associated with a custom object type called a Chassis.

Example 7-1. Example Action Set for a Flex-Based Extension

```
<extension id="mySolution.myPlugin.myActionSet">
    <extendedPoint>vise.actions.sets</extendedPoint>
    <object>
        <actions>
            <!-- first action -->
            <com.vmware.actionsfw.ActionSpec>
                <uid>com.mySolution.myPlugin.chassis.createChassis</uid>
                <label>Create Chassis</label>
                <description>Create a Chassis object.</description>
                <icon>#{myPluginImages:sample1}</icon>
                <acceptsMultipleTargets>false</acceptsMultipleTargets>
                <command className="com.mySolution.myPlugin.ChassisCommand"/>
            </com.vmware.actionsfw.ActionSpec>
            <!-- second action -->
            <com.vmware.actionsfw.ActionSpec>
                <uid>com.mySolution.myPlugin.chassis.editChassis</uid>
                <label>Edit Chassis</label>
                <description>Edit a Chassis object.</description>
                <icon>#{myPluginImages:sample2}</icon>
                <acceptsMultipleTargets>true</acceptsMultipleTargets>
                <conditionalProperty>actions.isEditAvailable</conditionalProperty>
                <command className="com.mySolution.myPlugin.ChassisCommand"/>
            </com.vmware.actionsfw.ActionSpec>
        </actions>
    </object>
    <metadata>
        <!-- filters the actions in the set to be visible only for Chassis -->
        <objectType>samples:Chassis</objectType>
    </metadata>
</extension>
```

HTML-Based UI Action Extension Definition

Example 7-2 shows an example extension definition for an HTML-based UI action extension. The extension must use a `<delegate>` object instead of the `<command>` object used by a Flex-based extension. The action in this definition is associated with a custom object type called a Chassis.

Example 7-2. Example Action Set for an HTML-Based Extension

```
<extension id="com.vmware.samples.chassis.listActionSet">
      <extendedPoint>vise.actions.sets</extendedPoint>
      <object>
         <actions>
            <com.vmware.actionsfw.ActionSpec>
               <uid>com.vmware.samples.chassis.createChassis</uid>
               <label>#{chassis.createAction}</label>
               <icon>#{addChassis}</icon>
               <delegate>
                  <className>com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate</className>
                  <object><root>

               <actionUrl>/vsphere-client/chassis/resources/createChassisDialog.html</actionUrl>
                     <dialogTitle>#{chassis.createAction}</dialogTitle>
                     <dialogSize>500,400</dialogSize>
                  </root></object>
               </delegate>
               <privateAction>true</privateAction>
            </com.vmware.actionsfw.ActionSpec>
         </actions>
      </object>
   </extension>
```

When the action is invoked the platform opens a modal dialog containing the HTML document specified in the `actionUrl` property. The following parameters are added to the URL:

- `actionUid`: the `<uid>` of the ActionSpec object defined in the `plugin.xml` file
- `targets`: a comma-separated list of objectIds
- locale: the current locale used

By default, the targets parameter takes only one `objectId`. To specify more than one `objectId`, set the flag `acceptsMultipleTargets` to `true`.

In this example, the full URL takes the following form:

`/vsphere-client/chassis/resources/createChassisDialog.html?actionUid=com.vmware.samples.chassis.createChassis&targets=`*`objectId`*`&locale=en`

The dialog script uses the REST API to GET or POST data requests. For instance, to get object properties using the Data Access API, you use a request similar to the following:

`/vsphere-client/htmltest/rest/data/properties/`*`objectId`*`?properties=`*`properties-list`*

After the dialog form is submitted or the operation is canceled, the Javascript code calls `WEB_PLATFORM.closeDialog()`.

HTML-Based Headless Action Extension Definition

Example 7-3 shows an example extension definition for an HTML-based headless action extension. The extension must use a `<delegate>` object instead of the `<command>` object used by a Flex-based extension. The action in this definition is associated with a custom object type called a Chassis.

Example 7-3. Example Action Set for a Headless HTML-Based Extension

```
<extendedPoint>vise.actions.sets</extendedPoint>
      <object>
         <actions>
            <com.vmware.actionsfw.ActionSpec>
               <uid>com.vmware.samples.chassis.deleteChassis</uid>
               <label>#{chassis.deleteAction}</label>
               <icon>#{deleteChassis}</icon>
               <delegate>
                  <className>com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate</className>
                  <object><root>
                     <actionUrl>/vsphere-client/chassis/rest/actions.html</actionUrl>
                  </root></object>
               </delegate>
            </com.vmware.actionsfw.ActionSpec>
      ...
```

When the headless action is invoked the HTML bridge makes a POST request to the actions controller on the Virgo server, using the `actionUrl` property. The following parameters are added to the URL:

- `actionUid`: the `<uid>` of the ActionSpec object defined in the `plugin.xml` file
- `targets`: a comma-separated list of objectIds

By default, the targets parameter takes only one `objectId`. To specify more than one `objectId`, set the flag `acceptsMultipleTargets` to `true`.

In this example, the full URL takes the following form:

`/vsphere-client/chassis/rest/actions.html?actionUid=com.vmware.samples.chassis.deleteChassis&targets=`*`objectId`*

Using `callActionsController` To Invoke HTML Actions

Your HTML-based action extension can invoke headless actions on its own initiative, by calling the Javascript method `WEB_PLATFORM.callActionsController(url, jsonData)`.

- The value of `url` has the form /vsphere-client/chassis/rest/actions.html?actionUid=*actionUid*
- The value of `jsonData` is a JSON map of parameters passed to the actions controller, or `null` if no parameters are needed.

Using Flex Command Classes To Handle Actions

In a Flex-based action extension, you use a command class to provide the code that performs the action operation in the vSphere environment when the user clicks your action.

When the user starts your action, the vSphere Web Client generates an `ActionInvocationEvent` using the ID you specified using the `<uid>` property in the `ActionSpec` object in your extension definition. The vSphere Web Client then instantiates the command class you specified in the `<command>` property. Your command class must contain a method annotated to handle the `ActionInvocationEvent` that is generated when the action is started. A single command class can contain handler methods for multiple actions, if those methods have the necessary annotations.

You annotate a method to handle an `ActionInvocationEvent` by using the `[RequestHandler]` metadata tag when you declare the method. In the `[RequestHandler]` tag, you must specify the `<uid>` property for the action to handle. To annotate a method to handle the "Create Chassis" command in Example 7-1, you must include the following `[RequestHandler]` tag in your method declaration.

```
[RequestHandler("com.mySolution.myPlugin.createChassis")]
```

Your method must also accept the `ActionInvocationEvent` as a parameter.

Example 7-4, on page 73, shows an example Flex command class. In the example, the `ChassisCommand` class contains a handler method for each of the actions defined in Example 7-1.

Example 7-4. Example Flex Command Class

```
package com.vmware.samples.chassisui {

import com.vmware.actionsfw.ActionContext;
import com.vmware.actionsfw.events.ActionInvocationEvent;
import com.vmware.core.model.IResourceReference;
import com.vmware.data.common.ObjectChangeInfo;
import com.vmware.data.common.OperationType;
import flash.events.EventDispatcher;
import mx.controls.Alert;

[Event(name="{com.vmware.data.common.events.ModelChangeEvent.MODEL_CHANGE}",
    type="com.vmware.data.common.events.ModelChangeEvent")]

public class ChassisCommand extends EventDispatcher {
    private var _proxy:ChassisServiceProxy = new ChassisServiceProxy();

    /** Create Chassis action handler **/
    [RequestHandler("com.mySolution.myPlugin.chassis.createChassis")]
    public function onCreateChassisActionInvocation(event:ActionInvocationEvent):void {
        _proxy.createChassis(onCreateChassisComplete);
    }

    /** Edit Chassis action handler **/
    [RequestHandler("com.mySolution.myPlugin.chassis.editChassis")]
    public function onEditChassisActionInvocation(event:ActionInvocationEvent):void {
        var chassisReference:IResourceReference = getIResourceReference(event);
        _proxy.editChassis(chassisReference, onEditChassisComplete, chassisReference);
    }

    private function onEditChassisComplete(event:MethodReturnEvent):void {
        if (event.error != null) {
            Alert.show(event.error.message);
            return;
        }
        var chassisReference:IResourceReference = event.callContext as IResourceReference;
        var mce:ModelChangeEvent = ModelChangeEvent.newSingleObjectChangeEvent(
            chassisReference,
            OperationType.CHANGE);
        dispatchEvent(mce);
    }

    private function onCreateChassisComplete(event:MethodReturnEvent):void {
        if (event.error != null) {
            Alert.show(event.error.message);
            return;
        }
        var chassisReference:IResourceReference = event.result as IResourceReference;
        var mce:ModelChangeEvent = ModelChangeEvent.newSingleObjectChangeEvent(
            chassisReference,
            OperationType.ADD);
        dispatchEvent(mce);
    }

    private function getIResourceReference(event:ActionInvocationEvent):IResourceReference {
        // actionContext.targetObjects is an array of objects on which the action is called
        // so actionContext.targetObjects[0] is the selected Chassis for this action.
        var actionContext:ActionContext = event.context;
        if (actionContext == null || (actionContext.targetObjects.length <= 0)
                || (!(actionContext.targetObjects[0] is IResourceReference))) {
            return null;
        }
        return (actionContext.targetObjects[0] as IResourceReference);
    }
}
}
```

In the example, the `onCreateChassisActionInvocation` and `onEditChassisActionInvocation` methods are annotated to handle the actions defined in the action set in Example 7-1. The actual action operations are performed by an imported Java service. In Example 7-4, `ChassisServiceProxy` refers to a proxy class for a Java service that performs the action operations. See "Performing Action Operations on the vSphere Environment" on page 75.

Java Actions Controller Classes for HTML Extensions

When you create an HTML-based action extension to the vSphere Web Client, you must create an actions controller class on the Virgo server to respond to REST API requests from the client code. A best practice is to implement the controller class as a simple dispatcher that maps the action UIDs to Java services. You can invoke custom services or translate REST API requests to Data Manager requests.

Example 7-5. Example Java Actions Controller Class

```
...
   @RequestMapping(method = RequestMethod.POST)
   @ResponseBody
   public Map invoke(
            @RequestParam(value = "actionUid", required = true) String actionUid,
            @RequestParam(value = "targets", required = false) String targets,
            @RequestParam(value = "json", required = false) String json)
            throws Exception {
      ...

      ActionResult actionResult = new ActionResult(actionUid, RESOURCE_BUNDLE);

      if (actionUid.equals("com.vmware.samples.chassis.editChassis")) {
         boolean result = _chassisService.updateChassis(objectRef, chassisInfo);
         actionResult.setObjectChangedResult(result, "editAction.notFound");

      } else if (actionUid.equals("com.vmware.samples.chassis.deleteChassis")) {
         boolean result = _chassisService.deleteChassis(objectRef);
         actionResult.setObjectDeletedResult(result, "deleteAction.notFound");

      } else if (actionUid.equals("com.vmware.samples.chassis.createChassis")) {
         Object result = _chassisService.createChassis(chassisInfo);
         if (result != null) {
            actionResult.setObjectAddedResult((URI)result, "samples:Chassis", null);
         } else {
            // Case where the name is already taken
            String[] params = new String[] { chassisInfo.name };
            actionResult.setErrorMessage("createAction.nameTaken", params);
         }

      } else {
         String warning = "Action not implemented yet! "+ actionUid;
         _logger.warn(warning);
         actionResult.setErrorLocalizedMessage(warning);
      }
      return actionResult.getJsonMap();
   }
...
```

In the example, the `invoke` method must return a JSON map that signals the framework to update its object lists in the client. The map must contain a `result` key whose value is the result of the service call. A utility class `ActionResult` is provided to define additional result parameters.

If the action adds, updates, or deletes objects, you should use an instance of `ActionResult` to notify the Web Client framework, so it can update object lists in the client. Use the `ActionResult` methods `setObjectAddedResult`, `setObjectChangedResult`, or `setObjectDeletedResult`. Pass the result returned by the service as the first parameter to the method.

The `ActionResult` methods accept an optional second parameter that contains a message key that the client code can use to report an error in the UI if the result is false or null. Alternatively, you can use the `setErrorMessage` method to add the error message key to the results map.

NOTE Headless actions cannot display error messages or otherwise report the results of the action. To display results in the UI, you must handle your action request from a UI action dialog.

Performing Action Operations on the vSphere Environment

Typically, an action requires you to make a change to the vSphere environment, such as changing the power state of a Virtual Machine, creating a new Host object, or deleting a vSphere object. You make changes to the vSphere environment using a Java service in the vSphere Web Client service layer. A command class for an action extension must either import the Java service, or use a service proxy class. Methods in the Java service perform the actual operations in the vSphere environment.

A best practice is to create a new Java service for your action extensions, and add that service to the vSphere Web Client service layer. You can then create a proxy class for the Java service, and use the service proxy in a command class. See Chapter 11, "vSphere Web Client Service Layer," on page 93.

Obtaining the Action Target Object

For some actions, such as the Edit Chassis action in Example 7-4, you must obtain a reference to the target vSphere object that the user selected before performing the action. You can obtain a reference to the target vSphere object by using the `ActionInvocationEvent` that the Actions Framework generates when an action is triggered.

The `ActionInvocationEvent` contains an object of type `ActionContext` in the Actions Framework library. The `ActionContext` object contains an array named `targetObjects`, from which you can obtain references to all vSphere objects that the user has selected in the user interface. In Example 7-4, the method `getIResourceReference` shows an example of how to obtain a reference to the target object.

Your Flex command class must import the packages `com.vmware.actionsfw.ActionContext` and `com.vmware.actionsfw.events.ActionInvocationEvent` to use these vSphere Web Client SDK features.

Updating the Client with Action Operation Results

When your Java service completes the action operation, you must update the vSphere Web Client with the results of that operation. You update the vSphere Web Client by dispatching a `ModelChangeEvent` in your command class.

To dispatch a `ModelChangeEvent`, your Flex command class must import the packages `com.vmware.data.common.OperationType`, `com.vmware.data.common.ObjectChangeInfo`, and `com.vmware.data.common.events.ModelChangeEvent`. You must also annotate your command class as follows.

```
[Event(name="{com.vmware.data.common.events.ModelChangeEvent.MODEL_CHANGE}",
    type="com.vmware.data.common.events.ModelChangeEvent")]
```

You can use a callback method to ensure that your command class dispatches the necessary `ModelChangeEvent`. Your service proxy class can invoke the callback method in your command class when the action operation finishes. In Example 7-4, the command class passes references to the appropriate callback methods, `onEditChassisComplete` and `onCreateChassisComplete`, when it invokes the action operations in the service proxy. The callback methods then process the results of the action operation and dispatch the `ModelChangeEvent` as necessary.

Organizing Your Actions in the User Interface

You can control some aspects of how the actions you add appear in the vSphere Web Client user interface. In addition to choosing the objects for which your actions appear, you can display actions in nested menus, add nested menus and separators to action menus, and prioritize which actions appear highest in an action menu.

Extending an Action Menu

You can extend an object's action menu by adding a nested solution menu. The solution menu can contain actions, additional nested menus, and separators. You can extend an action menu by defining a Solution Menu extension in your plug-in module's `plugin.xml` manifest file.

Defining a Solution Menu Extension

All Solution Menu extensions that you create use a common extension point in the vSphere Web Client extension framework, called `vsphere.core.menus.solutionMenus`. The extension definition must provide an extension object of type `com.vmware.actionsfw.ActionMenuItemSpec`.

The `ActionMenuItemSpec` object represents the new solution menu nested in an object's action menu. In the solution menu object, you define an array of additional `ActionMenuItemSpec` objects to represent actions, nested menus, and separators.

Table 7-3 lists the properties of the `ActionMenuItemSpec` object that you must set for your nested solution menu extension.

Table 7-3. ActionMenuItemSpec Extension Object Properties for a Nested Solution Menu

Property	Type	Description
`<label>`	string	String that appears as the text label for the nested solution menu. The string can be hard coded, or it can be a dynamic resource string included in your plug-in module.
`<uid>`	string	The unique identifier for the nested solution menu. A best practice is to use the English label for the nested solution menu, in camel case.
`<children>`	array	An array of additional `ActionMenuItemSpec` objects. Each `ActionMenuItemSpec` child object can represent an action, a nested menu, or a separator.

You associate your Solution Menu extension with a particular type of vSphere or custom object. A best practice is to use the vSphere Web Client extension filtering mechanism to ensure that the actions are only visible when the user selects the relevant type of vSphere object. See "Filtering Extensions" on page 28.

IMPORTANT If you omit the `<metadata>` element for extension filtering in your Solution Menu extension definition, your Solution Menu is shown for all vSphere objects. Use the `<metadata>` element in your Solution Menu extension definition to ensure that your Solution Menu appears only for the correct type of vSphere or custom objects.

Example 7-6, on page 77, shows an example Solution Menu extension definition. In the example, the extension adds a nested solution menu to the action menu for VirtualMachine objects.

Example 7-6. Example Solution Menu Extension Definition

```
<!-- Defines a solution sub-menu on VirtualMachine objects -->
<extension id="com.vmware.samples.actions.submenus">
    <extendedPoint>vsphere.core.menus.solutionMenus</extendedPoint>
    <object>
        <label>#{allSampleActions.label}</label>
        <uid>allSampleVMActions</uid>
        <children>
            <Array>
            <!-- array of ActionMenuItemSpec objects, which are items in the Solution Menu -->
                <com.vmware.actionsfw.ActionMenuItemSpec>
                ...
                </com.vmware.actionsfw.ActionMenuItemSpec>
                <com.vmware.actionsfw.ActionMenuItemSpec>
                ...
                </com.vmware.actionsfw.ActionMenuItemSpec>
            </Array>
        </children>
    </object>
    <metadata>
        <objectType>VirtualMachine</objectType>
    </metadata>
</extension>
```

Adding Items to a Nested Solution Menu

You can add items to a nested solution menu by using the `<children>` property in your Solution Menu extension's `ActionMenuItemSpec` object. In the `<children>` property, you can add one or more additional `ActionMenuItemSpec` objects to represent each item.

Adding an Action

To add an action to your solution sub-menu, you create a child `ActionMenuItemSpec` object with the properties listed in Table 7-4.

Table 7-4. ActionMenuItemSpec Child Object Properties for an Action

Property	Type	Description
`<label>`	string	String that appears as the text label for the action. The string can be hard coded, or it can be a dynamic resource string included in your plug-in module.
`<uid>`	string	Unique identifier for the action. The action UID must match the UID you set for the action when you defined it using the `ActionSpec` object. See "Defining Individual Actions in an Action Set" on page 68.
`<type>`	string	Type parameter for the item to add to the nested solution menu. For an action, `<type>` must be set to the string `action`.

Example 7-7, on page 78, shows an example of a nested solution menu with two action items as child objects.

Example 7-7. Nested Solution Menu Extension with Actions

```
<!-- Defines a solution sub-menu on VirtualMachine objects -->
<extension id="com.vmware.samples.actions.submenus">
    <extendedPoint>vsphere.core.menus.solutionMenus</extendedPoint>
    <object>
        <label>#{allSampleActions.label}</label>
        <uid>allSampleVMActions</uid>
        <children>
            <Array>
            <!-- array of ActionMenuItemSpec objects, which are items in the Solution Menu -->
                <com.vmware.actionsfw.ActionMenuItemSpec>
                <!-- first action -->
                    <type>action</type>
                    <uid>com.vmware.samples.actions.myVmAction1</uid>
                    <label>#{action1.label}</label>
                </com.vmware.actionsfw.ActionMenuItemSpec>
                <com.vmware.actionsfw.ActionMenuItemSpec>
                <!-- second action -->
                    <type>action</type>
                    <uid>com.vmware.samples.actions.myVmAction1</uid>
                    <label>#{action1.label}</label>
                </com.vmware.actionsfw.ActionMenuItemSpec>
            </Array>
        </children>
    </object>
    <metadata>
        <objectType>VirtualMachine</objectType>
    </metadata>
</extension>
```

Adding a Separator

To add a separator to your nested solution menu, you create a child `ActionMenuItemSpec` object with the properties listed in Table 7-5.

Table 7-5. ActionMenuItemSpec Child Object Properties for a Separator

Property	Type	Description
`<type>`	string	Type parameter for the item to add to the nested solution menu. For a separator, `<type>` must be set to the string `separator`.

Adding a Nested Menu

To add an action to your nested solution menu, you create a child `ActionMenuItemSpec` object with the properties listed in Table 7-6. For menus nested to another level, you must omit the `<type>` property and include a `<children>` property.

Table 7-6. ActionMenuItemSpec Child Object Properties for a Nested Sub-Menu

Property	Type	Description
`<label>`	string	String that appears as the text label for the nested menu. The string can be hard coded, or it can be a dynamic resource string included in your plug-in module.
`<uid>`	string	Unique identifier for the nested menu. A best practice is to use the English label for the menu in camel case.
`<children>`	array	Array of additional `ActionMenuItemSpec` objects. Each `ActionMenuItemSpec` child object can represent an action, a nested menu, or a separator.

Example 7-8, on page 79, shows an example solution menu that adds a Solution Menu extension for VirtualMachine objects. The nested solution menu includes a subordinate nested menu called “Configuration” with two actions, a separator, and an additional action.

Example 7-8. Example Solution Menu Extension with Nested Menu and Separator

```
<!-- Defines a solution sub-menu on VirtualMachine objects -->
<extension id="com.vmware.samples.actions.submenus">
    <extendedPoint>vsphere.core.menus.solutionMenus</extendedPoint>
    <object>
        <label>#{allSampleActions.label}</label>
        <uid>allSampleVMActions</uid>
        <children>
            <Array>
            <!-- array of ActionMenuItemSpec objects, which are items in the Solution Menu -->

                <!-- add a nested sub-menu called "configuration" -->
                <com.vmware.actionsfw.ActionMenuItemSpec>
                    <uid>configuration</uid>
                    <label>#{configurationMenu.label}
                    <children>
                    <array>
                        <!-- first action in sub-menu -->
                        <com.vmware.actionsfw.ActionMenuItemSpec>
                            <type>action</type>
                            <uid>com.vmware.samples.actions.myVmAction1</uid>
                            <label>#{action1.label}</label>
                        </com.vmware.actionsfw.ActionMenuItemSpec>

                        <!-- second action in sub-menu -->
                        <com.vmware.actionsfw.ActionMenuItemSpec>
                            <!-- second action -->
                            <type>action</type>
                            <uid>com.vmware.samples.actions.myVmAction1</uid>
                            <label>#{action1.label}</label>
                        </com.vmware.actionsfw.ActionMenuItemSpec>
                    </array>
                    </children>
                </com.vmware.actionsfw.ActionMenuItemSpec>

                <!-- add a separator after the sub-menu -->
                <com.vmware.actionsfw.ActionMenuItemSpec>
                <type>separator</type>
                </com.vmware.actionsfw.ActionMenuItemSpec>

                <!-- add an additional action -->
                <com.vmware.actionsfw.ActionMenuItemSpec>
                    <type>action</type>
                    <uid>com.vmware.samples.actions.myVmAction3</uid>
                    <label>#{action3.label}</label>
                </com.vmware.actionsfw.ActionMenuItemSpec>
            </Array>
        </children>
    </object>
    <metadata>
        <objectType>VirtualMachine</objectType>
    </metadata>
</extension>
```

Prioritizing Actions in the User Interface

If your plug-in module adds actions to the vSphere Web Client, you can change the order in which those actions appear at different points in the user interface. You can prioritize the action order in the global action menu for all vSphere actions, or in the context-specific action menu for a vSphere or custom object.

To change your actions' display order, you must create a Prioritization extension. You must define the Prioritization extension in the same plug-in package that contains your action extensions. You must create different extensions depending on whether you want to promote actions in the action menu for a certain object type, or to set the action priority in the global action list.

Prioritizing Context-Specific Actions

To change how actions are prioritized in the action menu for a specific object type, you must create a Prioritization extension at the extension point `vmware.prioritization.actions`. Your extension must provide a data object of type `com.vmware.vsphere.client.prioritization.ActionPriorityGroup`.

The `ActionPriorityGroup` object defines the priority order for the actions in the action menu, and the target object type. The actions listed first in your extension definition appear at the top of the action menu and to the right side of the actions toolbar. You can promote a maximum of five actions on the actions toolbar.

Table 7-7 lists the properties of the `ActionPriorityGroup` object that you must set in your Prioritization extension.

Table 7-7. ActionPriorityGroup Extension Object Properties

Property	Type	Description
`<prioritizedIds>`	array	Array of `<string>` elements. Each `<string>` element contains the UID of an action. You list each action UID string in order from highest priority to lowest priority, depending on how you want the actions to appear in the action menu.
`<actionTargetTypes>`	array	`<string>` element that contains the target object type, such as samples:Chassis or VirtualMachine. The Prioritization extension sets the display priority for the target object type's action menu.
`<regionId>`	string	Optional parameter you can use to specify a specific subset of actions in an action menu. If a vSphere or custom object has a different action list depending on that object's state, you can specify the UID of that action list in the `<regionId>` parameter to promote actions in that action list.

Example 7-9 shows an example Prioritization action that sets the display priority for actions in the default actions menu for a custom object called a Chassis.

Example 7-9. Prioritizing Items in the Chassis Action Menu

```
<extension id="com.vmware.samples.chassis.actionMenuPrioritization.editAction">
    <extendedPoint>vmware.prioritization.actions</extendedPoint>
    <object>
        <prioritizedIds>
            <string>com.vmware.samples.chassis.editChassis</string>
            <string>com.vmware.samples.chassis.deleteChassis</string>
            <string>com.vmware.samples.chassis.moveChassis</string>
            <string>com.vmware.samples.chassis.poweronChassis</string>
            <string>com.vmware.samples.chassis.poweroffChassis</string>
        </prioritizedIds>
        <actionTargetTypes>
            <string>samples:Chassis</string>
        </actionTargetTypes>
    </object>
</extension>
```

Prioritizing Global Actions Toolbar

To change how actions are prioritized in the global actions menu, you must create a Prioritization extension at the extension point `vmware.prioritization.listActions`. Your extension must provide a data object of type `com.vmware.vsphere.client.prioritization.ActionPriorityGroup`.

The `ActionPriorityGroup` object defines the priority order for the actions in the global actions toolbar, which is the toolbar that appears to the left of the context toolbar. The actions listed first in your extension definition appear farthest left in the global actions toolbar.

Table 7-8, on page 81, lists the properties of the `ActionPriorityGroup` object that you must set in your Prioritization extension.

Table 7-8. ActionPriorityGroup Extension Object Properties

Property	Type	Description
<prioritizedIds>	array	Array of <string> elements. Each <string> element contains the UID of an action. You list each action UID string in order from highest priority to lowest priority, depending on how you want the actions to appear in the global action list.
<regionId>	string	UID of the region of the global action list to modify.

Example 7-10 shows an example Prioritization action that sets the display priority for actions in the default actions menu for a custom object called a Chassis.

Example 7-10. Prioritizing Items Added to Global Actions Toolbar for Chassis Objects

```
<extension id="com.vmware.samples.chassis.actionListPrioritization.createAction">
    <extendedPoint>vmware.prioritization.listActions</extendedPoint>
    <object>
        <prioritizedIds>
            <string>com.vmware.samples.chassis.createChassis</string>
            <string>com.vmware.samples.chassis.createRack</string>
        </prioritizedIds>
        <regionId>com.vmware.samples.chassis.list</regionId>
    </object>
</extension>
```

Creating a New Relation Between vSphere Objects

8

The vSphere objects that make up the virtual infrastructure are organized as a graph of related objects. Each type of vSphere object maintains parent-child relationships with adjacent objects. For example, a host object can have several related virtual machine objects as children, while many host objects might have a single cluster object as a parent. The relationships between vSphere objects are reflected in different places in the vSphere Web Client GUI, including the object navigator and on the **Related Objects** tab in an object workspace. The object navigator and object workspaces can display the related items for a vSphere object, such as the virtual machine objects associated with a given host object.

The vSphere Web Client uses relation extensions to represent the relationships between objects. A relation extension defines a relationship between a vSphere object and any other objects in the hierarchy of the virtual infrastructure. Certain extensions, such as some object navigator nodes and object list views, reference relation extensions to obtain their information. You can create relation extensions to add new relationships between vSphere objects in your vSphere environment, and to have other areas of the vSphere Web Client reflect those relationships.

This chapter contains the following topics:

Use Cases for Adding a Relation Extension

You create a relation extension when you want the user to receive information about other vSphere or custom objects, either in a list view or in the object navigator, when they select or focus on a given vSphere or custom object. For example, if you created a chassis object type, it might contain one or more host objects. You can create a relation extension for the chassis object so that information about any related host objects also appears in the object workspace or object navigator when the user selects a chassis object.

You can add a relation extension to the vSphere Web Client to create a new relationship between a given type vSphere object and any other object type in the virtual infrastructure hierarchy. If you add a new type of object to the vSphere environment, you must create a relation extension to place your new object in the graph of vSphere objects, and to populate the Related Items views.

You can also create a relation extension for two existing vSphere objects that do not have an existing relationship. You might create such a relation to take advantage of the Related Items views in the object navigator and object workspaces for those objects, or if you created an object collection node for an existing object in the object navigator.

Defining a Relation Extension

To create a relation extension, you need to create the metadata extension definition in your user interface plug-in module `plugin.xml` file. You do not need to create a Flex class. Relation extensions use a single common extension point in the vSphere Web Client user interface layer, named `vise.relateditems.specs`. Relation extensions must specify this extension point, and provide a data object of type `com.vmware.ui.relateditems.model.ObjectRelationSetSpec`.

The `ObjectRelationSetSpec` object specifies the object type to which the relation pertains, and a list of `RelationSpec` objects that describe the current object's relationship to other vSphere objects. The `ObjectReltaionSetSpec` contains other optional properties that you can use to set the data view in which relations are displayed, and to further refine when relations appear.

Table 8-1 lists the properties you set for the `ObjectRelationSetSpec` object.

Table 8-1. ObjectRelationSetSpec Extension Object Properties

Property	Type	Description
`<type>`	string	Type of object to which the relation pertains. For example, if you are creating a relation for the `Host` object type, you set the `<type>` property to the string `HostSystem`.
`<relationSpecs>`	array	Array of objects of type `com.vmware.ui.relateditems.model.RelationSpec`. Each `RelationSpec` object describes the relation between the current object, specified in the `<type>` property, and a target object that you specify. See "Describing the Relation Using the RelationSpec Object" on page 84.
`<relationsViewId>`	string	Extension ID of the Related Items data view for the current object. When the user selects the object node using the object navigator, the Related Items data view you specify in the `<relationsViewId>` property appears in the main workspace in the vSphere Web Client GUI. This property is optional. If you do not include the `<relationsViewId>` property in your extension definition, no Related Items data view appears for that object.
`<conditionalProperty>`	string	Optional property that you can include as an additional constraint for the relation. To use this property, you specify the name of a Boolean property on the target object. For example, if you are creating a relation extension for a virtual machine object, you can additionally specify a Boolean property, such as the power-on state, on that virtual machine using `<conditionalProperty>`. The relation appears in the vSphere Web Client only if both the relation conditions are met and `<conditionalProperty>` is satisfied.

Describing the Relation Using the RelationSpec Object

You describe each of the relations for the vSphere object using `RelationSpec` objects, which you must create inside the `<relationSpecs>` property of the `ObjectRelationSetSpec` extension object. Each `RelationSpec` object contains information on the relation target object type, display properties such as icons and labels, and any relevant list extensions for the relation target object.

Table 8-2 lists the properties you set for each `RelationSpec` object.

Table 8-2. RelationSpec Extension Object Properties

Property	Type	Description
<id>	string	Unique identifier for the relation. The best practice for creating an ID value for a relation is to use the format [relation target object]For[relation object]. For example, if the parent ObjectRelationSetSpec object defines the relations for Chassis objects, you might use the value hostForChassis as the ID for the RelationSpec object that describes the relation to host objects.
<icon>	array	Icon resource for the relation. This icon appears in the Related Objects data view and in the object navigator. You can specify a dynamic resource from your plug-in module for the relation icon.
<label>	string	Text label for the relation. This label appears in the Related Objects data view and in the object navigator. You can hard code a string value or include a dynamic resource string from your plug-in module resources file.
<listViewId>	string	Extension ID for a list view that can display the relation target object type. For example, a hostForChassis relation can specify the extension ID of a host list view. The vSphere Web Client uses this list view extension to display the list of related target objects.
<relation>	string	Property name that is included in the relation constraint.
<conditionalProperty>	string	Additional property on the relation target object that can be used to constrain which related items are displayed in the vSphere Web Client GUI. You specify the name of a Boolean property on the target object using this property, and only objects for which the property is true appear in the vSphere Web Client GUI.
<targetType>	string	Relation target object type. For example, in the hostForChassis relation, which displays Host objects related to a selected Chassis object, you specify the <targetType> as HostSystem.

Example 8-1, on page 86, presents an example extension definition for a relation extension. In the example, the extension defines relations for the Chassis object type. When the user selects a chassis object, the vSphere Web Client provides related items information for the relations defined in the example. In the example, relations are defined for Rack and Host object types.

Example 8-1. Example Relation Extension for Chassis Entity

```
<!-- Chassis relations -->
<extension id="com.vmware.samples.relateditems.specs.chassis">
      <extendedPoint>vise.relateditems.specs</extendedPoint>
      <object>
         <type>samples:Chassis</type>
         <!-- relationsViewId references the "related items view" extension created
         by the object template. It must be defined for the related items view to
         be shown on the right-hand side.
         -->
         <relationsViewId>com.vmware.samples.chassis.related</relationsViewId>
         <relationSpecs>
            <com.vmware.ui.relateditems.model.RelationSpec>
               <id>rackForChassis</id>
               <icon>#{rack}</icon>
               <label>#{rackLabel}</label>
               <relation>rack</relation>
               <targetType>samples:Rack</targetType>
               <!-- listViewId must be defined for the chassis' related items tab
               to show the rack list. The extension itself gets created as part of
               the object view template. Here we use ${namespace}.list for id.
               -->
               <listViewId>com.vmware.samples.rack.list</listViewId>
            </com.vmware.ui.relateditems.model.RelationSpec>
            <com.vmware.ui.relateditems.model.RelationSpec>
               <id>hostForChassis</id>
               <icon>#{CommonImages:hostsystem}</icon>
               <label>#{Common:typeResource.hostPlural}</label>
               <relation>host</relation>
               <targetType>HostSystem</targetType>
               <!-- listViewId below is the id defined by vSphere client for the HostSystem
               type. This list is shown as one of the items in the chassis Related Items tab.
               Other vSphere types like VirtualMachine, ClusterComputeResource, etc., have
               similar lists are predefined.
               -->
               <listViewId>vsphere.core.host.list</listViewId>
            </com.vmware.ui.relateditems.model.RelationSpec>
         </relationSpecs>
      </object>
</extension>
```

Creating Home Screen Shortcuts 9

The home screen appears in the vSphere Web Client main workspace when the user first logs in to the system. The home screen serves as an entry point to key features in the vSphere Web Client, and contains information about getting started with common tasks in the vSphere environment. The home screen provides shortcuts to solutions and global views in several categories, including Monitoring Tools, Inventories, and Setup.

You can extend the home screen by adding a shortcut. A shortcut extension can provide access to a global view, an inventory list in the Virtual Infrastructure, or other solution in the vSphere Web Client. You can use a home screen shortcut to provide faster access to a data view or application that is not accessible through the top level pages of the object navigator.

This chapter includes the following topics:

Use Cases for Adding a Home Screen Shortcut

You can add a home screen shortcut extension to make your other extension solutions more visible and accessible to vSphere Web Client users. For example, home screen shortcuts are one of only two methods available to make global view extensions accessible to users.

Home screen shortcut extensions can be added for almost any solution or inventory in the vSphere Web Client. Their uses are flexible and will vary for different deployments of the vSphere environment.

Creating the Home Screen Shortcut Extension Definition

All home screen shortcut extensions must reference the `vise.home.shortcuts` extension point. The extension definition must provide an extension object of type `com.vmware.vsphere.client.views.ShortcutSpec`.

Example 9-1 shows an example extension definition for a home screen shortcut extension. In the example, the `<extendedPoint>` element specifies the home screen shortcut extension point. The `<object>` element defines a data object of type `com.vmware.vsphere.client.views.ShortcutSpec`.

Example 9-1. Example Home Screen Shortcut Extension

```
<extension id = "mySolution.myPlugin.sampleGVShortcut">
    <extendedPoint>vise.home.shortcuts</extendedPoint>
    <object>
        <name>Sample Shortcut to Global View</name>
        <categoryUid>vsphere.core.controlcenter.inventoriesCategory</categoryUid>
        <icon>#{samplePluginImages:logo}</icon>
        <targetViewUid>mySolution.myPlugin.myConsoleApp</targetViewUid>
    </object>
</extension>
```

Table 9-1 lists the properties you set for the `com.vmware.vsphere.client.views.ShortcutSpec` object.

Table 9-1. ShortcutSpec Extension Object Properties

Property	Type	Description
`<name>`	string	String value that appears as the title for the shortcut on the vSphere Web Client home screen. The string can be hard coded, or it can be a dynamic resource from your plug-in module.
`<categoryUid>`	string	Home screen category in which the shortcut extension appears. Home screen shortcut extensions must be added to one of the preexisting categories on the home screen. The following are valid values for the `<categoryUid>` property: ■ `vsphere.core.controlcenter.inventoriesCategory` for Inventories ■ `vsphere.core.controlcenter.monitoringCategory` for Monitoring ■ `vsphere.core.controlcenter.administrationCategory` for Administration
`<icon>`	string	Icon that appears for the shortcut. You can specify the icon as a dynamic resource from the resource SWF file in your plug-in module.
`<targetViewUid>`	string	Extension, such as a global view, object workspace, or other solution, that appears in the vSphere Web Client main workspace when the user clicks the shortcut. The value of `<targetViewUid>` property must match the `<extension id="...">` attribute in the target extension's definition.

In Example 9-1, the `<targetViewUid>` property has the value `mySolution.myPlugin.myConsoleApp`. This value matches the extension ID of the example global view extension defined in Chapter 3, "Adding a Global View Extension," on page 33. When the user clicks the shortcut created by the example extension, the `mySolution.myPlugin.myConsoleApp` extension appears in the vSphere Web Client main workspace, and the object navigator displays the appropriate pointer or virtual infrastructure node.

10 Extending vSphere Object List Views

The vSphere Web Client provides a list view for each type of object in the vSphere environment. The list view appears in the main workspace when the user selects an inventory list in the object navigator, or when the user selects the Related Objects data view in an object's workspace. Each list view is a table containing the names of all objects of the relevant type, along with information on status, properties, and related items. For example, the virtual machine object list view contains the names of all virtual machines in your vSphere environment, the power state of each virtual machine object, the related host object for each virtual machine object, and other relevant information.

You can extend the list view for any given vSphere object type by adding one or more columns to the list view table. You can also create a list view for a new type of vSphere object by using the standard template to create an object workspace. See Chapter 4, "Adding to vCenter Object Workspaces," on page 37.

This chapter contains the following topics:

Use Cases for Extending an Object List View

You can extend an existing vSphere object's list view to include additional information about each object in the list. The following are some examples of new information you might add to an object list view.

- You added a new object type to the vSphere environment that is related to the target object, and you want the related object to appear in the target object list view. For example, if you add a new object type called Backup to the vSphere environment, you can extend the Virtual Machine object list to show Backup objects related to a given Virtual Machine.
- You extended the vSphere Web Client Data Service to provide additional properties for the target object type, and you want those properties to appear in the object list view. For example, if you add additional properties to a virtual machine object, you can extend the virtual machine list view to display those properties.
- You want the vSphere Web Client to show an existing property or other information that does not appear as a column in the default object list view.

Defining an Object List View Extension

You extend an object list view by creating an <extension> element in your plugin.xml file. The extension point you specify depends on whether you extend a vSphere object or create a new object type.

Extending a vSphere Object List View

The vSphere Web Client provides an extension point for the list view for each type of vSphere object. The extension point names follow the format `vsphere.core.`*`objectType`*`.list.columns`, where the *`objectType`* placeholder corresponds to the type of vSphere object. For a list of extension points, see Appendix A, "List of Extension Points," on page 113.

Example 10-1 shows part of an example extension definition for an extension that adds a column to the virtual machine object list view. The column shows the name of the host object related to a virtual machine object.

Example 10-1. Example Extension to Object List View

```
<!-- Define the host name column for the vm object list view -->
<extension id="com.mySolution.myPlugin.vm.columns">
   <extendedPoint>vsphere.core.vm.list.list.columns</extendedPoint>
   <object>
      <items>
         ...
         <!-- Host name column -->
         <com.vmware.ui.lists.ColumnContainer>
            <uid>com.mySolution.myPlugin.vm.column.hostame</uid>
            <dataInfo>
               <com.vmware.ui.lists.ColumnDataSourceInfo>
                  <headerText>#{hostName}</headerText>
                  <!-- Object property whose text value will be displayed (1-element array) -->
                  <requestedProperties>
                     <String>n{['hostName']}</String>
                  </requestedProperties>
                  <sortProperty>hostName</sortProperty>
                  <exportProperty>hostName</exportProperty>
               </com.vmware.ui.lists.ColumnDataSourceInfo>
            </dataInfo>
         </com.vmware.ui.lists.ColumnContainer>
         ...
</extension>
```

The `<object>` element `type` attribute specifies the column or columns to add to the target object list view. The properties referenced in the `<requestedProperties>` elements are requested by the platform, with no UI code involved. On the Virgo application server, the Data Access Manager retrieves the requested property for you and returns it to the client.

Adding a List View for a New Object Type

If you added a new object type to the vSphere environment, and you used the standard object view template to create an object workspace for that object, that object workspace contains a list view. For more information on the object view template, see Chapter 4, "Adding to vCenter Object Workspaces," on page 37.

You can extend a new object's list view in the same way as you extend the pre-existing list views in the vSphere Web Client. In your extension definition, you must specify the list extension point created by the new object's template namespace. The format for a new object's list extension point is typically *`namespace`*`.list.columns`. The list extension point for a new object called a Chassis, for example, might appear as follows, where the template namespace is `myExtension.core.Chassis`:

```
myExtension.core.Chassis.list.columns
```

Example 10-2 shows a partial example of an extension definition for an extension that adds a chassis object list view. The list view shows the name of a custom `Chassis` object, in addition to any other properties specified inside `ColumnContainer` elements.

Example 10-2. Example Extension to Object List View

```
<!-- Define the chassis list columns -->
<extension id="com.vmware.samples.chassis.list.sampleColumns">
   <extendedPoint>com.vmware.samples.chassis.list.columns</extendedPoint>
   <object>
```

```
      <items>
         <!-- Chassis name column -->
         <com.vmware.ui.lists.ColumnContainer>
            <uid>com.vmware.samples.chassis.column.name</uid>
            <dataInfo>
               <com.vmware.ui.lists.ColumnDataSourceInfo>
                  <headerText>#{name}</headerText>
                  <!-- Object property whose text value will be displayed (1-element array) -->
                  <requestedProperties>
                     <String>name</String>
                  </requestedProperties>
                  <sortProperty>name</sortProperty>
                  <exportProperty>name</exportProperty>
               </com.vmware.ui.lists.ColumnDataSourceInfo>
            </dataInfo>
         </com.vmware.ui.lists.ColumnContainer>
         ...
</extension>
```

The `<object>` element `type` attribute specifies the column or columns in the target object list view. The properties referenced in the `<requestedProperties>` elements are requested by the platform, with no UI code involved. On the Virgo application server, you need to implement a Data Service Adapter to retrieve data and return it to the client.

Column Visibility

When you extend a list view with a new column, your users will not see the new column when they first install your plug-in. To see the extension, each user must select the new column to appear in the vSphere Web Client list view. The column selection is saved in the user's preferences so that the new column will be visible in future sessions.

vSphere Web Client Service Layer

11

The vSphere Web Client service layer is a collection of Java services that run on the vSphere Web Client application server, called the Virgo server. The Java services on the Virgo server communicate with vCenter Server, ESX hosts, and other data sources within and outside of the vSphere environment.

The principal Java service included in the vSphere Web Client service layer is the Data Service. The Data Service provides data on objects that vCenter Server manages, using a query-based information model. Components in the vSphere Web Client user interface layer, such as Flex data views, send queries to the Data Service for specific objects or attributes. The Data Service processes each query and returns responses.

Flex components in the user interface layer can use a Flex library called the Data Access Manager to communicate with the Data Service in the service layer. The Data Access Manager library is included with the vSphere Web Client SDK. See Chapter 5, "Architecting Data Views," on page 43.

HTML components in the user interface layer communicate with a controller service in the service layer, using a REST API. The controller service can use the Data Service to access data about vSphere objects, or extend the Data Service to access objects outside vSphere. The controller service can also use custom or third-party services to access objects outside vSphere.

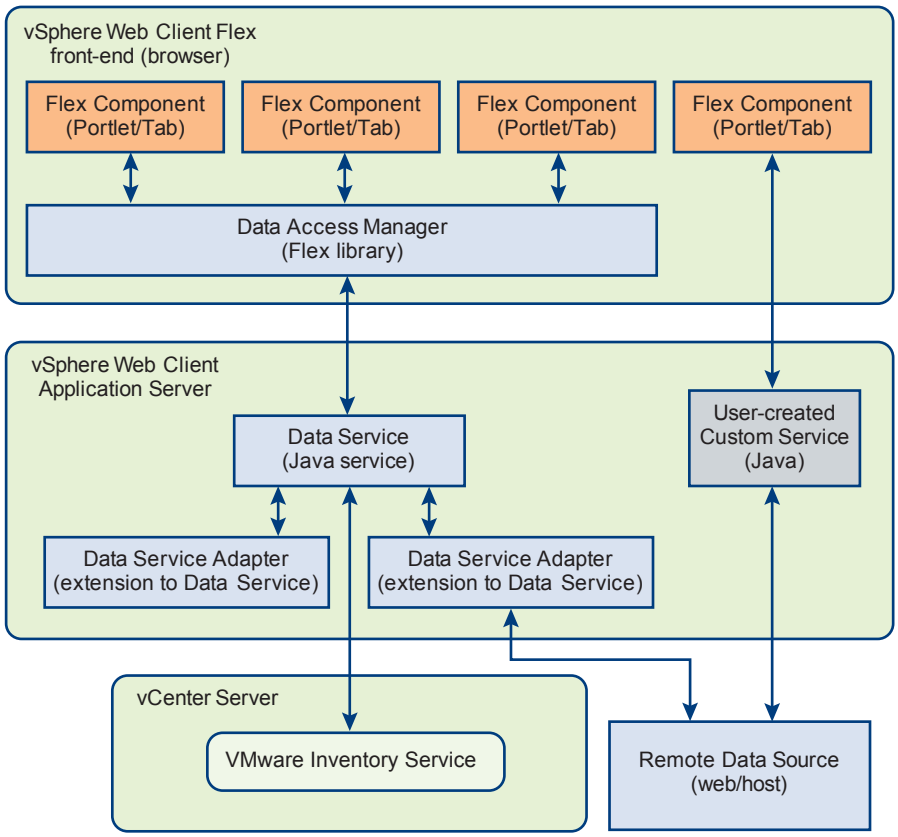
You can extend the vSphere Web Client Data Service to process queries for new data sources. The new data can come from other sources inside the vSphere environment, such as specific ESXi hosts, or from external Web sources. When you extend the vSphere Web Client Data Service, your extensions in the user interface layer can communicate with new data sources by using the existing methods and libraries, such as the Data Access Manager.

You extend the Data Service by creating a Java service called a Data Service Adapter. A Data Service Adapter can either retrieve new properties for existing vSphere objects, or it can retrieve information on new custom objects. You must create different types of Data Service Adapters, depending on whether your environment adds new data to existing vSphere objects, or adds custom objects to the virtual infrastructure.

You can create custom Java services to work with your UI components. These custom Java services are typically used for performing action operations that make changes to the vSphere environment. Custom Java services are generally used as pass-throughs to back-end processes or external data sources. A best practice is to limit your Java service to dispatching requests from the vSphere Web Client and returning data, and implementing extensive or resource-intensive logic on your own external server.

Figure 11-1, on page 94, shows the relationship between Flex components in the vSphere Web Client, Java components on the Virgo server, and data sources in the vSphere environment.

Figure 11-1. vSphere Web Client Data Flow



This chapter includes the following topics:

- "Understanding the vSphere Web Client Data Service" on page 94
- "Extending the Data Service with a Data Service Adapter" on page 95
- "Creating a Custom Java Service" on page 103
- "Importing a Service in a User Interface Plug-In Module" on page 104

Understanding the vSphere Web Client Data Service

The default Data Service provides a stateless, query-based interface to retrieve information about VMODL objects, as defined by the vSphere Web Services API. The default Data Service interface can access data from vCenter Server. The Data Service accesses various services on vCenter Server, including the Inventory and Property Collector services.

User interface components, such as Flex data views, act as Data Service clients. These clients retrieve information by creating Data Service queries. The Data Service processes each query and returns a set of result objects.

If your vSphere Web Client extensions require data from a different source, either within vCenter Server or outside vCenter Server, you can extend the Data Service by creating a Data Service Adapter. A Data Service Adapter provides a way for you to use Data Service queries to retrieve non-VMODL data, such as custom object data.

Extending the Data Service with a Data Service Adapter

If the components in your user interface extensions require data that is not available through the vSphere Web Client Data Service, you can extend the Data Service by creating a Data Service Adapter. A Data Service Adapter is a Java service that integrates with the Data Service, and gives the Data Service the ability to process and respond to Data Service queries for new object types or properties. Data Service Adapters can access data sources within vSphere, or outside data sources.

A Data Service Adapter must implement the same interface and information model as the Data Service. When you create a Data Service Adapter, it must handle Data Service queries and return information as a result set consisting of objects with associated properties.

Advantages of Providing a Data Service Adapter

Extending the Data Service by creating a Data Service Adapter has several advantages.

- The Flex components in your user interface extensions can use the Data Access Manager interface to access the new data. The Data Access Manager provides a consistent data access model throughout the component, easing maintenance and improving code consistency and re-use.
- The Data Service routes queries to the appropriate Data Service Adapters. This mechanism removes any distinction between data sources inside or outside of vSphere, and your extension components can access multiple data sources in a single call.
- Centralizing data access through the Data Service lets your extension components take advantage of services such as logging and error handling.

Designing a Data Service Adapter

To create a Data Service Adapter, you must create a Java service that implements one of the adapter interfaces published by the Data Service. The Data Service publishes interfaces for Property Provider Adapters and Data Provider Adapters. The type of Data Service Adapter you must create depends on the information you want to make available through the Data Service.

Property Provider Adapters

You create a Property Provider Adapter to allow the Data Service to access new properties for existing vSphere objects, such as virtual machines or hosts. For example, your vSphere environment might contain custom virtual machines or hosts that provide extra properties not normally available through the Data Service. You can create a Property Provider Adapter to extend the Data Service to fetch these additional properties.

Data Provider Adapters

You can use a Data Provider Adapter to extend the Data Service to fetch data that is not associated with an existing vSphere object. Typically, you create a Data Provider adapter for one of the following purposes.

- To retrieve information about a new type of object that you have added to the vSphere environment
- To retrieve information from a source outside the vSphere environment

For example, you might create a Data Provider Adapter to handle queries for a new type of vSphere object called a Chassis. You might also use a Data Provider Adapter to display data in the vSphere Web Client from an external Web source separate from vCenter Server.

Implementing an Adapter

To implement one of the adapter interfaces, your Java service must import the package `com.vmware.vise.data.query` from the vSphere Web Client Java SDK.

After you create the adapter service, you must add the adapter service to Virgo Server framework and register the adapter with the Data Service. You register an adapter by using the vSphere Web Client SDK `DataServiceExtensionRegistry` service, typically within your adapter's constructor method. See "Registering a Property Provider Adapter" on page 96 and "Registering a Data Provider Adapter" on page 99.

The registration process declares what types of objects and properties the Data Service Adapter can provide. When the Data Service receives a query for one of the registered object or property types, the Data Service routes the query to the proper Data Service Adapter.

Property Provider Adapters

Queries to a Property Provider Adapter accept one or more specific vSphere objects, and return one or more properties for those objects. A Property Provider Adapter registers with the Data Service to advertise which types of properties it can return. When the Data Service receives a query for one of the registered property types, the Data Service routes the query to the appropriate Property Provider Adapter for processing.

PropertyProviderAdapter Interface

A Property Provider Adapter must implement the `PropertyProviderAdapter` interface in the `com.vmware.vise.data.query` Java SDK package. The `PropertyProviderAdapter` interface publishes a single method named `getProperties()`. Your Property Provider Adapter service must provide an implementation of this method. The Data Service calls your adapter's `getProperties()` method in response to an appropriate query for the properties your adapter is registered to provide.

The method implementation in your service must accept as its parameter an object of type `com.vmware.vise.data.query.PropertyRequestSpec`, and must return an object of type `com.vmware.vise.data.query.ResultSet`.

```
public ResultSet getProperties(PropertyRequestSpec propertyRequest)
```

Your service's implementation of the `getProperties()` method can retrieve and format data in any way you choose. However, your implementation must return the results as a `ResultSet` object. You use the `PropertyRequestSpec` object to obtain the query's list of target vSphere objects and desired properties. The `PropertyRequestSpec` object contains an `objects` array and a `properties` array, which respectively contain the target vSphere objects and requested properties.

For additional information on `ResultSet`, `PropertyRequestSpec`, and other features in the `com.vmware.vise.data.query` package, see the Java API reference included in the vSphere Web Client SDK.

Registering a Property Provider Adapter

You must register your Property Provider Adapter for the adapter to work with the Data Service. You register your Property Provider Adapter with the Data Service by using the `DataServiceExtensionRegistry` service. The `DataServiceExtensionRegistry` service contains a method named `registerDataAdapter()` that you must call to register your Property Provider Adapter.

A best practice for registering your adapter is to pass `DataServiceExtensionRegistry` as a parameter to your Property Provider Adapter class constructor, and call `registerDataAdapter()` from that constructor.

Example Property Provider Adapter

Example 11-1, on page 97, shows an example of a Property Provider Adapter class. The class constructor method registers the adapter with the Data Service.

Example 11-1. Example Property Provider Adapter Class

```
package com.myAdapter.PropertyProvider;

import com.vmware.vise.data.query;
import com.vmware.vise.data.query.PropertyProviderAdapter;
import com.vmware.vise.data.query.ResultSet;
import com.vmware.vise.data.query.type;

public class MyAdapter implements PropertyProviderAdapter {

   public MyAdapter(DataServiceExtensionRegistry extensionRegistry) {
      TypeInfo vmTypeInfo = new TypeInfo();
      vmTypeInfo.type = "VirtualMachine";
      vmTypeInfo.properties = new String[] { "myVMdata" };
      TypeInfo[] providerTypes = new TypeInfo[] {vmTypeInfo};

      extensionRegistry.registerDataAdapter(this, providerTypes);
   }

   @Override
   public ResultSet getProperties(PropertyRequestSpec propertyRequest) {
       // Logic to retrieve properties and return as result set
   }
...
}
```

In Example 11-1, the class constructor method `MyAdapter()` constructs an array of property types that the adapter can supply to the Data Service in the array named `providerTypes`. The constructor then calls the Data Service Extension Registry method named `registerDataAdapter` to register the Property Provider Adapter with the Data Service.

The Data Service calls the override method `getProperties()` when the Data Service receives a query for the kinds of properties that were specified at registration. The `getProperties()` method must retrieve the necessary properties, format them as a `ResultSet` object, and return that `ResultSet`.

Data Provider Adapters

A Data Provider Adapter is responsible for all aspects of data retrieval, including parsing a query, computing the results of access operations, finding the matching objects or properties, and formatting results as responses compatible with the Data Service. You can use a Data Provider Adapter to retrieve almost any data, including data agnostic to vSphere, provided that you can format it as a set of objects and related properties.

Typically, you use a Data Provider Adapter to retrieve data on custom objects that you added to your vSphere environment. The specific implementation of the Data Provider Adapter's data access depends on the data source for your custom object. Your Data Provider Adapter might query a database for configuration data, or retrieve operational data directly from a particular device.

When designing a Data Provider Adapter, consider the following requirements:

- You must be able to represent the external data using the same object and property model as the vSphere Web Client Data Service.
- The Java service that you create to act as the Data Provider Adapter must perform all necessary data fetching operations from your remote data source.
- The service you create must process Data Service queries and return Data Service result sets.

DataProviderAdapter Interface

A Data Provider Adapter must implement the `DataProviderAdapter` interface in the `com.vmware.vise.data.query` Java SDK package. The `DataProviderAdapter` interface publishes a single method named `getData()`. Your Data Provider Adapter service must provide an implementation of this method. The Data Service calls your adapter's `getData()` method in response to the queries your adapter is registered to process.

Your implementation of the `getData()` method must accept an object of type `com.vmware.vise.data.query.RequestSpec` as a parameter, and must return an object of type `com.vmware.vise.data.query.Response`.

```
public Response getData(RequestSpec request)
```

The `RequestSpec` object parameter to the `getData()` method contains an array of Data Service query objects. Each query contains a target object and one or more constraints that define the information that the client requests, as well as the expected format for results.

Your `getData()` method determines what information it must fetch by processing each Data Service query and handling the included constraints. The `getData()` method must then retrieve that information, through whatever means your data source provides, such as a database query or a remote device method.

Your `getData()` method must format the retrieved information as a specific result type for each query, and then return those results as an array, packaged in a `Response` object.

Processing Data Service Queries

Data Service queries are passed to your Data Service Adapter through the `com.vmware.data.query.RequestSpec` object parameter. A `RequestSpec` object consists of an array of objects of type `com.vmware.data.query.QuerySpec`, each of which represents an individual query.

Each `QuerySpec` object defines the query target, the query constraints, and the expected formatting for the query results.

Query Target Object

The query target is the object for which your `getData()` method must retrieve information. In the QuerySpec object, the target is represented as an object of type `com.vmware.data.query.ResourceSpec`

Your `getData()` method can determine what information it must retrieve by using the values in the `ResourceSpec` object. The `ResourceSpec` object specifies the target object as a `String` type, which contains a URI for the target object. The requested properties are contained in an object of type `com.vmware.data.query.PropertySpec`.

Handling Constraints

Within the `QuerySpec` object, the query constraints are represented as an object of type `com.vmware.data.query.Constraint`. A query can specify the following types of constraints, each of which is a subclass of the base `Constraint` class.

- **ObjectIdentityConstraint**. Queries based on this constraint retrieve the properties of a known target object. For example, a query might retrieve the powered-on state of a given virtual machine. The object identifier can be any type that implements the `IResourceReference` interface.
- **PropertyConstraint**. Queries based on this constraint retrieve all objects with a given property value. For example, a query might retrieve all virtual machine objects with a power state of `on`. This constraint accepts the property name and comparator as strings, and the property value as an `Object`. `PropertyConstraint` is roughly analogous to a `SELECT` statement in a database query.
- **RelationalConstraint**. Queries based on this constraint retrieve all objects that match the specified relationship with a given object. For example, a query might retrieve all virtual machine objects related to a given host object. `RelationalConstraint` is roughly analogous to a `JOIN` statement in a database query.

- **CompositeConstraint**. Composite queries allow the combination of multiple constraints using the `and` or `or` operator, passed as a string. The combined subconstraints in `CompositeConstraint` are contained in an array of `Constraint` objects.

When processing constraints, a best practice is to read the entire set of constraints and then determine the most efficient processing order. For example, you can process relational constraints first to retrieve a smaller number of objects that meet any included property constraints.

Specifying Result Sets

In the `QuerySpec` object, the expected formatting for the query results are included in an object of type `com.vmware.data.query.ResultSpec`. The properties of the `ResultSpec` object specify a maximum number of results for the query to return, provide an offset into the returned results, and set ordering for the returned results. Your `getData()` method must use the values of the `ResultSpec` properties to format the information it has retrieved.

Resolving the Target Object

The target object for a query is identified by a Uniform Resource Identifiers (URI) string, which is a unique identifier for a specific custom object. In your Data Provider Adapter, you must resolve the URI for a query target object to the correct custom object type.

Implementing a Resource Type Resolver

A best practice is to use a Resource Type Resolver to resolve a URI to the correct custom object type. To use a Resource Type Resolver, you must create a Java class that implements the interface `com.vmware.vise.data.uri.ResourceTypeResolver`.

The class you create to implement `ResourceTypeResolver` must support the following methods.

String getResourceType(URI uri) The `getResourceType()` method must parse a URI and return a `String` containing the type of custom object to which the URI pertains. For example, for a URI that referred to a custom Chassis object, the `getResourceType()` method must return the `String samples:Chassis`.

String getServerGuid(URI uri) The `getServerGuid()` method must parse a URI and return a `String` containing the server global unique identifier for the URI target object. For example, for the URI string `urn:cr:samples:Chassis:server1/ch-2`, the `getServerGuid()` method must return the string `server1`.

Registering a Resource Type Resolver

To use your Resource Type Resolver, you must register the resolver with the Data Service. You typically register the Resource Type Resolver in your Data Provider Adapter class constructor by using the Resource Type Resolver Registry service, an OSGI service included with the vSphere Web Client. You must use the vSphere Web Client Spring framework to pass the Resource Type Resolver Registry OSGI service as an argument to your class constructor method. See "Passing Arguments to Your Class Constructor" on page 100.

Example 11-2, "Example Data Provider Adapter Class," on page 102, shows an example of how to register a Resource Type Resolver.

Registering a Data Provider Adapter

You must register your Data Provider Adapter for the adapter to work with the Data Service. You can register an adapter implicitly by declaring the Java service as an OSGI bundle, or you can register an adapter explicitly by using the Data Service Extension Registry service.

Registering Implicitly

You can register your Data Provider Adapter implicitly when you add the adapter to the Virgo server framework. To use implicit registration, you must declare the Java service that implements your Data Provider Adapter as an OSGI bundle when you add the service to the Virgo server framework. The vSphere Web Client detects new OSGI bundles as they are added and registers the Data Provider Adapters with the Data Service. You must also annotate the adapter class with the object types that the adapter supports.

Declaring the Service as an OSGI Bundle To declare the service as an OSGI bundle, you must define your adapter's Java service as a Java Bean in the `bundle-context.xml` file. You can find the `bundle-context.xml` file your plug-in module's `src/main/resources/META-INF/spring` folder.

To define the Java Bean, you must add the following XML element to the `bundle-context.xml` file.

```
<bean name="MyDataProviderImpl" class="com.example.MyDataProviderAdapter"> </bean>
```

The `name` attribute is an identifier that you choose for the Java Bean. You must set the value of the `class` attribute to the fully qualified class name of the Java class you have created that implements the `DataProviderAdapter` interface.

After you define your Data Provider Adapter as a Java Bean, you must modify the `bundle-context-osgi.xml` file to include the Java Bean as an OSGI service. The `bundle-context-osgi.xml` file is in your plug-in module's `src/main/resources/META-INF/spring` folder.

You must add the following XML element to the `bundle-context-osgi.xml` file.

```
<osgi:service id="MyDataProvider" ref="MyDataProviderImpl"
              interface="com.vmware.vise.data.query.DataProviderAdapter" />
```

The `id` attribute is an identifier that you choose for the Data Provider Adapter. You must set the value of the `ref` attribute to the same value as the `name` attribute that you defined when declaring your Java Bean. The `interface` attribute must be set to the fully qualified class name of the `DataProviderAdapter` interface.

You must update the `src/main/resources/META-INF/MANIFEST.MF` file to reflect any Java packages from the vSphere Web Client SDK that your Data Provider Adapter imports. You add the imported packages to the `import-packages` section of the `MANIFEST.MF` file.

In Example 11-2, on page 102, the example Data Provider Adapter imports the packages `com.vmware.vise.data.uri` and `com.vmware.data.query`. For the example, the `MANIFEST.MF` file must list those packages in the `import-package` section.

```
Import-Package: org.apache.commons.logging,
com.vmware.vise.data,
com.vmware.vise.data.query,
com.vmware.vise.data.uri
```

Annotating the Adapter Class You must annotate your Data Provider Adapter class with the object types for which the adapter processes queries. The vSphere Web Client uses these annotations to route queries for the specific types to the correct adapters. You use the `@type` tag to create the necessary annotation.

For example, if you have a custom object of type `WhatsIt`, you annotate the class like the following example:

```
@type("samples:WhatsIt") // declares the supported object types
public class MyAdapter implements DataProviderAdapter {
...
```

Passing Arguments to Your Class Constructor Most Data Provider Adapters use other OSGI services that the vSphere Web Client SDK provides. These services include the base Data Service, the Resource Type Resolver Registry, and the vSphere Object Reference Service. You can pass these OSGI services to your Data Provider Adapter as arguments to the Data Provider Adapter class constructor method.

All Data Provider Adapters can include the vSphere Web Client Data Service. To include the Data Service as an argument to your Data Provider Adapter class constructor, you add the following element to your service's `bundle-context-osgi.xml` file.

```
<osgi:reference id="dataService" interface="com.vmware.vise.data.query.DataService" />
```

If your Data Provider Adapter handles queries for multiple custom object types, you must include the Resource Type Resolver Registry OSGI service and register a Resource Type Resolver. To include the Resource Type Resolver Registry OSGI service as an argument to your Data Provider Adapter class constructor, you add the following element to your service's `bundle-context-osgi.xml` file.

```
<osgi:reference id="uriRefTypeAdapter"
              interface="com.vmware.vise.data.uri.ResourceTypeResolverRegistry" />
```

If your Data Provider Adapter handles queries for built-in vSphere object types, such as Hosts or Virtual Machines, you can include the vSphere Object Reference Service. To pass the vSphere Object Reference Service as an argument to your Data Provider Adapter class constructor, you add the following element to your service's `bundle-context-osgi.xml` file.

```
<osgi:reference id="vimObjectReferenceService"
                interface="com.vmware.vise.vim.data.VimObjectReferenceService" />
```

Your Data Provider Adapter can use the User Session Service to get information about the current user session. To pass the User Session Service as an argument to your Data Provider Adapter class constructor, you add the following element to your service's `bundle-context-osgi.xml` file.

```
<osgi: reference id="userSessionService"
                interface="com.vmware.vise.usersession.UserSessionService" />
```

If you pass OSGI services to your Data Provider Adapter's class constructor, you must include those constructor arguments when you declare your Data Provider Adapter as a Java Bean in the `bundle-context.xml` file. See "Declaring the Service as an OSGI Bundle" on page 100.

For each service your Data Provider Adapter includes, you must add a `<constructor-arg>` element to your adapter's Bean definition. In each `<constructor-arg>` element, you set the `ref` attribute to the same value as the `id` attribute in the `<osgi:reference>` element in the `bundle-context-osgi.xml` file.

If your Data Provider Adapter uses the Data Service, vSphere Object Reference Service, Resource Type Resolver Registry, and User Session Service, the Bean definition might appear as follows.

```
<bean name="MyDataProviderImpl" class="com.example.MyDataProviderAdapter">
    <constructor-arg ref="dataService"/>
    <constructor-arg ref="uriRefTypeAdapter"/>
    <constructor-arg ref="vimObjectReferenceService"/>
    <constructor-arg ref="userSessionService"/>
</bean>
```

Registering Explicitly

You can register your Data Provider Adapter with the Data Service by using the `DataServiceExtensionRegistry` service. `DataServiceExtensionRegistry` contains a `registerDataAdapter()` method that you must call to register your Data Provider Adapter.

A common way to register your adapter is to pass `DataServiceExtensionRegistry` as a parameter to your Data Provider Adapter class constructor, and call `registerDataAdapter()` from within that constructor.

Data Provider Adapter Example

Example 11-2, on page 102, presents an example of a Data Provider Adapter class that supports hypothetical `WhatsIt` objects. In the example, the class constructor method initializes the class member variables for the Data Service and registers a Resource Type Resolver. The example assumes that the Data Provider Adapter is registered implicitly by registering the service as an OSGI bundle. The Data Service and Resource Type Resolver Registry services are passed as arguments to the class constructor.

As a best practice, you can initialize the other services that your Data Provider Adapter requires in your Data Provider Adapter class constructor. These might include the vSphere Web Client Data Service, the Resource Type Resolver Registry if your adapter handles multiple custom object types, and the vSphere Object Reference Service if your adapter requires data from regular vSphere objects.

For more complete examples of Data Provider Adapters, see the sample extensions included in the vSphere Web Client SDK download file.

Example 11-2. Example Data Provider Adapter Class

```
package com.MyAdapter.DataProvider;

import java.net.URI;

import com.vmware.vise.data.uri.ResourceTypeResolverRegistry;
import com.vmware.vise.data.query.DataProviderAdapter;
import com.vmware.vise.data.query.QuerySpec;
import com.vmware.vise.data.query.RequestSpec;
import com.vmware.vise.data.query.Response;
import com.vmware.vise.data.query.type;

@type("samples:WhatsIt") // type that the adapter supports
public class MyAdapter implements DataProviderAdapter {

    private final DataService _dataService;

    // Resource resolver, used to resolve the URIs of objects serviced by this adapter
    private static final ModelObjectUriResolver RESOURCE_RESOLVER = new ModelObjectUriResolver();

    // constructor method
    public MyAdapter( DataService dataService,
                      ResourceTypeResolverRegistry typeResolverRegistry )
    {

        if ( dataService = null || typeResolverRegistry == null ) {
            throw new IllegalArgumentException("MyAdapter constructor arguments must be
                          non-null.");
        )
        _dataService = dataService;
        try {
            // Register the Resource Type resolver for multiple custom object types
            typeResolverRegistry.registerSchemeResolver(
                ModelObjectUriResolver.SCHEME,
                RESOURCE_RESOLVER);
        } catch (UnsupportedOperationException e) {
            _logger.warn("ModelObjectUriResolver registration failed.", e);
        }
    }

    @Override
    // All query requests for the types supported by this adapter are routed here by the vSphere
    // Web Client Data Service; this method is the starting point for processing constraints,
    // discovering objects and properties, and returning results
    public Response getData(RequestSpec request) {
          QuerySpec[] querySpecs = request.querySpec;
          List<ResultSet> results = new ArrayList<ResultSet>(querySpecs.length);
          for (QuerySpec qs : querySpecs) {
             // Call your logic for query processing, constraint processing, object discovery:
             ResultSet rs = processQuery(qs);
             results.add(rs);
          }
          Response response = new Response();
          response.resultSet = results.toArray(new ResultSet[]{});
          return response;
    }
```

The `getData()` method is called by the vSphere Web Client Data Service when it receives a query for one of the objects or properties specified at registration. In the `getData()` method, your Data Provider Adapter must parse the query, compute the results, and return that result data as a `Response` object. For a more complete example, see the `ChassisDataAdapter` class in the SDK.

Creating a Custom Java Service

You can extend the vSphere Web Client service layer with your own Java services. Typically, you create a Java service if your user interface extensions adds an action to the vSphere Web Client, where the Java service performs the action operation on the virtual infrastructure. You can also add a Java service to perform a complex calculation, retrieve data from an external source, or perform other miscellaneous tasks.

To add a Java service, you must provide a Java Archive (JAR) bundle that contains the classes in the service. Inside the JAR bundle, you must add an XML configuration file that declares all of the Java objects that the service adds to the vSphere Web Client Virgo server framework. The Virgo server uses Spring as the application server framework.

Making Java Services Available to UI Components in the vSphere Web Client

To make a custom Java service available to your extension components in the vSphere Web Client, complete the following tasks.

1 Create a Java interface for the service.

2 Create a Java class that implements the interface in Step 1.

3 Add the service to the Virgo server framework. You must export and expose the service to the framework by adding it as a `bean` in the Spring configuration Virgo server. See "Packaging and Exposing the Service" on page 103.

4 Import the service where your extension references it.

- For Flex-based extensions, import the service into the user interface plug-in module that contains your Flex components.
- For HTML-based extensions, import the service in the controller module that services your extension's data requests.

5 For Flex-based extensions, use ActionScript to create a proxy class in your Flex component. The proxy class is used to communicate between the user interface plug-in module and the service. HTML-based extensions access the service using a REST API that communicates with the controller module on the Virgo server.

Creating the Java Interface and Classes

To integrate with the Virgo server Spring framework, the Java service you create must provide separate interface and implementation classes. Example 11-3 shows a basic example of an interface class and an implementation class.

Example 11-3. Basic Java Service Interface and Implementation

```
package com.vmware.myService;

public interface MyService {
    String echo (String message);
}

public class MyServiceImpl implements MyService {
    public String echo (String message) {
        return message;
    }
}
```

Packaging and Exposing the Service

To make your Java service available for use with the vSphere Web Client, you must export the service and add it to the Spring configuration on the Virgo server. Spring uses the OSGI model to share Java libraries.

Exporting the Service

You must locate the /src/main/resources/META-INF/MANIFEST.MF file in your service JAR bundle and ensure that the Java service package is exported. To export the package, the following line must appear in the MANIFEST.MF file:

```
Export-Package: com.vmware.myService
```

In the example line, com.vmware.myService is the name of the service package you created.

Adding the Service to the Spring Configuration

You add your service to the Spring configuration on the Virgo server by creating a <bean> element in the Spring configuration file. In the JAR bundle, locate the /src/main/resources/META-INF/spring/bundle-context.xml file. The file contains a <beans> XML element containing services in the configuration. Add your service as a new <bean> as follows:

```
<bean name="myServiceImpl" class="com.vmware.myService.MyServiceImpl"/>
```

The name attribute is the name of your service implementation, and the class attribute contains the class you created that implements the service interface.

You must also expose the service interface as an OSGI bundle in the Spring framework. In the JAR bundle, locate the /src/main/resources/META-INF/spring/bundle-context-osgi.xml file. This file also contains a <beans> XML element. Add your service using the following line:

```
<osgi:service id="myService" ref="myServiceImpl" interface="com.vmware.myService.MyService"/>
```

The id attribute is the name of your service, the ref element specifies the service implementation you added to the bundle-context.xml file, and the interface element contains the class that defines the service interface.

Importing a Service in a User Interface Plug-In Module

To use a Java service you created and exposed in the vSphere Web Client service layer, a user interface plug-in module must import the service. You import the service by updating two metadata configuration files within your user interface plug-in module Web Archive (WAR) bundle.

In your user interface plug-in module WAR bundle, locate the /war/src/main/webapp/META-INF/MANIFEST.MF file and add the following lines:

```
Import-Package:
    com.vmware.myService
```

com.vmware.myService is the name of the service package you created.

Specifying Flex-to-Java Service Parameters

If you are adding a Flex extension, you need to specify the parameters for the Flex-to-Java framework. In the WAR bundle, locate the /war/src/main/webapp/WEB-INF/spring/bundle-context.xml file. This file specifies the necessary service parameters for the Flex-to-Java framework on the vSphere Web Client application server. Inside the <beans> element of the bundle-context.xml file, create the service references as follows:

```
<flex:message-broker id="myService-ui-broker"
                services-config-path="/WEB-INF/flex/services-config.xml"/>
<osgi:reference id="myService" interface="com.vmware.myService.MyService"/>
<flex:remoting-destination ref="myService" message-broker="myService-ui-broker"/>
```

The <flex:message broker> and <flex:remoting-destination> elements declare your service as a destination for Flex remote object invocation.

Creating a Proxy Class with ActionScript

Using a proxy class in your extension Flex component is the recommended way to manage communication between a custom Java service and your Flex data views. The vSphere Web Client includes a Flex utility library that includes a base proxy class. You can use this base proxy class to implement the proxy class that communicates with your service.

For more information about proxy classes and their role in the user interface layer, see "Architecting Data Views" on page 43.

Example 11-4, on page 105, presents a sample ActionScript proxy class implementation. The sample proxy class extends the `com.vmware.flexutil.proxies.BaseProxy` class that the vSphere Web Client provides.

Example 11-4. Example ActionScript Proxy Class

```
package com.vmware.samples.globalview {
import com.vmware.flexutil.proxies.BaseProxy;

/**
* Proxy class for the EchoService java service
*/
public class EchoServiceProxy extends BaseProxy {
   // Service name matching the flex:remoting-destination declared in
   // main/webapp/WEB-INF/spring/bundle-context.xml
   private static const SERVICE_NAME:String = "myService";

   // channelUri uses the Web-ContextPath define in MANIFEST.MF (globalview-ui)
   // A secure AMF channel is required because vSphere Web Client uses https
   private static const CHANNEL_URI:String =
         "/" + GlobalviewModule.contextPath + "/messagebroker/amfsecure";

   /**
    * Create a EchoServiceProxy with a secure channel.
    */
   public function EchoServiceProxy() {
      super(SERVICE_NAME, CHANNEL_URI);
   }

   /**
    * Call the "echo" method of the EchoService java service.
    *
    * @param message    Single argument to the echo method
    * @param callback   Callback in the form <code>function(result:Object,
    *       error:Error, callContext:Object)</code>
    * @param context    Optional context object passed back with the result
    */
   public function echo(message:String, callback:Function = null, context:Object = null):void {
      // "echo" takes a single message argument but callService still requires an array.
      callService("echo", [message], callback, context);
   }
}
}
```

You must set the Proxy constructor `destination` argument to the service ID that you imported in your plug-in module configuration files. In Example 11-4, the proxy constructor sets the `destination` parameter to the service ID `myService`, as defined in the WAR bundle configuration file (`/war/src/main/webapp/WEB-INF/spring/bundle-context.xml`).

In the proxy, you can call functions of the Java service by using the `callService` method. The `callService` method is included in the package `com.vmware.flexutil.ServiceUtil`. In Example 11-4, the proxy class uses the `callService` method to call the `echo` method in the Java service.

Creating and Deploying Plug-In Packages

12

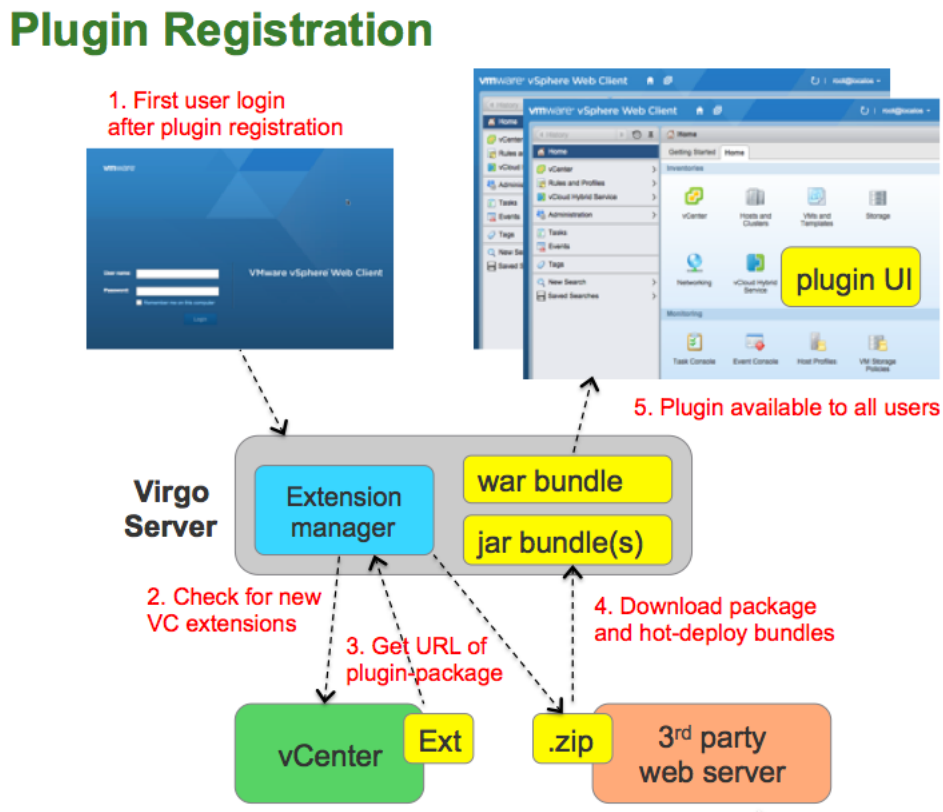
In the vSphere Web Client, you deploy extension solutions using plug-in packages. Each plug-in package can contain both user interface plug-in modules and service plug-in modules, and manages the deployment of those modules. The vSphere Web Client extensibility framework can perform live hot deployment of the plug-in modules in a package.

This chapter includes the following topics:

Overview of Plug-In Packages

A plug-in package is a ZIP archive file that contains all of the plug-in modules in your extension solution along with a package manifest. The package manifest describes deployment information for each plug-in module using XML metadata. The vSphere Web Client Extension Manager uses this metadata to install and deploy each plug-in module in the plug-in package.

Figure 12-1. Plug-In Deployment



Creating a Plug-In Package

To create a plug-in package, you must create a ZIP archive file with the following structure:

- At the root level, add a `plugin-package.xml` file to the root folder.
- At the root level, add a `plugins` folder.
- Inside the `plugins` folder, add one or more WAR files containing the plug-in UI modules.
- Inside the `plugins` folder, add zero or more JAR files, one for each Java service component created for your plug-in.
- Inside the `plugins` folder, add zero or more JAR files, one for each 3rd-party Java library used by your plug-in.

You can use any text or XML editor to create the `plugin-package.xml` file.

NOTE Each WAR file or JAR file must contain an OSGI-compliant `META-INF/MANIFEST.MF` file that describes the bundle.

Creating the Package Manifest File

The plug-in package manifest file specifies general information about the plug-in package, the deployment order for the plug-in modules in the package, and any dependencies for the plug-in package.

XML Elements in the Manifest File

The metadata in the manifest file follows a specific XML schema. The `<pluginPackage>` root element encapsulates the entire plug-in package manifest. The `<pluginPackage>` element can contain the `<dependencies>` element and the `<bundlesOrder>` element.

Example 12-1, on page 109, shows an example of a `plugin-package.xml` manifest file.

Example 12-1. Example plugin-package.xml file

```
<pluginPackage id = "com.vmware.client.myPackage"
    version="1.0.0"
    name="My Plugin Name"
    description="Demo package version 1"
    vendor="VMware"
    iconUri="assets/packageIcon.png">

    <dependencies>
        <pluginPackage id = "com.vmware.vsphere.client" version="5.5.0" />
    </dependencies>

    <bundlesOrder>
        <bundle id="com.mySolution.myDataServicePlugin" />
        <bundle id="com.mySolution.myUIViewPlugin" />
        <bundle id="com.mySolution.myActionPlugin" />
    </bundlesOrder>

</pluginPackage>
```

<pluginPackage> Element

The <pluginPackage> element is the root element of any plug-in package manifest file. The following attributes of the <pluginPackage> contain information about the entire package.

- **id**. The package's unique identifier, which you create. Each plug-in package name must be unique. A best practice is to use namespace notation, such as com.myCompany.PackageName.
- **version**. A dot-separated string containing the package version number, such as 1.0.0.
- **description.** A short description of the package.
- **vendor.** The name of the package vendor.
- **iconUri.** The URI of an icon to represent the package. The location is specified relative to the manifest file.

<dependencies> Element

The <dependencies> element defines any dependencies upon other plug-in packages. In the <dependencies> element, you specify each specific package dependency with a <pluginPackage> element. Each <pluginPackage> element in the <dependencies> element must have the following attributes.

- **id.** The unique identifier of the package that your package is dependent on.
- **version.** The version number of the package that your package is dependent on.
- **match.** The version matching policy. Possible values are equal, greaterThan, lessThan, greaterOrEqual, or lessOrEqual. The match attribute is optional and defaults to greaterOrEqual if omitted.

<bundlesOrder> Element

The <bundlesOrder> element specifies the order in which locally hosted plug-in modules are deployed to the vSphere Web Client. If your plug-in package contains both service plug-in modules and user interface plug-in modules, a best practice is to deploy the service plug-in modules first, because the user interface plug-in modules might import those services.

You specify each plug-in module using a <bundle> element inside the <bundlesOrder> element. The <bundle> element's id attribute contains the unique identifier of the plug-in module. The value of the id attribute must match the Bundle-SymbolicName specified in the plug-in module MANIFEST.MF file included in the WAR bundle.

NOTE Plug-in modules in the package that are not explicitly specified in the <bundlesOrder> list are still deployed, but in an undefined order.

Deploying a Plug-In Package

You deploy a plug-in package to the vSphere Web Client by registering the package as an extension on vCenter Server. When you register your plug-in as an extension on vCenter Server, your plug-in becomes available to any vSphere Web Client that connects to your vSphere environment.

You must register your plug-in on every vCenter Server where you need to use it. When a vSphere Web Client connects to a vCenter Server where your plug-in is not registered, the plug-in is not visible to the client.

When a vSphere Web Client establishes a user session to vCenter Server, the vSphere Web Client application server queries vCenter Server for a list of all available plug-in packages that are registered as vCenter extensions. Any plug-in packages that are not present on the application server are downloaded and installed.

NOTE Plug-in package .ZIP files are not always installed locally on a vCenter server. Some vCenter extensions can reference remote packages at any Web server URL. If a remotely hosted package is found on vCenter Server, the package .ZIP file is downloaded from the URL specified in the vCenter extension data.

The vSphere Web Client application server can run only one version of each plug-in package. If a plug-in package is present on the application server, but has an older version number than the corresponding package on vCenter Server, the package version on vCenter Server replaces the older package.

Registering a Plug-In Package as a vCenter Server Extension

To register your plug-in package as an extension with vCenter Server, you must create an `Extension` data object and register this data object with the vCenter Server `ExtensionManager`. See "Creating the vCenter Server Extension Data Object" for information on the XML properties.

You can create and register an `Extension` data object in the following ways:

- Use a utility application or script to create the `Extension` data object programatically, and register that data object using the vSphere API. You can use the `ExtensionManager.registerExtension()` method to register the data object.
- Use the traditional vSphere Client to register the extension.

  1. Create the `vim.Extension` data object in an XML file, and place that file in a file system available to the vSphere Client.
  2. Run the Windows vSphere Client and connect to the vCenter Server where you need to register your extension.
  3. From the Plug-ins menu select **Manage Plug-ins**.
  4. In the Plug-in Manager window right-click in an empty area at the bottom of the window, or in an empty column at the right edge of the window, and select **New Plug-in**.
  5. In the Register Plug-in window enter a file name or click the Browse button and locate the `vim.Extension` file.
  6. Click **Register Plug-in**.

Creating the vCenter Server Extension Data Object

Regardless of the registration method you choose, you must set the properties of the `Extension` data object as follows:

- **key**. The plug-in package id, defined in your plug-in package's manifest, `plugin-package.xml`.
- **client**. This property must contain one `ExtensionClientInfo` data object, with the following properties:
  - **type**. Must be set to `vsphere-client-serenity`.
  - **url**. The location of the plugin-package folder, which must be a `.zip` file. The ZIP archive root folder must contain the `plugin-package.xml` package manifest.
  - **server**. (conditional) If the URL uses HTTPS, you must define a `<server>` property in your extension data object. The `<server>` property must contain the SSL thumbprint for the server where your plug-in package ZIP file is stored. For information about the `<server>` property, see "Using a Secure URL for the Plug-in Location" on page 111.
  - **version**. The dot-separated version number of the plug-in package, defined in `plugin-package.xml`.

Example 12-2 shows an example `Extension` object defined in an XML file.

Example 12-2. Example vim.Extension XML Definition

```
<extension>
    <description>
        <label>My plugin</label>
        <summary>My first vSphere Client plugin</summary>
    </description>
    <key>com.acme.myPlugin.MyPlugin</key>
    <company>VMware</company>
    <version>1.0.0</version>
    <client>
        <version>1.0.0</version>
        <description>
            <label>My plugin</label>
            <summary>My first vSphere Client plugin</summary>
        </description>
        <company>VMWare</company>
        <type>vsphere-client-serenity</type>
        <url>http://a-web-server-path/mypluginPackage.zip</url>
    </client>
    <lastHeartbeatTime>2012-07-21T00:25:52.814418Z</lastHeartbeatTime>
</extension>
```

Using a Secure URL for the Plug-in Location

A best practice is to use a secure URL (HTTPS) for your plug-in package `.zip` file location. If you use an HTTPS URL, you must include a `<server>` property in your `vim.Extension` data object. The `<server>` property contains the SHA1 thumbprint for the server that corresponds to the URL. Example 12-3, on page 112, shows an example `server` property.

Example 12-3. Example <server> Property with SHA1 Thumbprint

```
<extension>
...
    <server>
        <url>https://myhost/helloworld-plugin.zip</url>
        <description>
            <label>Helloworld</label>
            <summary>Helloworld sample plugin</summary>
        </description>
        <company>VMware</company>
        <!-- SHA1 Thumbprint of the server hosting the .zip file -->
        <serverThumbprint>
            3D:E7:9A:85:01:A9:76:DD:AC:5D:83:1C:0E:E0:3C:F6:E6:2F:A9:97
        </serverThumbprint>
        <type>HTTPS</type>
        <adminEmail>your-email</adminEmail>
    </server>
</extension>
```

Verifying Your Plug-In Package Deployment

You can verify that your plug-in package deployed correctly by searching the log file on the vSphere Web Client Virgo server for your plug-in package ID. If the package deployed correctly, the plug-in package ID is included in a message about a successful package deployment. You can also check specific folders on the Virgo server for your expanded `.zip` files.

On the Windows operating system, the file appears in the following location.

`%PROGRAMDATA%/VMware/vSphere%20Web%20Client/vc-packages/vsphere-client-serenity/`

On the Linux operating system, the file appears in the following location.

`/var/lib/vmware/vsphere-client/vc-packages/vsphere-client-serenity/`

Unregistering a Plug-In Package

You can unregister a plug-in package that you previously registered with vCenter Server. You can unregister the plug-in package programatically, using the vSphere API `ExtensionManager` method `unregisterExtension()`, or you can use the vCenter Managed Object Browser (MOB) interface in your Web browser to manually delete the extension.

Unregistering a plug-in package on vCenter Server does not delete the plug-in package files that are installed locally on the vSphere Web Client Virgo server. The files are not used after you unregister the package. To remove the files for cleanup purposes, you must delete the plug-in package files manually.

NOTE In the current release of vSphere, any Java services you added are still active after you unregister a plug-in package, and the plug-in might still appear in the vSphere Web Client Plug-In Management view. This is a known issue, and a workaround is to restart the Virgo server.

List of Extension Points

A

Table A-1 contains the complete list of extension points published by the vSphere Web Client, the required extension definition type, and a brief description of the resulting extension.

Table A-1. vSphere Web Client Extension Points

Extension Point Name	Description
Action Extension Point	
`vise.actions.sets`	Contains a set of actions, each of which is represented by the class `com.vmware.actionsfw.ActionSpec`. Requires a data object of type `com.vmware.actionsfw.ActionSetSpec.`
Object Workspace Extension Points	
Each vSphere object type's object workspace provides a set of extension points. Each extension point corresponds to a specific data view, such as the **Summary** tab view or the **Manage** tab view. Every object workspace extension point requires a data object of type `com.vmware.ui.views.ViewSpec`. Object workspace extension points follow the format `vsphere.core.${objectType}.${view}`. The `${objectType}` placeholder corresponds to the type of vSphere object, and the `${view}` placeholder corresponds to the specific view. For example, the extension point `vsphere.core.cluster.manageViews` is the extension point for the **Manage** tab view for **Cluster** objects. The following names are valid **${objectType}** values. ■ cluster: `ClusterComputeResource` object ■ datacenter: `Datacenter` object ■ dscluster: `StoragePod` object ■ dvs: `DistributedVirtualSwitch` object ■ dvPortgroup: `DistributedVirtualPortgroup` object ■ folder: `Folder` object ■ host: `HostSystem` object ■ network: `Network` object ■ resourcePool: `ResourcePool` object ■ datastore: `Datastore` object ■ vApp: `VirtualApp` object ■ vm: `VirtualMachine` object	
`vsphere.core.${objectType}.summarySectionViews`	Portlets under the **Summary** tab view
`vsphere.core.${objectType}.monitorViews`	Second-level tabs under the **Monitor** tab view
`vsphere.core.${objectType}.monitor.performanceViews`	Views on the **Monitor** tab, under the **Performance** second-level tab
`vsphere.core.${objectType}.manageViews`	Second-level tabs under the **Manage** tab view
`vsphere.core.${objectType}.manage.settingsViews`	Views on the **Manage** tab, under the **Settings** second-level tab
`vsphere.core.${objectType}.manage.alarmDefinitionsViews`	Views on the **Manage** tab, under the **Alarm Definitions** second-level tab

Table A-1. vSphere Web Client Extension Points (Continued)

Extension Point Name	Description
Object Navigator Extension Point	
`vise.navigator.nodespecs`	Add an object collection node, category, or pointer node extension to the object navigator. Requires a data object of type `com.vmware.ui.objectnavigator.model.ObjectNavigatorNodeSpec.`
Domain Views Extension Point	
`vise.global.views`	Add a global UI view to the vSphere Web Client. Requires a data object of type `com.vmware.ui.views.GlobalViewSpec`.
Home Screen Extension Point	
`vise.home.shortcuts`	Add a shortcut to a global view or other data view to the home screen. Requires a data object of type `com.vmware.vsphere.client.views.ShortcutSpec.`
Related Objects Extension Point	
`vise.relateditems.specs`	Create a new relation between vSphere object types Requires a data object of type `com.vmware.ui.relateditems.model.ObjectRelationSetSpec.`
Object List Extension Points Each vSphere object type has a list view to which you can add columns. The vSphere Web Client SDK provides an extension point for the list view for each vSphere object type. Each extension point requires a data object of type `com.vmware.ui.lists.ColumnSetContainer.` The extension point names follow the format `vsphere.core.${objectType}.list.columns`, where the `${objectType}` placeholder corresponds to the type of vSphere object. The following names are valid `${objectType}` values. ■ cluster: `ClusterComputeResource` object ■ datacenter: `Datacenter` object ■ dscluster: `StoragePod` object ■ dvs: `DistributedVirtualSwitch` object ■ dvPortgroup: `DistributedVirtualPortgroup` object ■ folder: `Folder` object ■ host: `HostSystem` object ■ hp: `HostProfile` object ■ network: `Network` object ■ resourcePool: `ResourcePool` object ■ datastore: `Datastore` object ■ vApp: `VirtualApp` object ■ vm: `VirtualMachine` object ■ template: Virtual Machine template object	
`vsphere.core.${objectType}.list.columns`	Create a new column in the list of vSphere objects of type `${objectType}`.
Object Representation Extension Point	
`vise.inventory.representationspec`	Define one or more new icon and label sets for an object collection node in the object navigator, along with the conditions under which the icon and label sets appear. Requires a data object of type `com.vmware.ui.objectrepresentation.model.ObjectRepresentationSpec`.

Table A-2 contains the Extension Templates included with the vSphere Web Client SDK.

Table A-2. Extension Templates in the vSphere Web Client SDK

Template ID	Description
`vsphere.core.inventory.objectViewTemplate`	Creates a complete object workspace for a given custom object type. When you create an instance of the `objectViewTemplate`, the vSphere Web Client generates an extension point for each of the standard object workspace tabs, subtabs, and views. You must supply a `namespace` and an `objectType` for the `objectViewTemplate`. The `objectViewTemplate` creates extension points in the format *`namespace.extension-point-name`*. For a custom object called a Rack, for example, one extension point might be `com.myExtension.Rack.monitorViews`. For the full list of object workspace extension points, see Table A-1, "vSphere Web Client Extension Points," on page 113. A given tab does not appear in the vSphere Web Client user interface unless you explicitly create an extension that references that tab's extension point.
`vsphere.core.inventory.summaryViewTemplate`	Creates a standard **Summary** tab with an optional header at the top, and a main area that can contain one or more Portlet data views. You must supply a `namespace`, and can optionally supply a `summaryHeaderView`, for the `summaryViewTemplate`. You must create an instance of the `objectViewTemplate` before creating a `summaryViewTemplate` instance. The `summaryViewTemplate` creates the extension point *`namespace`*`.summarySectionViews`. When you create data view extensions at this extension point, they data views appear as Portlets in the **Summary** tab.
`vsphere.core.inventorylist.objectCollectionTemplate`	Creates an object collection node in the object navigator for a given custom object type. You must supply the following variables when creating an instance of the `objectCollectionTemplate`. `namespace`: a unique identifier used to avoid name clashes with other extensions `title`: the custom object text label `icon`: the custom object icon resource `objectType`: the vSphere type name for the custom object `listViewId`: the data view that appears when the user clicks the collection node, usually an object list `parentUid`: the object collection node's parent node in the object navigator control

Using Legacy Script Plug-Ins with the vSphere Web Client

B

The vSphere Web Client provides partial support for script plug-ins that you have created for previous versions of the Windows-based vSphere Client application. To use your vSphere Client script plug-ins with the vSphere Web Client, you must enable script plug-in support in the vSphere Web Client configuration. The vSphere Web Client uses the data in the XML configuration file for a given script plug-in, and renders that plug-in at an equivalent location in the vSphere Web Client user interface.

When you enable script plug-in support for the vSphere Web Client, the vSphere Web Client processes and displays any legacy script plug-in registered with your vCenter Server.

The vSphere Web Client does not fully support vSphere Client script plug-ins. To take full advantage of the vSphere Web Client extension points and tools, you must use the vSphere Web Client SDK to create plug-in modules.

This appendix contains the following topics:

Enabling Script Plug-In Support in the vSphere Web Client

Support for vSphere Client script plug-ins is disabled by default in the vSphere Web Client. To enable support for script plug-ins, locate the `webclient.properties` file in the vSphere Web Client install directory, typically `%ProgramData%\VMware\vSphere Web Client` on the Windows operating system, and `/var/lib/vmware/vsphere-client` on Linux and MacOS. Add the following line to the `webclient.properties` file.

```
scriptPlugin.enabled = true
```

You must also have registered your script plug-ins with the vCenter Server to which the vSphere Web Client is connecting. The script plug-in URL must use the HTTPS protocol to work with the vSphere Web Client. In the `extension.xml` file you use to register your script plug-in with vCenter Server, you must add the SHA1 thumbprint of the server where your `scriptConfig.xml` plug-in file is located. For an example `extension.xml` file with an SHA1 thumbprint, see "Creating the vCenter Server Extension Data Object" on page 111.

If you prefer to use a non-secure HTTP URL for your script plug-in, you can add the following flag to the `webclient.properties` file.

```
allowHttp = true
```

Where Script Plug-In Extensions Appear in the vSphere Web Client

The extensions in your vSphere Client script plug-ins appear in specific places in the vSphere Web Client. describes where each type of vSphere Client extension appears in the vSphere Web Client.

Table B-1. Script Plug-In Locations in the vSphere Web Client

Extension Type	Old Location in vSphere Client	New Location in vSphere Web Client
HomeView	Added a shortcut on the vSphere Client home screen in the **Inventory**, **Administration**, **Management**, or **Applications** category	Appears on the vSphere Web Client home screen in the **Classic Solutions** category.
InventoryView	Displayed a Web applet in the selected vSphere object's Inventory View	Appears as a subview inside the **Classic Solutions** tab for the target object's object workspace.
InventoryMenus	Added a Menu item to a specific object sub-menu under the top-level **Inventory** menu	Appears in the context menu for the target object type when the user right-clicks in the object navigator.
MainMenus	Added a menu item to one of the top-level menus in the vSphere Client, such as **File**, **Edit**, or **View**.	Appears in the user menu at the top of the vSphere Web Client, inside the **Classic Solutions** submenu.
Toolbars	Added an item to an object's toolbar	Toolbars are not supported in the vSphere Web Client.

Known Issues and Unsupported Features

Using vSphere Client script plug-ins with the vSphere Web Client has the following known issues.

- Setting the display attribute on a URL tag to a value of `"none"` runs the script located at the URL without opening the URL in a new page. This behavior is supported in the vSphere Web Client, but the Flash Player requires the domain where the script is located to have a cross-domain policy XML file. See http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.
- When using a script plug-in at a secure URL (HTTPS) in the Chrome or Firefox browsers, you must load the script plug-in page in an external tab at least once before it appears in the vSphere Web Client.
- Icons for a script plug-in shortcut do not appear in the vSphere Web Client. The vSphere Web Client displays a placeholder icon for the script plug-in icon on the home screen.
- Localized labels for script plug-ins are not supported in the vSphere Web Client. The vSphere Web Client displays all titles and labels using the English (EN) locale.
- Toolbar extensions are not supported in the vSphere Web Client.
- Script plug-ins that provide shortcuts to specific vSphere objects or object views are not supported in the vSphere Web Client, as the datacenter hierarchy is organized and displayed differently than in the vSphere Client.

Index

E

F

G

R

S

T

U

V

W

X