vSphere Web Client SDK Documentation

VMware vSphere Web Client SDK 6.5.1
VMware ESXi 6.5.1
vCenter Server 6.5.1

vmware®

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

https://docs.vmware.com/

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

docfeedback@vmware.com

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

Contents

vSphere Web Client SDK Documentation

1

The VMware vSphere® Web Client SDK is a collection of Java, Flex, and JavaScript libraries, utility tools, code samples, and documentation to help you create extensions to the vSphere Web Client and the vSphere Client .

Revision History

This *vSphere Web Client SDK Documentation* is updated with each release of the product or when necessary.

This table provides the update history of the *vSphere Web Client SDK Documentation*.

Revision	Description
13NOV2017	Improvements to SDK version compatibility information. Corrections to deprecated extension points.
11MAY2017	Initial release for vSphere 6.5 Update 1.

About vSphere Web Client SDK

The *vSphere Web Client SDK Documentation* provides information about developing extensions to the VMware vSphere® Web Client and the VMware vSphere® Client in vSphere 6.5.

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

Intended Audience

This information is intended for anyone who wants to extend the vSphere Web Client and the vSphere Client with custom functionality in vSphere 6.5. Users typically are software developers who use Flex or ActionScript, or HTML and JavaScript, to create graphical user interface components that work with VMware vSphere®.

VMware Technical Publications Glossary

VMware Technical Publications 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.

Knowledge Requirements for Using the vSphere Web Client SDK

Developing extensions for the vSphere Client and the vSphere Web Client by using the vSphere Web Client SDK, requires expertise with MXML and ActionScript, or HTML and JavaScript, and Java.

- The vSphere Web Client SDK provides a custom Model-View-Controller framework, also called Frinje framework. Understanding how the Frinje framework implements the MVC pattern is essential to developing data view extensions. The graphic elements of the View component of the MVC framework are implemented in MXML classes and the logic elements of the View component are implemented in ActionScript classes.
- The vSphere Client and the vSphere Web Client application servers provide the Virgo server that consists of a collection of Java services. These Java services communicate with vCenter Server, ESXi hosts, and other data sources. Basic understanding in Java development is required.

- You can extend the vSphere Client if you have a good understanding in Web application development by using JavaScript and HTML. You can use any user interface technology to create views for the vSphere Client UI layer. The samples provided within the SDK use the jQuery library.

vSphere Web Client SDK Contents

The vSphere Web Client SDK is a ZIP file that contains libraries for server and user interface development, API reference documentation, sample code, and SDK tools that helps you develop extensions for the vSphere Client and the vSphere Web Client applications.

After you download and unzip the SDK package, you see the following directory structure under the `vsphere-client-sdk` folder:

Table 1. SDK Contents

SDK Folder			Description
`/flex-client-sdk`	`/docs`		Contains the API reference documentation for the Java and Flex libraries.
	`/libs`		Contains the Flex and Java libraries for building UI and server plug-ins, and common libraries that are used to build the sample plug-ins.
	`/resources`		Contains Ant build scripts and configuration files.
	`/samples`		Contains the source code and project files for the Flex-based samples provided with the SDK.
	`/tools`	`/Eclipse plugin site`	Contains the Eclipse plug-in with various tools to ease the development of vSphere Web Client extensions.
		`/Plugin generation scripts`	Contains scripts that you can use to create Flex and Java plug-in template projects.
		`/vCenter registration scripts`	Contains the scripts that are used to generate the files the connect the local vSphere Web Client in your development environment to the remote vCenter Server instance.
	`/vsphere-client`	`/plugin-packages`	Contains the core plug-in packages for the vSphere Web Client application.
		`/server`	Contains the local vSphere Web Client application which you use to deploy your plug-in packages.
	`vsphere-client-sdk-version.xml`		Contains information about the version and the build number of the vSphere Web Client SDK.
`/html-client-sdk`	`/docs`		Contains the Java API reference documentation that you can use to create your service layer extensions.
	`/libs`		Contains common libraries that are used to build the sample plug-ins.
	`/resources`		Contains Ant build scripts, configuration files, and minimal Adobe Flex 4.6 SDK for building `.swf` files out of string resources.

Table 1. SDK Contents (Continued)

SDK Folder			Description
	/samples		Contains the source code and project files for the HTML samples provided with the SDK.
	/tools		Contains the same tools as the ones in the /flex-slient-sdk folder that can help you create and build your HTML extensions.
	/vsphere-ui	/plugin-packages	Contains core plug-in packages for the vSphere Client application.
		/server	Contains the runtime server of the local vSphere Client which you can use to deploy your HTML plug-ins.
	html-client-sdk-version.xml		Contains information about the version and the build number of the vSphere Client development kit.
License-Agreement.html			The license agreement for the SDK.

SDK Versions and Compatibility

When you upgrade from an older version of the vSphere Web Client SDK, you must consider whether your plug-ins will be compatible with the new vSphere Client and vSphere Web Client.

You can refer to the following tables for more information about the compatibility of the plug-ins you developed with the different versions of the vSphere Web Client SDK.

Table 2. Compatibility Between the HTML Plug-In Created with a Specific Version of the SDK and the Different Web Browser Applications

Version of the SDK That Is Used to Create the HTML Plug-In	vSphere Web Client 5.5	vSphere Web Client 6.0	vSphere Web Client 6.5 and vSphere Client 6.5
version 5.5	Yes	Yes*	Yes*
version 6.0	No	Yes	Yes**
version 6.5	No	Yes	Yes

Note * If you have HTML-based plug-ins that are created with the vSphere Web Client SDK 5.5, you must follow the instructions for upgrading your plug-in to ensure compatibility with the vSphere Web Client versions 6.0 and 6.5, and the vSphere Client 6.5. For more information, see Upgrading Plug-Ins Created with the vSphere Web Client SDK 5.5.x.

Note ** If you have HTML-based plug-ins that are created with the vSphere Web Client SDK 6.0, you must follow the steps for upgrading your plug-in to ensure compatibility with the 6.5 versions of the vSphere Web Client and the vSphere Client . For more information, see Upgrading Plug-Ins Created with the vSphere Web Client SDK 6.0, Upgrading Plug-Ins Created with the vSphere Web Client SDK 6.0 Update 1, and Upgrading Plug-Ins Created with the vSphere Web Client SDK 6.0 Update 2.

Table 3. Compatibility Between the Flex-Based Plug-In Created with a Specific Version of the SDK and the vSphere Web Client

Version of the vSphere Web Client SDK That Is Used to Create the Flex-Based Plug-In	vSphere Web Client 5.5	vSphere Web Client 6.0	vSphere Web Client 6.5
version 5.5	Yes	Yes	Yes*
version 6.0	No	Yes	Yes*
version 6.5	No	Yes	Yes*

Note * To make the plug-ins that are created with the older versions of the vSphere Web Client SDK compatible with the vSphere Web Client 6.5, make sure that you update the plug-ins according to the release notes provided within the SDKs.

Naming Convention for vSphere Objects in the vSphere Web Client SDK

The vSphere Web Client UI uses user-friendly names for the default types of vSphere managed objects. However, the service APIs, extension points, and libraries in the SDK refer to vSphere objects by using the naming convention established by the vSphere API.

Table 4. vSphere Entity Names in the User Interface and SDK Code

vSphere Entity Name in the vSphere Web Client User Interface	Entity Name in the SDK Code
Cluster	`ClusterComputeResource`
Data Center	`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`

About the vSphere Web Client and the vSphere Client 2

VMware vSphere® Web Client and the VMware vSphere® Client provide means for connecting to VMware vCenter Server® systems and managing the objects in the vSphere 6.5 infrastructure.

Starting with vSphere 6.5, VMware provides the vSphere Client that is an HTML5 Web browser-based application which you can use to connect to vCenter Server systems and manage vSphere objects.

This section includes the following topics:

- Introduction to the vSphere Web Client
- Introduction to the vSphere Client
- Understanding Extensibility in the vSphere Client and the vSphere Web Client

Introduction to the vSphere Web Client

The VMware vSphere® Web Client is a Web browser-based application that you can use to connect to vCenter Server systems and manage your vSphere infrastructure.

You can use the vSphere Web Client to monitor and modify the objects in your vSphere inventory.

You can extend your vSphere infrastructure in different ways to create a solution for your unique use case. 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.

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 the data in a Web browser.

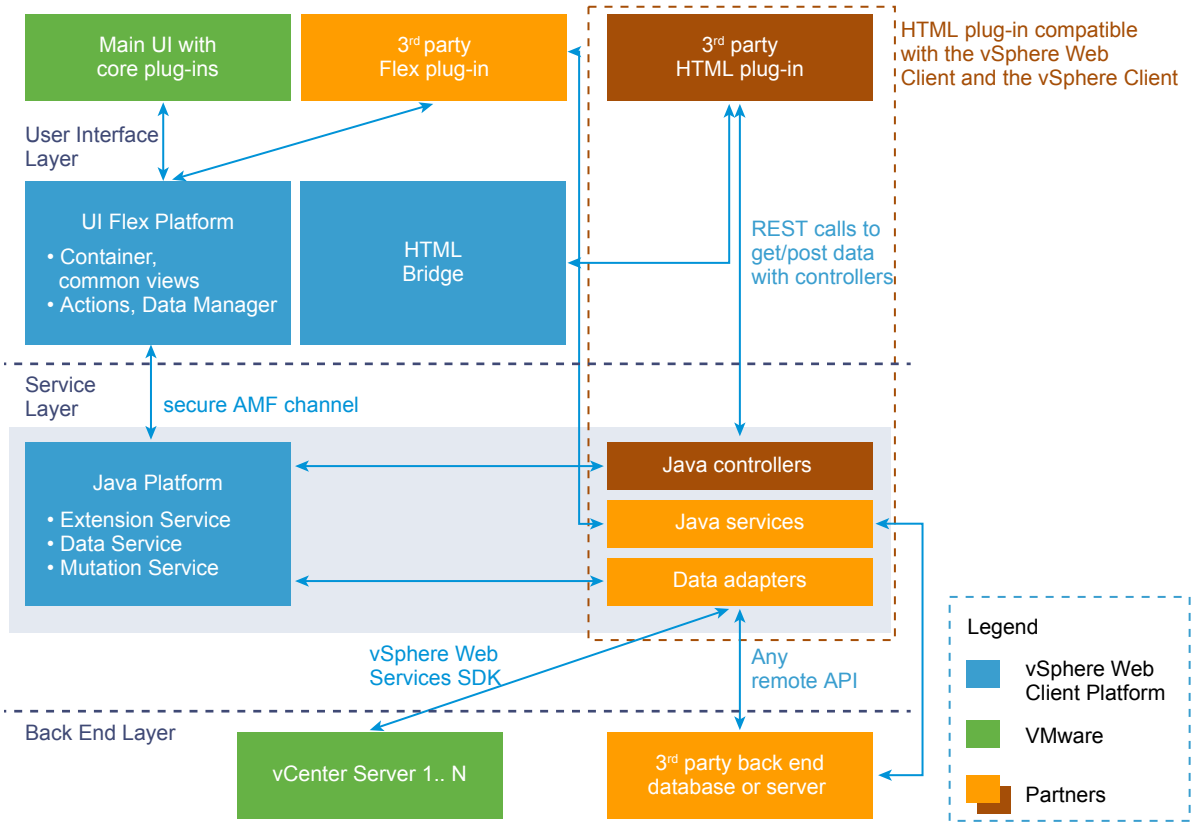
User Interface Layer The user interface layer consists of an Adobe Flex application that is displayed in a Web browser. The Flex application contains all of the user interface elements with which you interact, such as menus, navigation elements, data portlets, and commands. You can navigate through the 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. The Java services communicate with VMware 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 by the Flex user interface layer. When you perform an action from the Flex user interface, such as a management or administration command, the Java services perform that command on the virtual infrastructure.

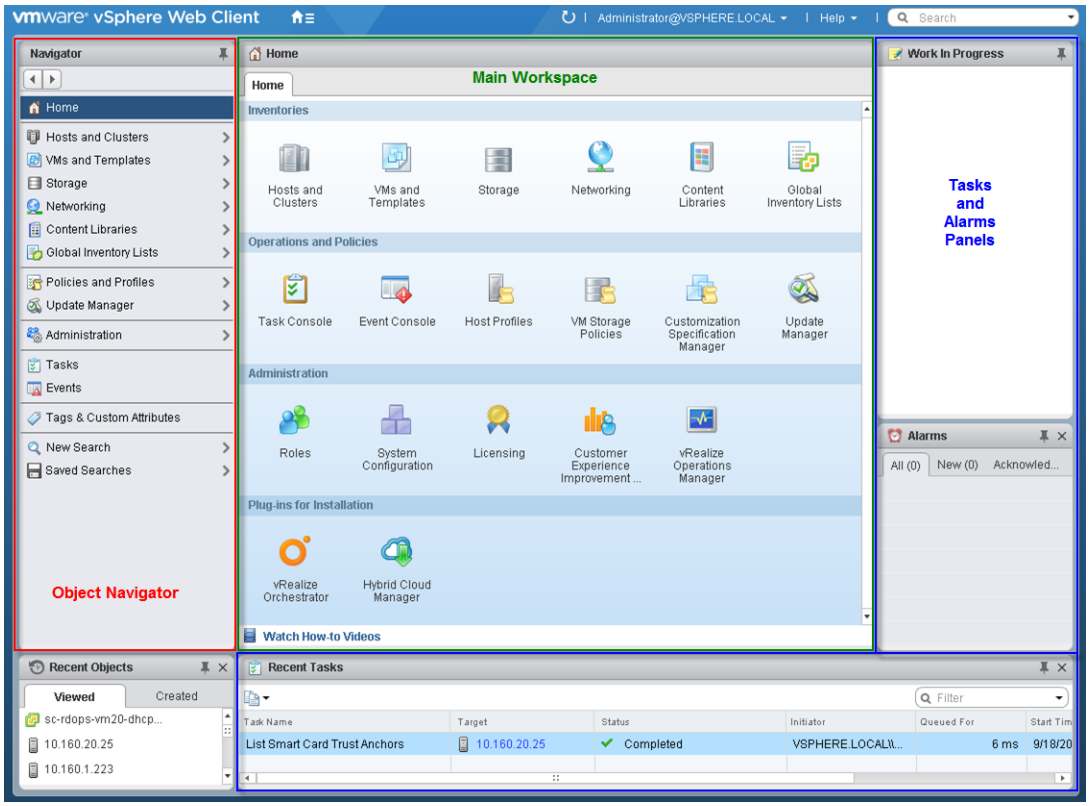
The vSphere Web Client application server contains a Spring framework that manages the communication between the user interface layer and the service layer.



Overview of the vSphere Web Client User Interface Layer Components

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.

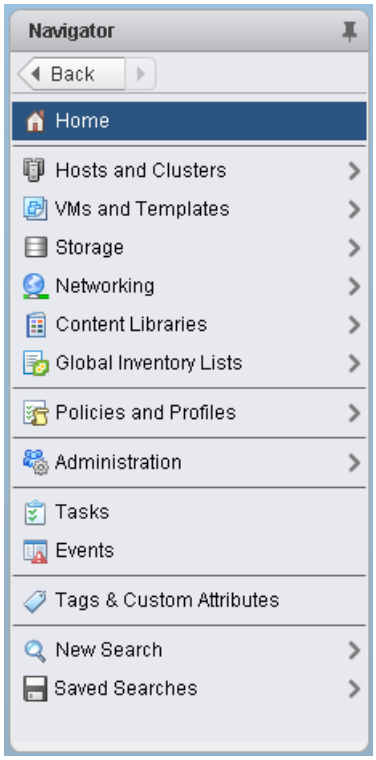


Object Navigator

You can use the object navigator to browse and select objects in the virtual infrastructure and to access other solutions and data views in the vSphere Web Client. When you select an object in the object navigator, the contents of the vSphere Web Client main workspace changes to display additional information about the selected object.

Object Navigator Top Level

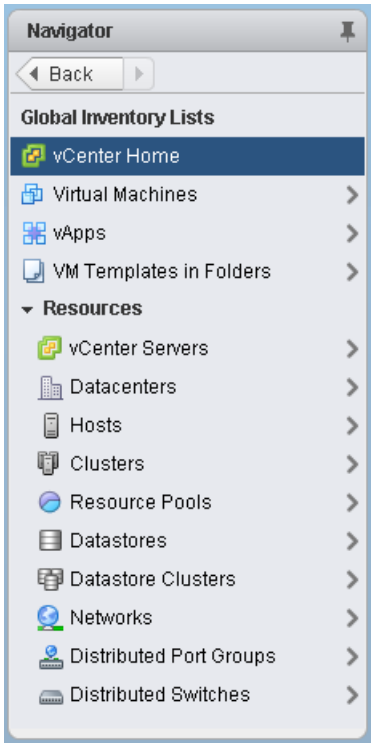
The top level of the object navigator contains links to the major features and solutions in the vSphere Web Client, including the vCenter Inventory Lists, the inventory tree, Policies and Profiles, and Administration applications. You can extend the object navigator top level with links to new solutions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



Object Navigator When Browsing the Virtual Infrastructure

When you browse the virtual infrastructure, the object navigator is the primary means of interacting with the vSphere objects in the data center. The object navigator vCenter Home level presents the vSphere objects in a graph-based view. You can navigate from an object to its related objects in the inventory, regardless of their type. When you select 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 Home level with new inventory lists or custom object lists. You can also add links to other solutions to the vCenter Home level.



Main Workspace

The vSphere Web Client displays the home screen, solutions and applications, and information about the virtual infrastructure in the main workspace.

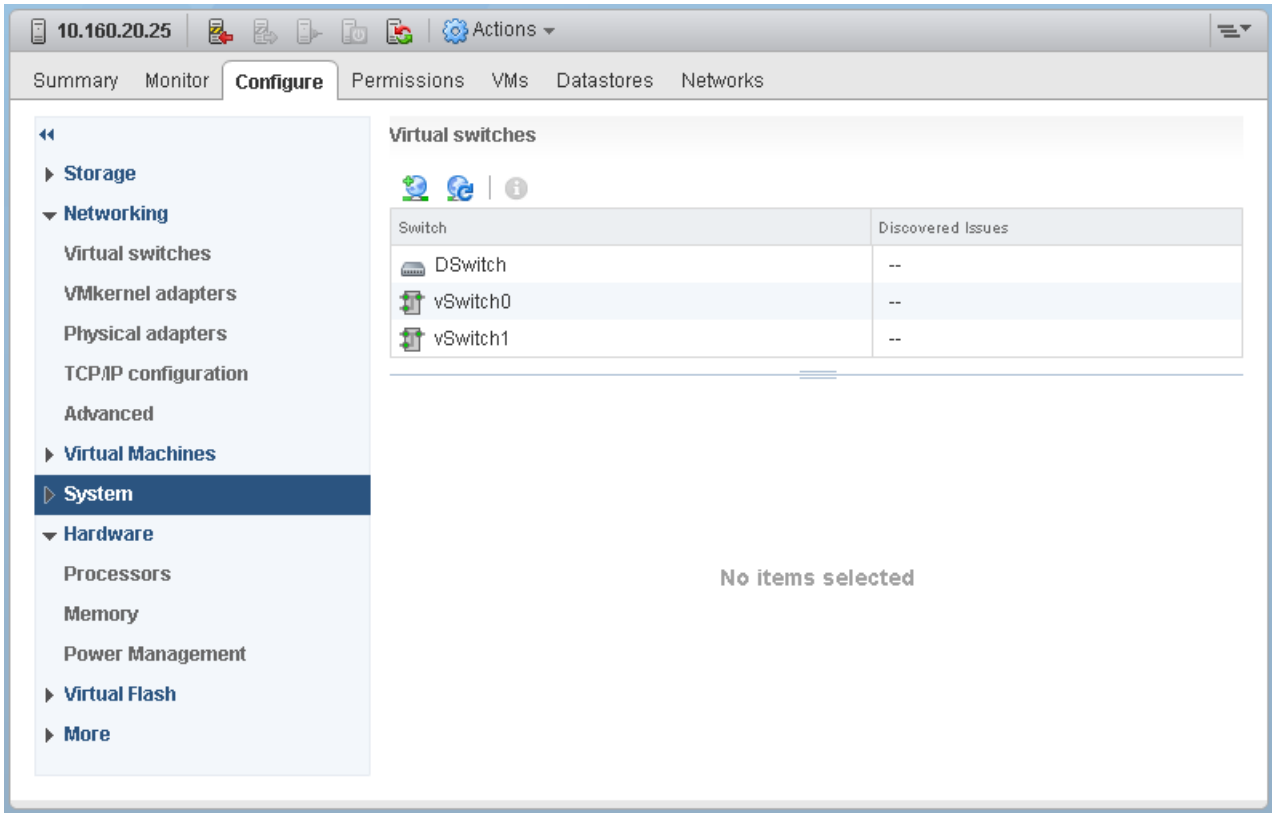
The main workspace is the center of the vSphere Web Client graphical user interface and contains data views, navigation elements such as tabs and toolbars or context menus for user actions.

Home Screen

The home screen is the initial view shown in the main workspace when you log into 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 you browse the virtual infrastructure by 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**, **Configure**, **Permissions**, and categorized relations top-level tab screens. Some of these tabs contain table of contents elements and views within these elements.



Some of the nested data views are contextual. For example, the **Monitor** tab always contains a second-level **Issues** tab, and contain also 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 the following top-level tabs.

Table 2-1. Top-Level Tabs for vSphere Objects

Tab	Description
Getting Started	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	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 features. You can use the **Summary** tab to view information about the specific object so that you can understand the role of the object in the virtual infrastructure.

Table 2-1. Top-Level Tabs for vSphere Objects (Continued)

Tab	Description
Monitor	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 you might respond. The **Monitor** tab contains data views that show information about the health status, performance statistics, event logs, issues, and alarms that are related to the object.
Configure	The **Configure** tab displays settings and configurations that determine how the selected object behaves. Using the **Configure** tab, you can perform operations on a vSphere object, such as provisioning or maintenance. You can also change object settings and issue management commands from the **Configure** tab.
Permissions	The **Permissions** tab displays the permissions that are assigned to the logged-in user.
Categorized Tabs for Object Relations	Each categorized tab shows the vSphere objects related to the currently selected object. For example, if you select a cluster from the object navigator, the categorized tabs that you see are **Hosts**, **VMs**, **Datastores**, and **Networks**. Users can select a related object directly from the first-level tabs, and view the workspace for that related object. Relations to custom vSphere objects are displayed as second-level tabs under the **More Objects** first-level tab.

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.

Workspace 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**, **Configure**, **Permissions**, and the categorized tabs for object relations.

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 you select the **Administration** application, the object navigator displays different categories of services such as **Access Control**, **Single Sign-On**, **Licensing**, **Solutions**, **Deployment**, and **Customer Experience Improvement Program**. Information about specific services appears in the main workspace. The vSphere Web Client and additional plug-in modules appear as services in the **Administration** navigation interface.

Introduction to the vSphere Client

The VMware vSphere® Client is an HTML5-based Web application that you can use to connect to vCenter Server systems and manage your vSphere infrastructure.

Use the vSphere Client in the same way as the Flex-based vSphere Web Client. Your existing HTML-based plug-ins that are created for the HTML Bridge need some adjustments before they can work property in the vSphere Client . For further information about how to create an HTML-based plug-in that is compatible with both Web browser applications, see Hybrid Plug-Ins.

Understanding the vSphere Client Architecture

The vSphere Client architecture consists of three layers: the user interface layer, the Java service layer, and the back end layer.

The user interface and the Java service layers have different roles in communicating with the back end layer for retrieving data, and presenting the data in a Web browser.

User Interface Layer The user interface layer consists of an HTML platform that provides a plug-in architecture for the extensions displayed in a Web browser. The HTML application contains all user interface elements with which the user interacts, such as menus, commands, home screen shortcuts, and other views. You can use the user interface elements to view information about an object in the vSphere environment, send a command, and make changes to your vSphere infrastructure.

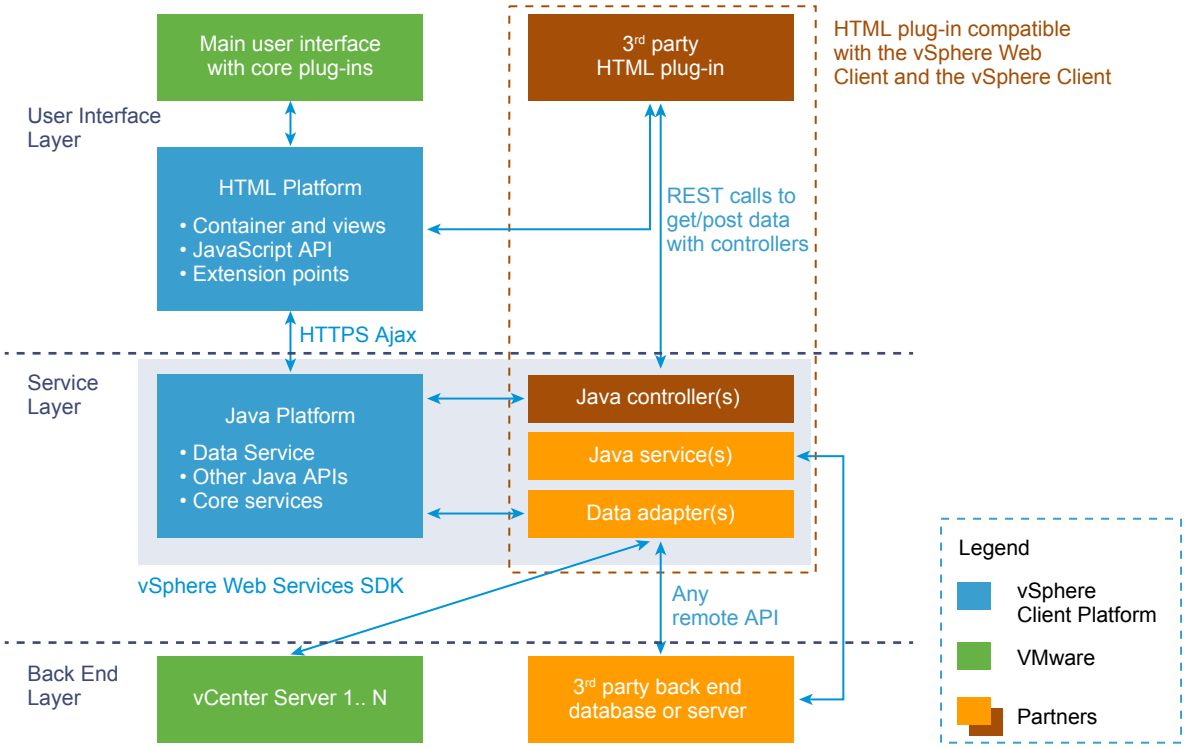
The HTML platform ensures that each plug-in view is isolated from the vSphere Client application which allows you to use the UI technology of your choice when developing HTML plug-ins. You can also use any library to implement the UI components within your views.

You can use the same extension points for your HTML plug-ins as the ones that you use for the Flex-based plug-ins.

Java Service Layer

The Java service layer remains the same as the one in the vSphere Web Client application.

You can use the Spring MVC framework and the Ajax Web development techniques to establish communication between the user interface layer and the service layer.

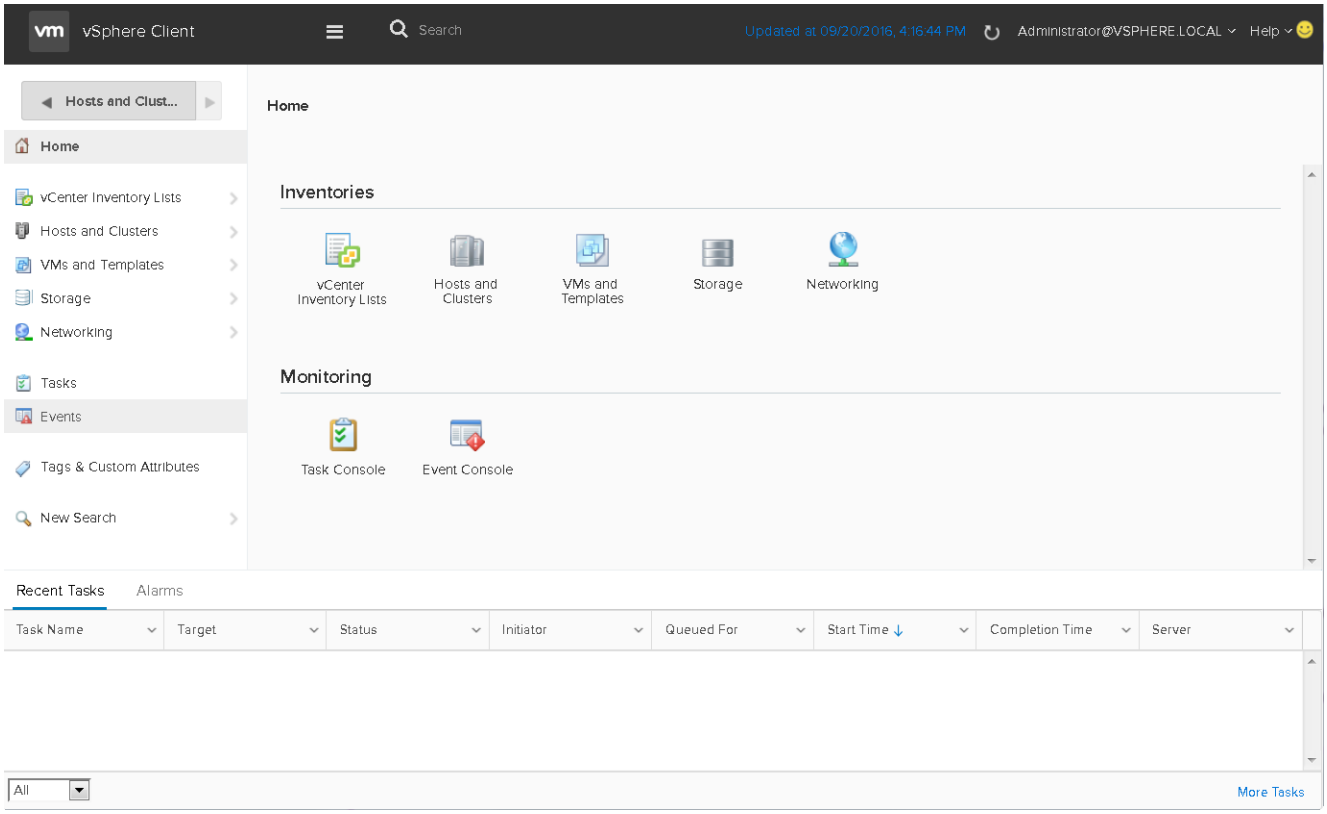


Overview of the User Interface Layer Components

The user interface layer of the vSphere Client 6.5 contains a limited set of the views and the features that are provided by the vSphere Web Client for managing vSphere objects.

The user interface layer of the vSphere Client contains HTML views, such as the data views, portlets, navigation options, and search bar. The vSphere Client provides a vSphere objects navigator, the same top-level tabs for the vSphere objects in the main workspace area, and a panel that displays the recent tasks and events.

You navigate through the user interface of the vSphere Client application in the same way as you do with the vSphere Web Client.



Understanding Extensibility in the vSphere Client and the vSphere Web Client

You extend the vSphere Client and 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 Client or 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 Client and the vSphere Web Client graphical user interfaces.

In general, you extend the vSphere Client and 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 Client and 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 Client and 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 Client and the vSphere Web Client can involve creating both user interface plug-in modules and service plug-in modules. For more information about the architectures of both clients, see Understanding the vSphere Client Architecture and Understanding the vSphere Web Client Architecture.

- Extending the User Interface Layer

  A user interface plug-in module adds one or more extensions to the vSphere Client and the vSphere Web Client user interface layer.

- Extending the Java Service Layer

  You can add new Java services to the service layer. The Java services you add can perform any of the functions of a typical Java Web service.

Extending the User Interface Layer

A user interface plug-in module adds one or more extensions to the vSphere Client and 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 Client and the vSphere Web Client service layers. You can use the libraries included with the vSphere Web Client SDK to enable communication between your GUI objects and the service layer or if you create an HTML plug-in, you can use a library of your choice.

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.

Extending the Java Service Layer

You can add new Java services to the service layer. The Java services you add can perform any of the functions of a typical Java Web service.

The Java services you add to the Java service layer are used to retrieve data from the vSphere environment and display the data 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 Client and the vSphere Web Client application servers, 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 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 Client or the vSphere Web Client services do not already provide, you must extend the Data Service to provide such 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 Client or the vSphere Web Client user interfaces. 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.

vSphere Web Client SDK Installation and Setup Guide

3

To develop HTML and Flex-based custom plug-ins for the vSphere Web Client and the vSphere Client, you must first set up your development environment.

This section includes the following topics:

Software Requirements

You can set up your development environment for developing HTML-based and Flex-based extensions by using specific software components.

To set up your development environment, you can use the following software components with their respective versions.

Software Component	Minimum Required Version	Description
Java Standard Edition Development Kit (JDK)	1.8.x	For information about the required setup for Java development, see Set Up for Java Development. The local Virgo server runtime requires JDK 1.8.x to work with the vCenter Server 6.5 instance.
Apache Ant	1.9.x	For more information about how to use Ant to automate the build process of your plug-ins, see Automate the Plug-In Build Process.
Eclipse IDE for Java EE Developers or Spring Tool Suite	Eclipse 4.2 (Juno) and Eclipse 4.3 (Kepler), or Spring Tool Suite 2.9 and later	For more information about how to set up the Eclipse IDE, see (Optional) Set Up the Eclipse Integrated Development Environment. **Note** For developing HTML plug-ins you can use Eclipse Neon.
vSphere Web Client SDK Tools	Use the version from the downloaded vSphere Web Client SDK	For more information about how to install the vSphere Web Client SDK Tool, see Install the vSphere Web Client Tools Eclipse Plug-In. The plug-in works on Widows and Mac OS setups.
Adobe Flash Builder	4.7	The Adobe Flash Builder is required only for developing Flex-based plug-ins.
Adobe Flex SDK	4.6	For more information about how to set up the Adobe Flex SDK, see Set Up the Adobe Flex 4.6 Software Development Kit.

Software Component	Minimum Required Version	Description
Flash Player and Flash Player Content Debugger	11.5 and later	You can install Flash Player and Flash Player Content debugger in the browser that you use.
IntelliJ IDEA	Standard Edition	You can use the IntelliJ IDEA as an alternative to the Eclipse IDE for developing your Java, JavaScript, and Flex code. The Standard Edition also provides Flex debugging utilities.

Development Environment Requirements Overview

Before you start setting up your development environment, you must download the vSphere Web Client SDK to your working machine and have access to a vCenter Server for Windows or a vCenter Server Appliance instance.

To create a vSphere Web Client extension, your development environment must include the following items.

- A development environment capable of developing Web applications by using ActionScript and MXML, or JavaScript and HTML. You can use the Eclipse Integrated Development Environment or Spring Tools Suite. The vSphere Web Client SDK includes tools and plug-ins for the both IDEs to aid you in creating vSphere Web Client plug-ins.
- A development environment capable of developing Java-based Web applications. You can again use the Eclipse IDE or the Spring Tools Suite. The vSphere Web Client contains tools and plug-ins for both IDEs to aid you in creating services compatible with the vSphere Web Client Virgo server framework.
- Access to avCenter Server for Windows or a vCenter Server Appliance instance to register your extension and in this way allow the vSphere Web Client to download and install the plug-in.

You can set up the vSphere Web Client SDK on a machine with Windows or Mac OS operating systems. Before you begin the SDK setup, you can set up your Java environment and Apache Ant, install and configure the Eclipse IDE and the Adobe Flex SDK.

Setting Up for HTML-Based Plug-In Development

The vSphere Web Client SDK contains libraries, sample plug-ins, and various SDK tools that help you develop and build extensions to the vSphere Web Client and the vSphere Client.

In vSphere 6.5, system administrators can use the vSphere Client to manage their vSphere infrastructures. The vSphere Client development kit is added to the vSphere Web Client SDK allowing developers to create custom HTML plug-ins and hybrid plug-ins for both Web browser-based applications.

Setting up your development environment for creating HTML and hybrid plug-ins for the vSphere Web Client involves several tasks.

1 Download the vSphere Web Client SDK

   Download the `.zip` file that contains all components of the vSphere Web Client SDK.

2 Set Up for Java Development

You must set up your Java development environment to create extensions to the Service Layer.

3 Automate the Plug-In Build Process

Apache Ant is used by the scripts in the SDK to generate plug-in project templates and to build plug-ins.

4 (Optional) Set Up the Eclipse Integrated Development Environment

You can use an IDE of your choice to develop custom plug-ins for the vSphere Client and the vSphere Web Client. The SDK provides a plug-in for Eclipse to ease your development process which suggests that you have installed the Eclipse IDE on your machine.

5 Install the vSphere Web Client Tools Eclipse Plug-In

The vSphere Web Client SDK provides an Eclipse plug-in that adds tools and wizards to your Eclipse IDE to ease your HTML and Flex-based plug-in development process. This step is required only if you use the Eclipse IDE for developing HTML plug-ins.

6 Register Your Local vSphere Client with the vCenter Server Instance

If you want to verify your custom plug-ins, you can deploy the plug-ins first on your local vSphere Client and vSphere Web Client. You must register your local instances of the Web browser applications with the vCenter Server Appliance or vCenter Server for Windows to be able to deploy your plug-ins locally.

7 Configure the Virgo Server in Your Eclipse IDE

You can use the Virgo server from the Eclipse IDE in your development environment to test easily your HTML plug-ins. This step is required only if you use the Eclipse IDE for developing HTML plug-ins.

8 Set Up the Adobe Flex 4.6 Software Development Kit

To compile the resources of your HTML plug-ins, you must install the Adobe Flex 4.6 SDK on your development machine.

Download the vSphere Web Client SDK

Download the `.zip` file that contains all components of the vSphere Web Client SDK.

Prerequisites

Create a My VMware account at https://my.vmware.com/web/vmware/.

Procedure

1 Download the vSphere Web Client SDK from the VMware Web site at https://my.vmware.com/web/vmware/downloads.

The vSphere Web Client SDK is part of the VMware vCloud Suite and VMware vSphere, listed under Datacenter & Cloud Infrastructure.

2 Conform the mid5sum is correct.

   See the VMware Web site topic Using MD5 Checksums at http://www.vmware.com/download/md5.html.

3 Extract the content of the SDK in a directory on your development machine.

   Note The name of the directory where you extract the vSphere Web Client SDK must be short and without spaces.

What to do next

Open the `README.html` file and review the information about the other files and directories in the vSphere Web Client SDK.

Set Up for Java Development

You must set up your Java development environment to create extensions to the Service Layer.

You might already have the Java platform installed on your development machine. To check the version of your Java installation, open a command prompt and enter `java –version`.

Procedure

1 From the Oracle Web site at http://www.oracle.com/technetwork/java/javase/downloads/index.html, download the Java SE Development Kit installer.

   - For developing Flex-based plug-ins, download JDK1.7.0_17.
   - For developing HTML plug-ins, download JDK 1.8.x.

   Download the 64-bit version of the JDK installer if you need to allocate more memory.

2 Install the JDK following the instructions of Oracle for the operating system of your development machine.

3 Specify the location to the JDK.

Operating System	Java Location
Windows	Use the `JAVA_HOME` environment variable to specify the location of the JDK. For example, set the environment variable to `C:\Program Files\Java\jdk1.7.0_17`.
Mac OS	Open the Terminal application and enter the following command: `echo export "JAVA_HOME=\$(/usr/libexec/java_home)" >> ~/.bash_profile`. In case you have more than one Java Development Kits installed, you can specify only the version you want by using the following command: `echo export "JAVA_HOME=\$(/usr/libexec/java_home –v 1.7.0_17)" >> ~/.bash_profile`.

What to do next

Set the Java compiler compliance level to Java 1.7 in your automation build scripts or in Eclipse.

Automate the Plug-In Build Process

Apache Ant is used by the scripts in the SDK to generate plug-in project templates and to build plug-ins.

You can setup Apache Ant in your development environment to generate plug-in project templates and build plug-ins out of the projects. You can build also the samples provided with the vSphere Web Client SDK and the vSphere Client development kit.

To use the SDK build scripts inside Eclipse, you can use the Apache Ant version provided with the Eclipse package. The following procedure setups Apache Ant for running scripts by using the command line console.

Prerequisites

Verify that you have a Java environment installed on your development machine. See Set Up for Java Development.

Procedure

1. From the Apache Ant site at http://ant.apache.org/bindownload.cgi, download the Apache Ant binary distribution.
   - For developing Flex-based plug-ins, download Apache Ant 1. 7.1 or later.
   - For developing HTML plug-ins, download Apache Ant 1.9.x.
2. Install Apache Ant by following the provided instructions for the operating system of your development machine.
3. Set the `ANT_HOME` environment variable to the directory on your development machine where you installed Apache Ant.
4. Set the `VSPHERE_SDK_HOME` environment variable to the directory on your development machine where you extracted the vSphere Web Client SDK.
   - For developing Flex-based plug-ins, set the environment variable to the `flex-client-sdk` directory on your development machine.
   - For developing HTML plug-ins, set the environment variable to the `html-client-sdk` directory on your development machine.

(Optional) Set Up the Eclipse Integrated Development Environment

You can use an IDE of your choice to develop custom plug-ins for the vSphere Client and the vSphere Web Client. The SDK provides a plug-in for Eclipse to ease your development process which suggests that you have installed the Eclipse IDE on your machine.

Procedure

1. From the Eclipse Web site at http://www.eclipse.org/downloads/eclipse-packages/, download the Eclipse IDE for Java EE Developers package.
   - For developing Flex-based plug-ins, download Eclipse Juno or Eclipse Kepler.
   - For developing HTML plug-ins, download Eclipse Neon.
2. Extract the contents of the downloaded file into an appropriate location on your development machine.
3. If you do not have the minimum and maximum heap size automatically set up for Eclipse, edit the `eclipse.ini` file before you start Eclipse. You must add the location to the JDK you installed and to increase the heap space and the maximum permanent space used by the VM.

   You must add or edit the Eclipse initialization file to have the following lines:

   ```
   -vm
   C:/<your JAVA_HOME directory>/bin/java.exe
   -Xmx1024m
   -XX:MaxPermSize=512m
   ```

4. Start Eclipse and edit the Eclipse preferences to set up your workspace for developing plug-ins for the vSphere Client and the vSphere Web Client.
   a. Go to **Window > Preferences**.

      The Preferences dialog opens.

   b. From the General page, select the **Show heap status** option to display information about the current Java heap usage.

   c. From **General > Network Connections**, configure the proxy settings to be used when opening a connection.

   d. From **General > Workspace**, select the **Build automatically** and **Refresh using native hooks or polling** check boxes.

   e. From **Java > Code Style > Formatter**, configure your code and naming conventions.

   f. From **Java > Installed JREs**, add the location to the JDK you installed. See Set Up for Java Development.

g From **General > Workspace > Linked Resources**, set the location to your SDK.

- For developing Flex-based plug-ins, set the path to the `flex-client-sdk` folder as a value of the `VSPHERE_CLIENT_SDK` path variable.
- For developing HTML plug-ins, set the path to the `html-client-sdk` folder as a value of the `VSPHERE_CLIENT_SDK` path variable.

h From **Java > Build Path > Classpath Variables**, set the location to your SDK.

- For developing Flex-based plug-ins, set the path to the `flex-client-sdk` folder as a value of the `VSPHERE_CLIENT_SDK` classpath variable.
- For developing HTML plug-ins, set the path to the `html-client-sdk` folder as a value of the `VSPHERE_CLIENT_SDK` classpath variable.

What to do next

After you install and set up the Eclipse IDE on your development machine, you can install the vSphere Web Client SDK Tools Eclipse plug-in.

Install the vSphere Web Client Tools Eclipse Plug-In

The vSphere Web Client SDK provides an Eclipse plug-in that adds tools and wizards to your Eclipse IDE to ease your HTML and Flex-based plug-in development process. This step is required only if you use the Eclipse IDE for developing HTML plug-ins.

Prerequisites

- Configure the proxy to be used for your development machine. For more information, see (Optional) Set Up the Eclipse Integrated Development Environment.

Procedure

1 Start Eclipse on your development machine.

2 From **Help > Install New Software ...**, click **Add** in the Install dialog box.

The Add Repository dialog box appears.

3 In the Name text box, enter a name for this local site, such as `vSphere Client plug-in site`.

You can reuse the created repository, if you install a new version of Eclipse at the same place on your machine.

4 Click **Local ...** and browse to the *`your_SDK_location`*`/tools/Eclipse plugin site` directory.

5 Select the vSphere Web Client SDK Tools node from the discovered software and click **Next**.

6 Check the installation details and accept the license agreement.

7 Click **Finish** to complete the wizard.

Click **OK** on the warning pop-up dialog box that shows up during the installation process.

8 Restart your Eclipse SDK to apply the changes.

What to do next

Verify that the Eclipse plug-in is installed correctly by going to **Help > About Eclipse** and selecting **Installation Details**.

Register Your Local vSphere Client with the vCenter Server Instance

If you want to verify your custom plug-ins, you can deploy the plug-ins first on your local vSphere Client and vSphere Web Client. You must register your local instances of the Web browser applications with the vCenter Server Appliance or vCenter Server for Windows to be able to deploy your plug-ins locally.

The SDK provides a registration script that you can run in the vCenter Server instance. The files generated by this script connect your local Web browser application to the remote vCenter Server system.

Prerequisites

- Verify that you have access to a vCenter Server instance.

Procedure

1. Navigate to the `vCenter registration scripts` folder under `tools` in your SDK installation.
2. Copy the `dev-setup` script to one of the following locations on the vCenter Server system depending on your vSphere deployment.
   - On the vCenter Server Appliance, use the `root` directory to copy the script. You must make the file executable.
   - On the vCenter Server for Windows, use the `C:\Users\Administrator` directory to copy the script.
3. Run the `dev-setup` script in the corresponding directory.

   The script generates the following files: `webclient.properties`, `store.jks`, and `ds.properties`.
4. Copy the generated files on your development machine in one of the following locations.

   Note On a Windows operating system, you might not be able to see the `ProgramData` folder. To change the way items are displayed on a Windows machine, use **Folder Options** from **Control Panel**.

Operating System	Generated File	Location on Your Development Machine
Windows	`webclient.properties`	`C:\ProgramData\VMware\vCenterServer\cfg\vsphere-client\`
Mac OS		`/var/lib/vmware/vsphere-client/vsphere-client/`
Windows	`store.jks`	`C:\ProgramData\VMware\vCenterServer\cfg\`
Mac OS		`/var/lib/vmware/vsphere-client/`

Operating System	Generated File	Location on Your Development Machine
Windows	ds.properties	C:\ProgramData\VMware\vCenterServer\cfg\vsphere-client\config\
Mac OS		/var/lib/vmware/vsphere-client/vsphere-client/config/

5 Set up the VMWARE_CFG_DIR environment variable on your local machine to point to one of the following directories:

- For a Windows development environment, set C:\ProgramData\VMware\vCenterServer\cfg\ as a value to the variable.
- For a Mac OS development environment, set /var/lib/vmware/vsphere-client or /var/lib/vmware/vsphere-ui as a value to the variable.

6 If you use a Mac OS development environment, edit the following properties to point to the /var/lib/vmware/vsphere-client/store.jks file.

a In the webclient.properties file, edit the keystore.jks.path property.

b In the ds.properties file, edit the solutionUser.keyStorePath property.

7 If you use a Mac OS development environment, before you connect your local Web browser application to the vCenter Server system for the first time, edit the tomcat-server.xml file. Add the /var/lib/vmware/vsphere-client/store.jks value to the keystoreFile.

You can locate the file at *your_SDK_folder*/vsphere-client-sdk/html-client-sdk/vsphere-ui/server/configuration or *your_SDK_folder*/vsphere-client-sdk/flex-client-sdk/vsphere-client/server/configuration depending on the Web browser application.

8 Start the local vSphere Client or vSphere Web Client by running the startup script located at the bin directory of the server folder.

For example, if you use a Mac OS development environment, the script for starting the vSphere Client is located at *your_SDK_folder*\vsphere-client-sdk\html-client-sdk\vsphere-ui\server\bin.

9 Open a Web browser and log into your local vSphere Client at https://localhost:9443/ui or into your local vSphere Web Client at https://localhost:9443/vsphere-client.

Your local vSphere Client or vSphere Web Client connects to the vCenter Server instance and displays the vSphere inventory.

What to do next

You can deploy your custom plug-ins to the local vSphere Client or vSphere Web Client and verify whether the plug-ins function properly in your development environment before deploying them on the remote Web browser applications.

Configure the Virgo Server in Your Eclipse IDE

You can use the Virgo server from the Eclipse IDE in your development environment to test easily your HTML plug-ins. This step is required only if you use the Eclipse IDE for developing HTML plug-ins.

Prerequisites

- Configure the proxy to be used for your development machine. For more information, see (Optional) Set Up the Eclipse Integrated Development Environment.
- Register your local vSphere Client with a vCenter Server instance. See Register Your Local vSphere Client with the vCenter Server Instance.

Procedure

1. Start Eclipse on your development machine.
2. From **Help > Install New Software ...**, click **Add** in the Install dialog box.

   The Add Repository dialog box appears.
3. In the Name text box, enter a name for the Virgo server tools and in the Location text box, enter `http://download.eclipse.org/virgo/snapshot/tooling`.
4. From the displayed software, select **Eclipse Virgo Tools** and click **Next**.
5. Check the installation details and accept the license agreement.
6. Click **Finish** to complete the wizard.

   Click **OK** on the warning pop-up dialog box that shows up during the installation process.
7. Restart your Eclipse SDK to apply the changes.
8. When Eclipse starts again, go to **Window > Show View > Servers**, right-click in the Servers view, and select **New > Server**.
9. In the New Server wizard, select **EclipseRT > Virgo Runtime** and click **Next**.
10. Use the **Browse** button to navigate to the `your_SDK_folder\vsphere-client-sdk\html-client-sdk\vsphere-ui\server` directory on your development machine.
11. Click **Finish** to complete the Virgo server creation.

What to do next

Before you start the local Virgo server application, you must register the server with a vCenter Server instance. See Register Your Local vSphere Client with the vCenter Server Instance.

You must reconfigure also the following settings:

- Set the `VMWARE_CFG_DIR` environmental variable in the **Edit Configuration** dialog box that opens when you select **Open launch configuration** from the **General Information** pane.
  - In a Mac OS development environment, click the **Environment** tab and set the `/var/lib/vmware/vsphere-client` value to the variable.

- In a Windows development environment, click the **Environment** tab and set the `C:/ProgramData/VMware/vCenterServer/cfg/` value to the variable.

- From the **Overview** page of the Virgo server instance that opens when you double-click in the instance in the Server view, configure the following options:
  - From the **Server Startup Configuration** pane, select the **Tail application trace files into Console view** and **Start server with -clean option** startup options.
  - From the **Redeploy Behavior** pane, remove the `*.xml` file extension.
  - From the **General Information** pane, select **Open launch configuration**. Click the **Arguments** tab and under VM arguments add the following lines at the end of the text:

```
-XX:+CMSClassUnloadingEnabled
-XX:MaxPermSize=512m
```

Set Up the Adobe Flex 4.6 Software Development Kit

To compile the resources of your HTML plug-ins, you must install the Adobe Flex 4.6 SDK on your development machine.

Procedure

1. From the Adobe Web site at http://www.adobe.com/devnet/flex/flex-sdk-download.html, download the Adobe Flex 4.6 SDK.
2. Extract the contents of the SDK distribution file on your development machine.
3. Set up the `FLEX_HOME` environment variable to point to the location of the Adobe Flex SDK on your machine.
4. (Optional) Go to the *`your_Flex_sdk`*`/frameworks/libs/player` directory and verify that you have a `11.5` folder that contains the `playerglobal.swc` file.
   a. If the folder is not present in your Adobe Flex SDK, download the missing file from the following location http://fpdownload.macromedia.com/get/flashplayer/installers/archive/playerglobal/playerglobal11_5.swc.
   b. Rename the downloaded file to `playerglobal.swc`.
   c. Create a `11.5` folder under the *`your_Flex_sdk`*`/frameworks/libs/player` directory and place the renamed file there.

vSphere Web Client SDK Upgrade

4

The vSphere Client development kit and the vSphere Web Client SDK provide different options for upgrading your existing plug-ins to be compatible with the vSphere Client .

Upgrade options and recommendations depend on the version of the SDK that you used to create your plug-ins.

This section includes the following topics:

Upgrading Plug-Ins Created with the vSphere Web Client SDK 6.0 Update 2

You can upgrade your existing plug-ins created with the vSphere Web Client SDK 6.0 Update 2 by applying changes to the URL formatting and the icon usage.

To make your existing plug-ins compatible with the vSphere Web Client 6.5 and the vSphere Client, you must consider making changes to the URLs and icon definitions.

URL Changes

The URL changes that you must apply affect the `web-platform.js`, HTML, and JavaScript files where the `/vsphere-client` root path must not be hard-coded. You must keep the `/vsphere-client` root path only in the `MANIFEST.MF` and `plugin.xml` files.

The user interface bundle of each plug-in contains a specific generation of the `web-platform.js` file. This file is used to bootstrap the JavaScript API based on the Web browser application on which the plug-in is deployed. In vSphere 6.5, the `WEB_PLATFORM.getRootPath` function is added to set the correct path to both Web browser applications. To reflect this change in your existing plug-ins, add the following line inside the `if(!WEB_PLATFORM)` block:

```
WEB_PLATFORM.getRootPath = function() { return "/vsphere-client"; }
```

Then you must use this function to define the Web context path in the following way: *my_plugin_namespace*.webContextPath = WEB_PLATFORM.getRootPath() + "*/my_plugin*";. For example, if you have the following definition of the Web context path in vSphere 6.0 Update 2: com_vmware_samples_chassisa.webContextPath = "/vsphere-client/chassisa";, in vSphere 6.5 you must change the definition to: com_vmware_samples_chassisa.webContextPath = WEB_PLATFORM.getRootPath() + "/chassisa";.

In the HTML files of your UI extensions, you must change the URLs that start with /vsphere-client/ to use relative URL paths. This change allows the root context path to be adjusted at runtime depending on the Web browser application on which the plug-in is deployed. For example, you must change the URLs in the HTML files of the ChassisA sample from:

```
  <link rel="stylesheet"
       href="/vsphere-client/chassisa/assets/css/jquery-ui-1.10.3.marge.css" />
<script src="/vsphere-client/chassisa/assets/jquery-1.10.2.min.js"></script>
<script src="/vsphere-client/chassisa/assets/jquery-ui-1.10.3.custom.min.js"></script>
<script src="/vsphere-client/chassisa/resources/js/web-platform.js"></script>
<script src="/vsphere-client/chassisa/resources/js/jquery-util.js"></script>
```

to the following relative paths:

```
 <link rel="stylesheet" href="../assets/css/jquery-ui-1.10.3.marge.css" />
<script src="../assets/jquery-1.10.2.min.js"></script>
<script src="../assets/jquery-ui-1.10.3.custom.min.js"></script>
<script src="./js/web-platform.js"></script>
<script src="./js/jquery-util.js"></script>
```

In the JavaScript code of your UI extensions, you can use the buildDataUrl(objectId, propList) utility function to make data requests that use the correct context path. You must remove also all mentions of /vsphere-client/ from the URLs in your JavaScript code.

External Icons

Your plug-in might include external icons, such as Home view shortcut icons, menu icons, dialog box title icons, and object list icons. In vSphere 6.0 Update 2, these icons are defined in the plugin.xml by using the following #{key} format and their values are set in the .properties files. The vSphere Web Client uses the default resource bundle declared in the plugin.xml manifest file to load the icons.

In vSphere 6.5, the vSphere Client loads the external icons that are defined with separate CSS classes in the additional plugin-icons.css file located in the assets/css/ directory of your UI component. The CSS class names must follow the *.bundleName-iconName* pattern and you must set the width, height, and display attributes for each class. Use the following settings for object list icons:

```
vertical-align: text-bottom;
margin: 1px 4px 0;
```

For Home screen shortcuts and dialog box title icons, use the following setting:

```
vertical-align: top;
```

To make the vSphere Client aware of these external icons, declare the `plugin-icons.css` CSS file as a dependency in the `plugin.xml` manifest file.

Upgrading Plug-Ins Created with the vSphere Web Client SDK 6.0 Update 1

You can upgrade your existing plug-ins created with the vSphere Web Client SDK 6.0 Update 1 by applying changes to the properties filenames, the initialization code of the `web-platform.js` file, and the `bundle-context.xml` file.

To make your plug-ins developed with the vSphere Web Client SDK 6.0 Update 1 compatible with vSphere 6.5, you must apply the following changes.

- Change the filenames suffix of the `.properties` files located in the `webapp/locales` folder of your existing plug-ins for example from `_en_US` to `-en_US`. If you keep the existing filenames of your localization resources, the vSphere Client and the vSphere Web Client display the keys of the strings instead of the localized values.

  After you apply this change, open the `build-war.xml` Ant build script for generating `.war` files and change the value of the `REGEXP2` environment variable to `\2_\1.properties`. Rebuild the plug-ins to make them compatible with the current version of both Web browser applications.
- Remove all mentions of `viewResolvers` from your `bundle-context.xml` files to avoid conflicts.
- Edit your `web-platform.js` JavaScript API initialization files to start with the following code:

```
    var WEB_PLATFORM = self.parent.WEB_PLATFORM;
if (!WEB_PLATFORM) {
   WEB_PLATFORM = self.parent.document.getElementById("container_app");
   self.parent.WEB_PLATFORM = WEB_PLATFORM;
   WEB_PLATFORM.getRootPath = function() { return "/vsphere-client"; }
}
```

Upgrading Plug-Ins Created with the vSphere Web Client SDK 6.0

You can upgrade your existing plug-ins created with the vSphere Web Client SDK 6.0 by applying changes to your existing plug-ins.

You must apply all changes required to upgrade plug-ins created with the vSphere Web Client SDK 6.0 Update 1 before you proceed with the next three changes.

Adapt your plug-ins to the following changes:

- The error handling mechanisms in the Java controllers. The code using the `@ExceptionHandler` annotation must be updated to return the correct error map in case of exception. For more information about the returned error response, see the `ServicesController.java` code sample located at the `globalview-html-service\src\main\java\com\vmware\samples\globalview\mvc` directory of the Global View sample.
- The properties of the `UserSession` object that the `WEB_PLATFORM.getUserSession()` function returns no longer contain the `samlTokenXml` property.
- The `WEB_PLATFORM.onGlobalRefreshRequest()` function is replaced with the `WEB_PLATFORM.setGlobalRefreshHandler(callback)` function. This function is called when a global refresh is initiated by the user.

Upgrading Plug-Ins Created with the vSphere Web Client SDK 5.5.x

You can upgrade your existing plug-ins created with the vSphere Web Client SDK 5.5.x to be compatible with vSphere 6.5 by applying changes to your JavaScript code and `plugin.xml` manifest files.

You must apply all changes required to upgrade plug-ins created with the vSphere Web Client SDK 6.0 before you proceed with the next three changes.

To upgrade your existing plug-ins that are created with the vSphere Web Client SDK 5.5.x, you must adapt the plug-ins to the following additional changes:

- Use the `WEB_PLATFORM.setDialogSize()` and `WEB_PLATFORM.setDialogTitle()` JavaScript functions introduced to the APIs for vSphere 6.0 to change the dialog size and title at runtime.
- Use the `<dialogIcon>` and `<showCloseButton>` XML elements in your `plugin.xml` manifest files to specify an icon and control the close button visibility for the dialog boxes and **Summary** view portlets.
- Use the `WEB_PLATFORM.openModalDialog()` function introduced to the JavaScript APIs for vSphere 6.0 to open a modal dialog box from within an HTML view.

Quick Startup Guide for Developing HTML-Based Plug-Ins

5

After you successfully install and configure your vSphere Client development kit, you can easily create an HTML plug-in project, build and test your plug-in with a local or remote vSphere Web Client, or a remote vSphere Client .

This section includes the following topics:

Before Creating an HTML Plug-In

Ensure that your development machine is prepared for developing HTML plug-ins by verifying that you installed and set up the required software.

To develop HTML plug-ins make sure that you have familiarized yourself with the folder structure of the SDK and you have followed the steps for setting up your development environment.

For more information about developing a plug-in that can run on both Web browser applications, see Hybrid Plug-Ins.

Creating an HTML Plug-In Project

HTML plug-ins for the vSphere Client have two components. User interface components run in the Web browser and Java service components run on the Virgo server. The vSphere Client development kit provides tools for creating an HTML plug-in project template for each of these components.

You have two options for creating the HTML plug-in project template from which you can choose depending on your development setup:

- Create the project template by using the scripts provided in the `vsphere-client-sdk\html-client-sdk\tools\Plugin generation scripts` directory.
- Create an HTML plug-in project by using the vSphere Web Client Tools Eclipse plug-in. For more information about how to set up the SDK Eclipse plug-in, see Install the vSphere Web Client Tools Eclipse Plug-In.

Generate an HTML Plug-In Project with a Script

You can run the plug-in project generation scripts to create an HTML plug-in project template and build a plug-in out of the project.

The vSphere Client development kit provides two project generation scripts which you can use depending on the operating system of your development environment.

Prerequisites

- Verify that you set up the correct paths for the `ANT_HOME` and `VSPHERE_SDK_HOME` environment variables. See Automate the Plug-In Build Process.
- Verify that you set up the correct path for the `FLEX_HOME` environment variable. See Set Up the Adobe Flex 4.6 Software Development Kit.

Procedure

1. In your development environment, open a command prompt or launch the Terminal application.
2. Navigate to the `Plugin generation scripts` folder.

   On a Windows machine, the generation scripts are located at *`SDK_folder`*`\vsphere-client-sdk\html-client-sdk\tools\Plugin generation scripts`.
3. Run the `create-html-plugin.bat` or the `create-html-plugin` script depending on your OS.
4. When prompted, enter the plug-in name, the directory on your machine where the project template folder structure will be created, and the plug-in package name.

   If you do not specify a value when prompted, the generation script uses predefined values.

The script generates two folders, `myplugin-service` and `myplugin-ui`. For more information about the contents of each folder, see Contents of the HTML Plug-In Project Template.

What to do next

After you generate the HTML plug-in project template, you can build the plug-in package and test whether your plug-in works by deploying the plug-in on the vSphere Client or the vSphere Web Client. For detailed information, see Building a Plug-In Package from the Project Template and Testing the Generated Plug-Ins.

Create an HTML Plug-In Project with Eclipse

If you have the Eclipse IDE set up on your development environment, you can create HTML plug-in projects by using the vSphere Web Client Tools Eclipse plug-in.

Prerequisites

- Verify that you have the Eclipse IDE installed and configured correctly on your development environment. See (Optional) Set Up the Eclipse Integrated Development Environment.

- Verify that you have the vSphere Web Client Tools Eclipse plug-in installed and configured in the Eclipse IDE. See Install the vSphere Web Client Tools Eclipse Plug-In.
- Verify that you have the Virgo server set up in your Eclipse IDE. See Configure the Virgo Server in Your Eclipse IDE.

Procedure

1. Start Eclipse on your development machine.
2. From **File > New**, select **Other**.

   The New wizard appears.
3. From the New wizard, select the HTML Plug-in Project node under the vSphere Web Client folder and click **Next**.

   The New vSphere Client HTML plug-in dialog box appears.
4. In the New vSphere Client HTML plug-in dialog box, enter the name of the plug-in project and click **Finish**.

   Best practice is to use `–ui` at the end of the project name for the user interface components and to use lowercase letters. The project name is used for creating the Web context path of the plug-in.

   Optionally, you can change the location where your plug-in project is stored and also the default plug-in and plug-in package names.

Two plug-in projects are created for the HTML plug-in, user interface project and Java service project. Before you start editing the files generated by the wizard, make sure that you understand what each file must contain. For detailed information about the structure of the HTML plug-in project template, see Contents of the HTML Plug-In Project Template.

What to do next

Before you start editing the generated HTML project files, you can build and deploy the HTML plug-in on the vSphere Client or the vSphere Web Client.

Contents of the HTML Plug-In Project Template

Once you create a template project for your HTML plug-in, you must be familiar with the folder structure of the project and the purpose of each file inside the project. This knowledge will help you to easily create your custom plug-ins for the vSphere Client.

The following tables contain detailed information about the structure of the UI and Java service components of the HTML plug-in project template.

Table 5-1. Details About the UI Project Template Structure

User Interface Project Structure					Description
/myplugin-ui	/src/main	locale	en_US		Contains string resources for the English locale. You can use the build-resources script to compile the resources.
			fr_FR		Contains string resources for the French locale. You can use the build-resources script to compile the resources.
		webapp	assets	css	Contains assets used in the plug-in such as images, CSS files, and JavaScript libraries. The css folder contains the plugin-icons.css file that you can use to define the external icons. For more information, see Guidelines for Creating Plug-Ins Compatible with the vSphere Client.
				images	
			META-INF	MANIFEST.MF	Contains the manifest file of the WAR bundle. The file contains the following manifest headers: ■ Bundle-SymbolicName - Specifies a unique identifier for the bundle. ■ Web-ContextPath - Specifies the Web application context path which must start with vsphere-client/. ■ Import-Package - Lists the packages on which the UI bundle depends including references to classes from the Java service layer.
			resources	js	Contains the HTML and JavaScript resources of the UI bundle. The js folder includes also the web-platform.js file. This file contains the JavaScript initialization code that is generated for each project. For more information, see vSphere Client JavaScript APIs.
				host-monitor.html mainView.html	
			WEB-INF	spring bundle-context.xml	Contains the Spring configuration which you can use to declare the services from the service layer that the UI bundle uses.
				web.xml	The deployment descriptor that describes the classes, resources and configuration of the application.

Table 5-1. Details About the UI Project Template Structure (Continued)

User Interface Project Structure	Description
plugin.xml	The manifest file that uses metadata to define the plug-in extensions and resources. See User Interface Plug-In Module Manifest.
build-plugin-package.bat build-plugin-package.sh build-plugin-package.xml build-resources.bat build-resources.sh build-resources.xml build-war.bat build-war.sh build-war.xml plugin-package.xml	Contains the automation scripts for generating the WAR file, compiling the resources for the UI component, for building the plug-in package folder from the UI and Java service components, and the template `plugin-package.xml` manifest file. For more information about he structure of the plug-in package and the metadata provided with the `plugin-package.xml` file, see Plug-In Package Overview and XML Elements of the Plug-In Package Manifest File.

Table 5-2. Details About the Java Service Project Template Structure

Java Service Project Structure				Description
myplugin-service	/src/main	java/com/mycompany/myplugin	mvc	Contains the controllers that manage requests to the `/actions.html`, `/data`, and `/services` endpoints. The folder contains also utility classes related to the controllers.
			services	Contains the sample `Echo` service and the sample service that responds to user actions.
		resources/META-INF	spring	Contains the Spring configuration files and the bundle manifest file.
			MANIFEST.MF	
	build-java.bat build-java.sh build-java.xml			Contains the automation scripts for generating the Java service bundle.

Building a Plug-In Package from the Project Template

You can validate your development environment setup by generating an HTML plug-in from the plug-in project template. Use the automation scripts provided with the SDK for building the plug-in.

To create a plug-in package from the project template, run the `build-plugin-package.bat` or the `build-plugin-package.sh` script depending on your operating system. You can locate these scripts in the *`plugin_name`*`-ui` folder of the project template.

After running the script, you see the *`plugin_name`* folder that contains the `plugin-package.xml` manifest file and the `plugins` folder with the WAR and JAR files generated for the UI and service components. For more information about the XML elements of the `plugin-package.xml` manifest file, see XML Elements of the Plug-In Package Manifest File.

Example: Plug-In Package Manifest File

The following example shows the contents of the `plugin-package.xml` manifest file that is generated for the template HTML plug-in.

```
<pluginPackage id="com.mycompany.myplugin" version="1.0.0" type="html" name="myplugin"
      description="Add plugin description" vendor="Add vendor">
   <dependencies>
      <pluginPackage id="com.vmware.vsphere.client" version="6.0.0" />
   </dependencies>
   <bundlesOrder>
      <bundle id="com.google.gson" />
      <bundle id="com.mycompany.myplugin.myplugin-service" />
      <bundle id="com.mycompany.myplugin.myplugin-ui" />
   </bundlesOrder>
</pluginPackage>
```

You can follow these recommendations for the `plugin-package.xml` file to ensure that your plug-in can be deployed on both Web browser applications:

- Specify a unique plug-in package ID for the `id` attribute of the `pluginPackage` XML element.
- Add the `type="html"` attribute to the `pluginPackage` elements. This attribute is required if you want your plug-in to be deployed on the vSphere Client.
- Specify that your plug-in depends on the `com.vmware.vsphere.client` package with `6.0.0` version. This dependency ensures that your plug-in can be deployed on the vSphere Web Client 6.0.

Testing the Generated Plug-Ins

You can verify that your plug-in packages work correctly on both Web browser applications by deploying the plug-ins on a local and remote vSphere Client and vSphere Web Client.

Deploy the Plug-In on a Local vSphere Client

You can verify that your plug-in works as expected by first deploying the plug-in on the local vSphere Client provided with the vSphere Client development kit.

You can use two options to deploy your plug-ins on the local vSphere Client.

- Copy the plug-in package folder into the `plugin-packages` directory. For example, on a Windows machine, you can locate this folder under *`your_SDK_location`*`\vsphere-client-sdk\html-client-sdk\vsphere-ui`.
- Copy the UI and Java service bundles into the `pickup` directory on the server. For example, on a Windows machine you can locate this folder under *`your_SDK_location`*`\vsphere-client-sdk\html-client-sdk\vsphere-ui\server`.

If you use the first option make sure that you restart the vSphere Client server to pick up the new plug-ins.

The procedure that follows describes how you can use the second option for speeding up your development process. You can repeat the steps for each new version of the UI and Java service components of your plug-in.

Note If you use the `pickup` directory, you must be aware that the string resources are not reloaded when you update the bundles until you restart the vSphere Client Virgo server.

Prerequisites

- Verify that you register the local vSphere Client with the vCenter Server instance. See Register Your Local vSphere Client with the vCenter Server Instance.
- Verify that you run successfully the automation script for generating the plug-in package folder for your plug-in. See Building a Plug-In Package from the Project Template.

Procedure

1. Navigate to the `plugin` folder where the WAR and JAR files of your plug-in are generated.

   For example, on a Windows machine if you used the default settings of the plug-in generation script, go to *`your_SDK_location`*`\vsphere-client-sdk\html-client-sdk\tools\Plugin generation scripts\plugin-packages\myplugin\plugin`.

2. Copy JAR files to the `pickup` folder on the server. If the JAR files are deployed successfully, copy the WAR files to the same folder.

   For example, on a Windows machine you can paste the files in the *`your_SDK_location`*`\vsphere-client-sdk\html-client-sdk\vsphere-ui\server\pickup` directory.

   The Virgo server console is updated when the bundles are deployed on the local vSphere Client.

3. Refresh your Web browser at `https://localhost:9443/ui` to see the changes.

What to do next

To complete the verification of your plug-in, deploy the plug-in on a local vSphere Web Client, and on a remote vSphere Client and vSphere Web Client.

Deploy Your Plug-In on a Local vSphere Web Client

You can deploy your custom plug-ins on the local vSphere Web Client to verify whether the plug-ins run as expected.

You can use the same options for deploying your plug-ins to the local vSphere Web Client as the ones described for the vSphere Client. See Deploy the Plug-In on a Local vSphere Client.

The procedure that follows describes how you can deploy your plug-ins by using the `plugin-packages` folder. For example, on a Windows machine you can locate this folder under *`your_SDK_location`*`\vsphere-client-sdk\flex-client-sdk\vsphere-client`.

Prerequisites

- Verify that you register your local vSphere Web Client to a vCenter Server instance. See Register Your Local vSphere Client with the vCenter Server Instance.

Procedure

1. On your development machine, navigate to the `plugin-packages` folder where your custom plug-in is generated.

   For example, on a Windows machine if you use the default options when running the plug-in generation script, you can go to *`your_SDK_location`*`\vsphere-client-sdk\flex-client-sdk\tools\Plugin generation scripts\plugin-packages`.

2. Copy the *`plugin_name`* folder to the `plugin-packages` folder under `vsphere-client`.

   For example, on a Windows machine copy the folder to *`your_SDK_location`*`\vsphere-client-sdk\flex-client-sdk\vsphere-client\plugin-packages`.

3. Start the vSphere Client Virgo server by running the `startup` script under `bin`.

   For example, on a Windows machine you can find the `startup` script at *`your_SDK_location`*`\vsphere-client-sdk\flex-client-sdk\vsphere-client\server\bin`.

4. Open your Web browser and log in at `https://localhost:9443/vsphere-client` to see the changes.

What to do next

To complete the verification of your plug-in, deploy the plug-in on a remote vSphere Client and vSphere Web Client.

Deploying Your Plug-In on a Remote vSphere Client or vSphere Web Client

You can verify whether your custom plug-in runs as expected by deploying the plug-ins on a remote vSphere Client and vSphere Web Client.

You can register your plug-ins with the remote Web browser applications by using one of the following options:

- Create an `Extension` data object and register the data object with the `ExtensionManager` by using the Managed Object Browser (MOB) of your vCenter Server instance. See Register a Plug-In Package as a vCenter Server Extension.
- Use the vCenter Server plug-in registration tool provided with the vSphere Client development kit.

vCenter Server Plug-In Registration Tool

In vSphere 6.5, the vSphere Client development kit provides a tool to ease the registration of custom plug-ins with the vSphere Client and vSphere Web Client. You can locate the tool at the `vCenter plugin registration` folder under `html-client-sdk\tools`.

The `prebuilt` folder contains the `extension-registration` script that allows you to register and unregister your plug-ins as extensions to the vCenter Server instance. You can also update the registration of an existing extension.

The `project` folder contains the source code and build scripts for the plug-in registration tool which you can use to extend the logic of the tool.

To use the plug-in registration tool, run the script from the command line by providing the following command-line options:

```
extension-registration -action <action> [-c <company>] [-k <key>]
 [-n <name>] [-p <vc pass>] [-pu <plugin url>] [-s <summary>]
 [-show] [-st <server thumbprint>] [-u <vc user>] [-url <vc url>]
 [-v <version>]
```

Table 5-3. Command-Line Options for the Plug-In Registration Tool

Command-Line Option	Description
`-action` <*action*>	The action that the tool must perform. You can choose from the following options: ■ **`registerPlugin`** ■ **`unregisterPlugin`** ■ **`isPluginRegistered`**
`-c` or `--company` <*company*>	The company that developed the plug-in.
`-k` or `--key` <*key*>	The unique extension key that must be the same as the plug-in package ID of your plug-in.
`-n` or `--name` <*name*>	The name of your plug-in.
`-url` <*vc url*>	The URL of the vCenter Server instance where you want to register your plug-in. The URL must end with `/sdk`.
`-p` or `--password` <*vc pass*>	The credentials for logging into the vCenter Server instance.
`-u` or `--username` <*vc user*>	
`-pu` or `--pluginUrl` <*plugin url*>	The URL from which your plug-in package ZIP file is downloaded.
`-s` or `--summary` <*summary*>	The short description of your plug-in.
`-show` or `--showInSolutionManager`	The plug-in is available under **Administrator > Solutions > vCenter Server Extensions**. **Note** In vSphere 6.5, Administration menus are not implemented for the vSphere Client.
`-st` or `--serverThumbprint` <*server thumbprint*>	The thumbprint of the Web server hosting your plug-in package. This option is required when your plug-in package ZIP file location is a secure URL (HTTPS).
`-v` or `--version` <*version*>	The dot-separated version number of the plug-in package that is defined in the `plugin-package.xml` manifest file.

For example, to register the `com.acme.myplugin` plug-in with version `1.0.0` that is located at `https://150.20.23.254/MyPluginpackage.zip`, use the following command on a Mac OS development machine:

```
./extension-registration.sh -url https://10.23.222.35/sdk -username administrator@vsphere.local -
password adninistrator -action registerPlugin -key com.acme.myplugin -version 1.0.0 -pluginUrl
https://150.20.23.254/MyPluginpackage.zip -serverThumbprint 99:FD:2B:0D:12:85:37:AA:DA:A0:08:E1:F4:3B:
4A:E6:08:AC:49:CD
```

After you register your custom plug-in, log in the vSphere Client or vSphere Web Client to verify that the plug-in is visible in the remote Virgo server. You can also use the MOB of your vCenter Server instance to view all registered extensions.

Developing Flex-Based User Interface Extensions

6

You can create extensions to the user interface layer of the vSphere Web Client that add GUI elements that are specific for your vSphere environment.

The vSphere Web Client user interface is based on a modular Flex application, hosted on the application server, that runs in the 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 extensions. A Flex user interface plug-in module contains one or more extensions, which add Flex GUI elements to the vSphere Web Client user interface.

This section includes 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, send commands, or change 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 user 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 user interface.

User Interface Plug-In Module Manifest

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 File

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.

Table 6-1. Attributes of the <plugin> Element

Attribute Name	Attribute Description
`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 must 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

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 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 your Web browser specifies at runtime.

For the vSphere Web Client to import the resource locale properly 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.

<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.

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 resource 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 by using the following 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.

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 appears, including which Flex classes contain the actual GUI code.

Depending on the type of extension that you create for the vSphere Web Client, you must create different metadata definition for each extension, 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.

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 Objectscategorized object relations 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.

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 the Example: Extension Definition, 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. To find the required extension object type for any given extension point, see Chapter 12 List of Extension Points.

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 specify explicitly a type. You only have to specify explicitly 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 the Example: Extension Definition, 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.

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: Extension Definition, 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 HTML Bridge Flex classes provided by the SDK. Otherwise, you must create the Flex component. In Example: Extension Definition ,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

Example: Extension Definition

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 the virtual machine objects and appears only to users with the **Global.Licenses** privilege.

```
<!-- 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>
    <metadata>
       <privilege>Global.Licenses</privilege>
    </metadata>
 </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 by using the extension unique identifier.

Ordering Extensions

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 you specify the same value for the `<precedingExtension>` element for several extensions, the extensions are rendered 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 ensure that the `PerformanceView` extension is rendered first, followed by the `UtilityView` extension.

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 the `<metadata>` element of your extension definition.

Filtering Based on Selected Object Type

You can filter your extension to appear only 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.

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.

Example: Extension Filtered by Entity Type

The following example filters the extension action to appear only when the user has selected a virtual machine object.

```
<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>
```

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

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

In the 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.

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: Example Property Value Filter

The following example filters an action extension only when the value of the isRootFolder property is true and the selected object contains child objects that are virtual machines.

```
<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>true</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 appear only 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 specified privilege values for the extension to appear.

Example: Extension Filtered by User Privileges

The following example filters the extension to appear only when the user privileges include **Global.Licenses**.

```
<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>
```

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 create automatically extension definitions. The vSphere Web Client SDK provides templates for some 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 Extending the vCenter Object Workspaces and Extending the Object Navigator.

Types of User Interface 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.

Global View Extensions

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 Creating Data View Extensions.

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.

Object Workspace Extensions

When you select a vSphere object in the object navigator, the object 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.

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 about the available extension points, see Defining Extensions.

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 Creating Data View Extensions.

You can also add an object workspaces for a custom object that has the default structure and tabs. In general you create object workspace extensions to support custom vSphere object types that you add to the vSphere environment.

Object Navigator Extensions

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

You can navigate to the objects in the virtual infrastructure by using the object navigator vCenter Server page. The vCenter Server page shows the objects in the vSphere environment in inventory trees and inventory lists. You can extend this level of the object navigator with an 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 Server page.

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

Action Extensions

Actions represent the commands and requests that the users send to the vSphere environment to make changes to the virtual infrastructure.

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.

Relation Extensions

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** tabcategorized tabs for objects relations in the selected object 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 an HTML class.

Home Screen Shortcut Extensions

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.

Object List View Extensions

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.

Adding Global View Extensions

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 by using the XML elements in the plug-in module manifest file, and create the Flex code that appears in the main workspace.

Use Cases

You can use global view extensions to create dashboard-style data views or console-style applications.

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.

Console-style applications are displayed in the vSphere Web Client main content area. For example, the vSphere Web Client Task Console and Event Console are console-style applications.

Creating Global View Classes

Each global view extension is an independent Flex 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. Flex 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 by 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 Creating Data View Extensions.

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. For more information about how to create a pointer to a global view, see Extending the Object Navigator and Creating Home Screen Shortcuts.

Properties of the `GlobalViewSpec` Extension Object

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.vsphere.client.views.GlobalViewSpec`.

The following table describes the properties that you provide for the `com.vmware.vsphere.client.views.GlobalViewSpec` extension data object by using the `<object>` element in your extension definition.

Property	Type	Description
<name>	string	The name of the global view that appears as a title in the main workspace. In the Example: Example Flex-Based Global View Extension, the value is hard-coded as My Console App. You can also use a dynamic resource string from your plug-in module.
<componentClass>	string	The Flex class that implements the global view. The Flex class appears in the vSphere Web Client main workspace. You must set the value of the <componentClass> element className attribute to the fully qualified name of the Flex view class. If you develop an HTML-based extension, use the com.vmware.vsphere.client.htmlbridge.HtmlView Flex object. See Properties of the HtmlView Extension Object.
<applyDefaultChrome>	boolean	A Boolean value that indicates whether the global view appears with the default header, footer, title, and icon graphics provided by the vSphere Web Client. The property is optional and the default value is true. If the property is omitted, the property value is considered to be the default.

Example: Example Flex-Based Global View Extension

The following example extension definition adds a console application to the main area by creating a global view extension. The <object> element defines a data view object of type com.vmware.vsphere.client.views.GlobalViewSpec.

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

Extending the vCenter Object Workspaces

The vSphere Web Client displays a standard object workspace for each type of vSphere object. You can extend the existing object workspaces or create an object workspace for a custom 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 **Summary**, **Monitor**, **Configure**, and **Related Objects** categorized object relations top-level tabs , 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.

Use Cases

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 8 Developing Extensions to the Service Layer.

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.

Extending an Existing Object Workspace

The vSphere Web Client SDK provides a unique extension point for each of the default tabs and subordinate views for all object types in the vSphere environment. You can define a data view extension that references one of these extension points to create a subordinate view inside that tab or view.

Extension points use the following naming convention:

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

The `${objectType}` value indicates the type of vSphere object which object workspace you want to extend. The `${view}` value specifies the exact view to which you 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 **Configure** tab in the object workspace for virtual machine objects.

For a complete list of object workspace extension points, see Object Workspace Extension Points.

Types of Data Views

A data view extension appears differently depending on the vSphere object that you specified with the extension point. Data views can appear in the object workspace having one of the following structures.

- Second-level tab - If you define an extension to a top-level tab, such as **Summary**, **Monitor**, or **Configure**, a data view extension appears as a second-level tab in the object workspace.
- Second-level tab view - If you define an extension to one of the existing second-level tabs, such as the **Performance** view inside the **Monitor** tab, a data view extension appears as a view within the second-level tab in the object workspace.
- Table of contents views - If you define an extension to the **Configure** tab, an item view appears in the table of contents located on the left side of the tab.
- Portlet - If you define the **Summary** tab extension point, a data view extension appears as a portlet in the 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.

Properties of the ViewSpec Extension Object

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`.

The following table describes the properties that you provide for the `com.vmware.ui.views.ViewSpec` extension data object by using the `<object>` element in your extension definition.

Property	Type	Description
`<name>`	string	The name of the data view that appears as a title where appropriate in the user interface. For example, the string can be the name of the second-level tab or the portlet title. The value of the `<name>` property can be a hard-coded string or a dynamic resource included in your plug-in module. In Example: Example Portlet Extension for Host Summary Tab, the portlet title is hard-coded as `My Summary Portlet`.
`<componentClass>`	string	The Flex class for your data view extension that appears at the user interface location corresponding to the extension point that you specify. You must set the value of the `className` attribute of the `<componentClass>` element to the fully qualified name of the Flex view class. If you implement the view by using the HTML bridge, specify the `HtmlView` container class. For more information about the `HtmlView` container class, see Properties of the HtmlView Extension Object.

Example: Example Portlet Extension for Host Summary Tab

The following example extension definition 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`.

```
<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>
```

Creating an Object Workspace for a Custom Object

If you add a custom vSphere object, you can create the object workspace by using the XML templates provided with the SDK. You instantiate the XML template in the `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**, **Configure**, and the tabs for the different related objects groups.

Using the `objectViewTemplate` to Create an Object Workspace

You can create an instance of the standard object view template by using the `<templateInstance>` XML element in your `plugin.xml` manifest file.

To create an instance of the standard object view template, you use the `<templateInstance>` element in your `plugin.xml` manifest file. Inside the `<templateInstance>` element, you define the variables that describe the specific instance of the template. The `<templateInstance>` element can appear anywhere inside the root `<plugin>` element in the `plugin.xml` manifest file, but a best practice is to include the template definition with other extension definitions.

You must add the `<templateId>` element inside the `<templateInstance>` element. This 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, use the `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. Extension points for a custom object workspace are formed by using the following schema for each data view:

```
<namespace_value>.<default_extension_point_name>
```

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

- `<namespace>.gettingStartedViews` - Adds views under the **Getting Started** tab.
- `<namespace>.summaryViews` - Adds views under the **Summary** tab.
- `<namespace>.monitorViews` - Add views under the **Monitor** tab.
- `<namespace>.monitor.issuesViews` - Adds views to the **Issues** second-level tab of the **Monitor** tab.
- `<namespace>.monitor.performanceViews` - Adds views to the **Performance** second-level tab of the **Monitor** tab.
- `<namespace>.monitor.performance.overviewViews` - Adds views to the **Overview** performance charts under the **Monitor** tab.

- `<namespace>.monitor.performance.advancedViews` - Adds views to the **Advanced** performance charts under the **Monitor** tab.
- `<namespace>.monitor.tasksViews` - Adds views to the **Tasks** second-level tab under the **Monitor** tab.
- `<namespace>.monitor.eventsViews` - Adds views to the **Events** second-level tab under the **Monitor** tab.
- `<namespace>.manageViews` - Adds an entry in the table of contents under the **Configure** tab.
- `<namespace>.manage.settingsViews` - Adds an entry to the **Settings** node in the table of contents under the **Configure** tab.
- `<namespace>.manage.alarmDefinitionsViews` - Adds views to the **Alarm Definitions** entry in the table of contents under the **Issues** second-level tab of the **Monitor** tab.
- `<namespace>.manage.permissionsViews` - Adds views to the **Permissions** tab.

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.

Creating the Data Views Within the Template

After you create an object workspace by 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. For more information about defining data view extensions, see Properties of the ViewSpec Extension Object.

Top-level tabs, second-level tabs, and views created by 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 for the Rack object from the following example, the **Performance** second-level tab under the **Monitor** tab does not appear for Rack objects.

Example: Instantiating the Standard Template

The following example instantiates the object view template for a new object called Rack. 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 by using the same convention.

```
<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>
```

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 header view displayed at the top of the main workspace, and an extension point for the portlet views under the header view. 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 `plugin.xml` manifest file by using the `<templateInstance>` element. The **Summary** tab template definition must appear after the definition for the standard object workspace.

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 parent object workspace of the **Summary** tab. 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.

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 extension. 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. For more information about defining data view extensions, see Properties of the ViewSpec Extension Object.

Example: Instantiating the Summary Tab Template

The following example instantiates the **Summary** tab template for a custom object called Rack. The `<templateInstance>` element contains an `id` attribute, which is a unique identifier of your choice.

```
<!-- 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>
```

Creating Data View Extensions

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

Flex classes for GUI elements in the vSphere Web Client are implemented by using the Model-View-Controller (MVC) 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 develop Flex-based data view extensions for the vSphere Web Client, 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.

MVC Pattern Overview

The Model-View-Controller pattern separates the domain logic of the software from the user interface elements. The 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.

Understanding the Frinje Framework

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 the Frinje framework implements the Model-View-Controller architecture is essential to reading and making use of the Flex-based samples for data view extensions provided with the vSphere Web Client SDK. The vSphere Web Client SDK contains an Eclipse plug-in that enables you to create Flex and Java projects for your vSphere Web Client extensions.

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. As a result, your source code has 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, view and mediator classes.

The graphic elements of the View component are implemented in an MXML class and the logic elements of the View component are implemented in an ActionScript class. The Frinje framework refers to the MXML class as a view class, and the ActionScript class as a mediator class.

The view class and the mediator class are associated by using metadata tags. The MXML view class uses the `[DefaultMediator]` tag to specify the associated mediator class. The ActionScript mediator class uses the `[View]` tag to specify the associated view class.

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, such as 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.

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 by 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 by 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 has currently 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: Example Mediator Class Implementing `IContextObjectHolder`

The following example mediator class 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. 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.

```
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;
  }
   ...
}
```

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 (DAM) 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 Frinje Event-Driven APIs.

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.

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.

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: Simple Frinje Proxy Class

The following sample proxy class calls the simple `EchoService` service in the vSphere Web Client service layer.

```
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 =
ServiceUtil.getDefaultChannelUri(GlobalviewModule.contextPath);

    /**
    * 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.

Frinje Event-Driven APIs

You can develop APIs that are based on the Frinje framework and use the Frinje event bus.

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.

When you construct your classes, you can use the following metadata annotations: `[DefaultMediator]`, `[Event]`, `[Model]`, `[RequestHandler]`, `[ResponseHandler]`, `[EventHandler]` and `[View]`. Each annotation has certain required attributes you must include.

Note String arguments in metadata annotations are not validated at compile time. If you encounter errors or your extensions do not function correctly, review your metadata annotations. 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: `[DefaultMediator]` Metadata Tag in MXML view

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

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

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 Data Access Manager library. Your classes can dispatch these events to request data from the vSphere environment. See Using the Data Access Manager Library

Example: [Event] Metadata Tag in Mediator Class

The following example mediator class is 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. 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.

```
// 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));
  }
    ...
```

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

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 Creating Action Extensions.

```
[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: `[ResponseHandler]` Metadata Tag in Event Handler Method

The following example method is 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. The `onData` function is annotated to receive the event `com.vmware.data.query.events.DataByModelRequest.Response_ID`. The event must be generated from the parent class of the `onData` function. 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.

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

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: `[EventHandler]` Metadata Tag

The following example method is annotated to handle the global `DataRefreshInvocationEvent`.

```
[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.

Example: `[View]` Metadata Tag in Mediator Class

The following example mediator class 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.

```
/**
* 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 (DAM) 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.

Create a 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.

Procedure

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 mediator class of the extension must retrieve by 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 start from Step 2.

2. In the mediator class of your extension, 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 request events that retrieve a single property, you specify the property name by 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 the mediator class of your extension 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 Data Acess Manager.

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.

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.

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.

Example: Data Model Class

The following example shows a data model class. 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. 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.

```
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;
  }
}
```

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.

The following data model class is 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.

```
  [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. The following example 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.

```
  [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.

The following example data model class is 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.

```
[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. The following example shows a data model class that uses a nested data model to create an array of objects based on the Host data model.

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

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. The following example shows a data model class that uses a nested data model for a user-defined custom Chassis object.

```
[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

The mediator class of your extension 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`.

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 by 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.

Request-to-Response Mapping

The following table contains the types of data request events, the resulting data response events, and the associated result types included in the DAM API.

Table 6-2. Data Request and Data Response Events

Request Event	Result Type	Description
`DataByModelRequest`	Instance of specified data model class	Retrieves 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	Retrieves 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	Retrieves properties specified by data model for vSphere or custom objects that match the given constraint.
`DataByQuerySpecRequest`	`ArrayCollection` of `ObjectDataObject` (property-value map)	Retrieves data specified by query specification.

Example: Examples of Data Request and Data Response Events

The following 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.

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 by 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.

```
[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
    }
}
```

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.

```
[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 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.

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: Mediator Class Using DAM Binding Specification for Data Refresh

The following example mediator class 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.

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.

```
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;
        }
    }
```

Extending the Object Navigator

The object navigator is the primary navigation interface for reaching solutions and applications in the vSphere Web Client, and for finding and focusing on objects in the virtual infrastructure. 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.

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 virtual infrastructure through vCenter Server, and pointer nodes for other major solutions in the vSphere Web Client. The object navigator Administration page appears when you click Administration on the vSphere Web Client home page. The Administration page contains categories and pointer nodes specific to that application. The object navigator vCenter Server page appears when the user browses the objects in the virtual infrastructure. On the vCenter Server page, the object navigator displays the available objects in the vSphere environment in collections, such as inventory tree nodes and inventory list nodes.

Use Cases

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 to 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 page. See Creating Home Screen Shortcuts.

- Object Collections

  The object navigator vCenter Server 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 the workspace and data views of that object.

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 Server 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.

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.

The following table lists the pre-defined categories in the object navigator and the corresponding id values to which you must set the <parentUid> property.

Table 6-3. Object Navigator Categories and IDs

Category Type	Category ID
Object Navigator Home Page > **Administration** > **Solutions**	`vsphere.core.navigator.solutionsCategory`
Object Navigator **Global Inventory Lists** Page	`vsphere.core.navigator.virtualInfrastructure`
Object Navigator **Global Inventory Lists** Page > **Resources**	`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.

The following table lists the properties of the `ObjectNavigatorNodeSpec` data object that you must set to add a category to the object navigator.

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 the <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.

The following example extension definition is 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.

```
<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.

The following table lists the properties of the ObjectNavigatorNodeSpec data object that you must set to add a pointer node to the object navigator.

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.
<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: Example Pointer Extension to Object Navigator

The following example shows an 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.

```
<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.

The following table lists the properties of the `ObjectNavigatorNodeSpec` data object that you must set to add an object collection node to the object navigator.

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. Typically, most object collection nodes are found in the Inventory Lists category of the object navigator vCenter Server page. A best practice is to either add your object collection node to the Inventory Lists category, or to create a new category extension.

Property	Type	Description
<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.
<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.
<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: Example Entity Collection Node Extension

The following example shows an extension definition for an object navigator extension that adds an object collection node for a custom object type called Rack. In the example, the Rack object collection node appears under the Inventory Lists category in the object navigator vCenter Server 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.

```
<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 Create an Object Collection Node Extension

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: Example Usage of Object Collection Template

The following example shows how to instantiate the object collection template for a custom object type called Chassis. 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.

```
<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>
```

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 is 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.

The following table lists the properties of the `ObjectRepresentationSpec` data object that you must set.

Property	Type	Description
`<objectType>`	string	The object type for which the icon and label sets apply.
`<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.

Example: Object Representation Extension for Rack Object

The following example shows an extension definition for an object representation extension. In the example, the extension adds one new icon and label set for a custom object called Rack.

```
<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>
```

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: Object Representation Extension for Rack Object, 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.

Creating Action Extensions

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

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.

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 Java service. The Java service is responsible for performing the action operation on the target vSphere object.

Use Cases

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.

Actions Framework Overview

The Actions Framework governs all available actions in the vSphere Web Client. 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 action sets in the Actions Framework.

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.

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.

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.

Note 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 custom objects.

Defining Individual Actions for Flex-Based Action Extensions

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 by using a `<com.vmware.actionsfw.ActionSpec>` XML element. In this element, you set the properties for the action.

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	Optional. String containing a short description of the action, suitable for a tooltip.
`<icon>`	resource	Optional. Icon class or dynamic resource for the action.
`<acceptsMultipleTargets>`	boolean	Optional. If you set this value to `true`, the action can be made available when the user selects multiple vSphere objects.

Property	Type	Description
<conditionalProperty>	string	Optional. Indicates the 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.
<privateAction>	boolean	Optional. A flag that indicates that this action must not be used by the Actions Framework except when explicitly identified by its <uid>. For example, you can set the property value to true when you define a context-less action. Examples of such actions are global actions in a list toolbar or drag-and-drop actions that must not be available in the default menu.
<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.

Example: Flex-Based Action Extension Definition

The following example shows an 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 Chassis.

```
<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>
```

Handling Actions with Flex Command Classes

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: Flex-Based Action Extension Definition, 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: Example Flex Command Class

The following example shows an example Flex command class. In the example, the `ChassisCommand` class contains a handler method for each of the actions defined in Example: Flex-Based Action Extension Definition.

```
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: Flex-Based Action Extension Definition. The actual action operations are performed by an imported Java service. In the example, `ChassisServiceProxy` refers to a proxy class for a Java service that performs the action operations. See Performing Action Operations on the vSphere Environment.

Performing Action Operations on the vSphere Environment

You can create a Java service for your action extension to perform operations on objects in the vSphere environment. After the action operation completes, you must update accordingly the vSphere Web Client user interface.

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 by using a Java service in thevSphere 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 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 8 Developing Extensions to the Service Layer.

Obtaining the Action Target Object

For some actions, such as the Edit Chassis action in Example: Example Flex Command Class, 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: Example Flex Command Class, 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 vSphere Web 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 following classes: 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: Example Flex Command Class, 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 the action menu of an object 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 the plugin.xml manifest file of your plug-in module.

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 action menu. In the solution menu object, you define an array of additional ActionMenuItemSpec objects to represent actions, nested menus, and separators.

The following table shows the properties of the ActionMenuItemSpec object that you must set for your nested solution menu extension.

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.

Note 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: Example Solution Menu Extension Definition

The following example shows a Solution Menu extension definition. In the example, the extension adds a nested solution menu to the action menu for VirtualMachine objects.

```
<!-- 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.

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

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 for Flex-Based Action Extensions.
`<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: Nested Solution Menu Extension with Actions

The following example shows a nested solution menu with two action items as child objects.

```
<!-- 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>
```

To add a separator to your nested solution menu, you create a child `ActionMenuItemSpec` object with a `<type>` property which value must be set to the string `separator`.

To add an action to your nested solution menu, you create a child `ActionMenuItemSpec` object with the following properties.

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.

Example: Example Solution Menu Extension with Nested Menu and Separator

The following example shows a 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.

```
<!-- 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.

Following is a list of the properties of the `ActionPriorityGroup` object that you must set in your Prioritization extension.

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	The <string> element that contains the target object type, such as samples:Chassis or VirtualMachine. The Prioritization extension sets the display priority for the action menu of the target object type.
<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 state, you can specify the UID of that action list in the <regionId> parameter to promote actions in that action list.

Example: Prioritizing Items in the Chassis Action Menu

The following example shows a Prioritization action that sets the display priority for actions in the default actions menu for a custom object called Chassis.

```
<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.

The following table lists the properties of the ActionPriorityGroup object that you must set in your Prioritization extension.

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: Prioritizing Items Added to Global Actions Toolbar for Chassis Objects

The following example shows a Prioritization action that sets the display priority for actions in the default actions menu for a custom object called Chassis.

```
<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 Relation Extensions

You can create relation extensions to define relationships between the objects in the vSphere environment and organize the objects in groups under categorized tabs.

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 the categorized tabs for relations in an object workspaceon 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.

Use Cases

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 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

You create a relation extension by defining the objects relationships in the `plugin.xml` manifest file of your user interface plug-in module.

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.

The following table lists the properties that you set for the `ObjectReltaionSetSpec` object.

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 a Relation by Using the RelationSpec Object.

Property	Type	Description
<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 a Relation by Using the RelationSpec Object

You define relations between the vSphere objects by using the RelationSpec object in the plugin.xml manifest file of your vSphere Web Client extension.

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.

The following table lists the properties of the RelationSpec object.

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. The icon appears in the Related Objectsrelated 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 Objectsrelated 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: Example Relation Extension for Chassis Entity

The following example 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.

```
<!-- 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

The home screen appears in the vSphere Web Client main workspace when the user first logs in to the system. You can extend the home screen by adding a shortcut.

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.

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.

Use Cases

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.

Properties of the ShortcutSpec Extension Object

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`.

The following table lists the properties you set for the `com.vmware.vsphere.client.views.ShortcutSpec` object.

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 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 definition.

Example: Example Home Screen Shortcut Extension

The following example shows an 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.

```
<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>
```

In the example, the `<targetViewUid>` property has the value `mySolution.myPlugin.myConsoleApp`. This value matches the extension ID of the example global view extension defined in Adding Global View Extensions. 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.

Creating Object List View Extensions

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 custom vSphere object by using the standard template to create an object workspace.

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 one of the categorized relations tabs in an object 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.

Use Cases

You can extend an existing vSphere object 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 custom 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 Object List View Extensions

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 Chapter 12 List of Extension Points.

Example: Example Extension to Object List View

The following example shows part of an 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.

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.

```
<!-- Define the host name column for the vm object list view -->
<extension id="com.mySolution.myPlugin.vm.columns">
   <extendedPoint>vsphere.core.vm.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>
```

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 Extending the vCenter Object Workspaces.

You can extend a custom object 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 custom object template namespace. The format for a custom object list extension point is typically `<namespace>.list.columns`. For example, the list extension point for a custom object called Chassis might appear as `myExtension.core.Chassis.list.columns`, where the template namespace is `myExtension.core.Chassis`.

Example: Example Extension to Object List View

The following code snippet shows 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.

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.

```
<!-- 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>
```

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 preferences so that the column will be visible in future sessions.

Developing HTML-Based User Interface Extensions

7

You can create extensions to the user interface layer of the vSphere Client that extend the GUI with components specific for your vSphere environment.

The vSphere Client is a Web browser-based application that provides extensible plug-in architecture. 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 HTML application by creating user interface extensions. An HTML user interface plug-in contains one or more extensions, which add HTML GUI elements to the vSphere Client user interface.

This section includes the following topics:

Overview

The vSphere Client provides an extensible plug-in architecture which you can use to create custom solutions for your environment. Use the vSphere Client development kit to develop HTML plug-ins that are compatible with both Web applications.

The vSphere Client development kit provides the following features:

- You can use the same JavaScript APIs provided with the vSphere Web Client SDK 6.0 which ensures that your HTML plug-ins are backward compatible.
- You can use the JavaScript libraries of your choice to develop the user interface components of your extensions.
- You can easily access the vSphere Client server by using REST calls and the Spring Web MVC framework integration.
- You can create extensions to the service layer by using the same Data Service APIs and mechanisms.
- You can examine the sample HTML plug-ins provided with the vSphere Web Client SDK. The samples demonstrate how you can add different extensions to the vSphere Client .

Global View Extensions

In the vSphere Client , you can create global view extensions to create custom solutions for the 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 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 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 Client home screen.

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

Use Cases

You can use global view extensions to create dashboard-style data views or console-style applications.

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.

Console-style applications are displayed in the vSphere Client main content area. For example, the vSphere Client Task Console and Event Console are console-style applications.

Creating Global View Extensions

You create global view extensions by using the `vise.global.views` extension point. To define a global view extension, you must use the `com.vmware.vsphere.client.htmlbridge.HtmlView` class for the `<componentClass>` property of the extension object. For more information about the properties of this HTML object, see Properties of the HtmlView Extension Object.

Since there is no context object for a global view extension, the global view document is opened with a request that contains only the `locale` parameter.

Accessing Data

Each global view extension is an independent HTML element that must communicate with the vSphere environment to retrieve data, or send commands, that the view requires. The vSphere Client development kit includes an MVC framework and a JavaScript library that you can use when creating global view extensions. The JavaScript code can use REST-based Ajax queries to retrieve data from the vSphere environment and call actions on objects in the virtual infrastructure by using the `callActionsController()` function.

You can create a global view extension by using any pattern or framework, but using the vSphere Client development kit MVC framework and the Data Access Manager is a best practice. See Creating Data View Extensions.

Properties of the `HtmlView` Extension Object

The vSphere Client provides the `com.vmware.vsphere.client.htmlbridge.HtmlView` for creating object views and global views for your HTML-based extensions. You must define the `HtmlView` object in the `plugin.xml` manifest file of your HTML view extensions.

The following table describes the properties that you provide for the `com.vmware.vsphere.client.htmlbridge.HtmlView` object class by using the `<root>` element inside the `<object>` element of your extension definition.

Property	Type	Description
`<url>`	string	The relative URL path starting with your plug-in Web context path to the HTML view. If your HTML-based extension displays external context, use an HTTPS URL to the context.
`<showVCenterSelector>`	boolean	The vCenter Server selector allows you to switch between all instances that are connected to the vSphere Client . If an object view or a global view from your extension needs to display data for a particular vCenter Server instance, include the selector by adding the `<showVCenterSelector>` property with value set to `true`. The default value is `false`.
`<dialogTitle>`	string	The name of the view. The property is applicable only for portlet views that are displayed in a separate dialog box.
`<dialogSize>`	integer	The size of the dialog box of a portlet view which is provided in pixels and the property value must have the following format: `<width>,<height>`.
`<dialogIcon>`	string	The icon that you add to the title bar of the portlet view, if necessary.

Property	Type	Description
<scrollPolicy>	string	Indicates whether scrollbars are added to the view. You choose between the following values: yes, no, and auto. You set auto as a value to the property to let the Web browser decide whether scrollbars are needed for your views. The default value is no.

Example: Example HTML-Based Global View Extension

The following example extension definition adds the **VM Summary view** to the VM workspace.

```
<extension id="com.vmware.samples.vspherewssdk.vm.summary">
   <extendedPoint>vsphere.core.vm.summarySectionViews</extendedPoint>
   <object>
      <name>#{summaryView.title}</name>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/vspherewssdk/resources/vm-summary.html</url>
                 <dialogTitle>WSSDK Summary Sample</dialogTitle>
                 <dialogSize>440,400</dialogSize>
            </root>
          </object>
      </componentClass>
   </object>
</extension>
```

Adding a vCenter Server Selector

You can display data for a particular vCenter Server instance for a global view extension by using the vCenter Server selector feature of the HTMLView object class.

To use the vCenter Server selector, you must create a global view by using the com.vmware.vsphere.client.htmlbridge.HtmlView object. In the plugin.xml manifest file, define the properties of the extension object and add the <showVCenterSelector> property with value set to true.

When enabled, the vCenter Server selector displays a drop-down menu that contains all vCenter Server instances that the vSphere Client is connected to. You can easily switch between the data specific for the different vCenter Server systems by using the selector.

The JavaScript code for the extension makes an Ajax GET request that contains the following parameters:

- serviceGuid
- sessionId
- serviceUrl

You can use the vSphere Web Services SDK and pass these parameters to a Java service to access data about a specific vCenter Server instance.

Example: vCenter Server Selector

The sample demonstrates how you can create a global view extension and enable the vCenter Server selector to display specific properties for the selected vCenter Server system.

```
<!-- Global app vCenter view -->
   <extension id="com.vmware.samples.h5.globalview.vcView">
      <extendedPoint>vise.global.views</extendedPoint>
      <object>
         <name>#{app.vcView}</name>
         <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
            <object>
               <root>
                  <url>/vsphere-client/globalview/resources/vcView.html</url>
                  <showVCenterSelector>true</showVCenterSelector>
               </root>
            </object>
         </componentClass>
      </object>
   </extension>
```

Extending the vCenter Object Workspace

The vSphere Client displays a standard object workspace for each type of vSphere object. You can extend the existing object workspaces or create an object workspace for a custom vSphere object.

The object workspace is a collection of data views with a tabbed navigation structure and detailed views with table of contents entries. The workspace for a given vSphere object appears in the vSphere Client main workspace for a selected object from the virtual infrastructure.

Each vSphere object type has **Summary**, **Monitor**, **Configure**, and categorized object relations top-level tabs , and may contain additional detailed views within each tab. You can add extensions to create your own sub-tabs and detailed views within any of the top-level tab views. You can also create new object workspaces with the default top-level tab, sub-tabs and detailed views structure.

Use Cases

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 the standard workspace of the object. You might need to create an extension to the vSphere Client service layer, such as a Data Service adapter, to provide the data for your data view extension. See Chapter 8 Developing Extensions to the Service Layer.

The vSphere Client development kit provides the same set of extension points for each type of object in the vSphere environment as the vSphere Web Client SDK. Each extension point corresponds to a specific view in the object workspace, to which you can add your own extension view.

When you create an object workspace, you use the same XML extension templates that you use for creating Flex-based extensions.

Extending an Existing Object Workspace

The vSphere Client development kit provides the same extension points as the vSphere Web Client SDK. 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.

The extension points for the HTML views use the `com.vmware.vsphere.client.htmlbridge.HtmlView` generic class. For more information about the properties of the `HtmlView` class, see Properties of the HtmlView Extension Object.

To add HTML extensions to the **Monitor** and **Configure** tabs in the vSphere Client, use the following generic extension points:

- `vsphere.core.${objectType}.monitorViews`
- `vsphere.core.${objectType}.manageViews`
- `vsphere.core.${objectType}.manage.settingsViews`
- `${namespace}.monitorViews`
- `${namespace}.manageViews`
- `${namespace}.manage.settingsViews`

For example, if you define an extension that extends the `vsphere.core.vm.manageViews` extension point, your extension appears as an entry in the table of contents under the **Configure** tab in the object workspace for virtual machine objects.

For a complete list of object workspace extension points available for the vSphere Client, see Object Workspace Extension Points.

Types of Data Views

A data view extension appears differently depending on the vSphere object that you specified with the extension point. Data views can appear in the object workspace having one of the following structures.

- Table of contents entry - If you define an extension to a top-level tab, such as **Monitor**, or **Configure**, a data view extension appears as an entry in the table of contents on the left in the object workspace.
- Second-level tab view - If you define an extension to one of the existing second-level tabs, such as inside the **Monitor** tab, a data view extension appears as a view within the second-level tab in the object workspace.
- Portlet - If you define the **Summary** tab extension point, a data view extension appears as a portlet in the 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.

Configure and Monitor Views Extensions

You can extend a vSphere object view under the **Configure** and **Monitor** tabs by using `HtmlView` to specify the generic HTML class that implements the new data view. You must also specify the URL to the HTML source of the data view.

Example: Adding a Host **Monitor** View

Following is an example of how you can add an HTML view to the **Monitor** tab of host objects.

```
<!-- Host monitor HTML view, visible in both Flex client and HTML client -->
<extension id="com.vmware.samples.vspherewssdk.host.monitor">
   <extendedPoint>vsphere.core.host.monitorViews</extendedPoint>
   <object>
      <name>#{monitorHtml.label}</name>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/vspherewssdk/resources/host-monitor.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

The value of the `<url>` property is a relative URL that starts with the Web context path of the plug-in, `/vsphere-client/vspherewssdk`. You must set the same URL without the first slash as a value to the `Web-ContextPath` manifest header of the Web application `MANIFEST.MF` file.

Make sure that the relative URL finishes with the `.html` file extension to enable session authentication for the view.

To display content from another domain in the view, you can use HTTPS URLs. Note that the content is not loaded the first time that the user open the view, unless the domain certificate is already verified. You must not use HTTP URLs because the contemporary Web browsers are designed to block any insecure content that you try to display inside the secure vSphere Client domain.

The `host-monitor.html` document view is opened with a REST request that contains the following parameters:

- `objectId` - The context object ID used to create the REST URL that is used to retrieve object properties. For more information, see the description of the `buildDataUrl(objectId, propList)` function in vSphere Client JavaScript APIs.
- `objectType` - The context object type.
- `locale` - The current locale of the Web browser.

Creating an Object Workspace for a Custom Object

If your vSphere environment contains a custom vSphere object, you can create the object workspace by using the XML templates provided with the vSphere Client development kit. You instantiate the XML template in the `plugin.xml` manifest file.

The vSphere Client development kit includes the same object view template as the vSphere Web Client SDK. 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**, **Configure**, and the tabs for the different related objects groups.

Using the `objectViewTemplate` to Create an Object Workspace

You can create an instance of the standard object view template by using the `<templateInstance>` XML element in your `plugin.xml` manifest file.

To create an instance of the standard object view template, you use the `<templateInstance>` element in your `plugin.xml` manifest file. Inside the `<templateInstance>` element, you define the variables that describe the specific instance of the template. The `<templateInstance>` element can appear anywhere inside the root `<plugin>` element in the `plugin.xml` manifest file, but a best practice is to include the template definition with other extension definitions.

You must add the `<templateId>` element inside the `<templateInstance>` element. This element contains the identifier of the template you want to create. To use the standard object view template included with the vSphere Client development kit, use the `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. Extension points for a custom object workspace are formed by using the following schema for each data view:

```
<namespace_value>.<default_extension_point_name>
```

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

- `<namespace>.gettingStartedViews` - Adds views under the **Getting Started** tab.
- `<namespace>.summaryViews` - Adds views under the **Summary** tab.
- `<namespace>.monitorViews` - Adds an entry in the table of contents under the **Monitor** tab.
- `<namespace>.monitor.performance.overviewViews` - Adds views to the **Overview** performance charts in the **Performance** entry in the table of contents under the **Monitor** tab.
- `<namespace>.monitor.performance.advancedViews` - Adds views to the **Advanced** performance charts in the **Performance** entry in the table of contents under the **Monitor** tab.
- `<namespace>.manageViews` - Adds an entry in the table of contents under the **Configure** tab.

- `<namespace>.manage.settingsViews` - Adds views to the **Settings** node in the table of contents under the **Configure** tab.
- `<namespace>.manage.permissionsViews` - Adds views to the **Permissions** tab.

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 Client displays the object workspace when the user selects an object of the specified type.

Creating the Data Views Within the Template

After you create an object workspace by 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. For more information about defining data view extensions, see Creating Data View Extensions.

Top-level tabs, second-level tabs, and detailed views created by using the standard object view template do not appear in the vSphere 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.chassisb.monitor.performanceViews` extension point for the ChassisB object from the following example, the **Performance** second-level tab under the **Monitor** tab does not appear for ChassisB objects.

Example: Instantiating the Standard Template

The following example instantiates the object view template for a new object called ChassisB. The `namespace` variable has a value of `com.vmware.samples.chassisb`. The vSphere Client uses the value to create an extension point for each data views included in the standard object workspace. The extension point for the **Monitor** tab for the ChassisB object workspace is named `com.vmware.samples.chassisb.monitorViews`. The other extension points in the ChassisB object workspace are named by using the same convention.

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

Creating Extensions to the **Summary** Tab

The vSphere Client development kit contains the same extension points for creating **Summary** tab views as the vSphere Web Client SDK.

The **Summary** tab view contains a header view displayed at the top of the main workspace, and an extension point for the HTML portlet views under the header view. The extension point that you can use for adding views to the **Summary** tab is `<namespace>.summaryViews`. Use the generic `com.vmware.vsphere.client.htmlbridge.HtmlView` class to describe your **Summary** view extension.

You can add also a portlet to the **Summary** tab of a vSphere object by using the `vsphere.core.host.summarySectionViews.html` HTML-specific extension point. This extension point is used in hybrid plug-ins that provide portlet view implementations visible in both Web applications. For more information about how to create hybrid plug-ins, see Hybrid Plug-Ins.

Example: Instantiating a vSphere Object Summary Tab

The following example instantiates the **Summary** tab template for a custom object called ChassisB. The **Summary** view extension uses the .

```
<!-- Chassis summary view -->
<extension id="com.vmware.samples.chassisb.SummaryView">
   <extendedPoint>com.vmware.samples.chassisb.summaryViews</extendedPoint>
   <object>
      <name>#{summary.view}</name>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/chassisb/resources/chassis-summary.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

Adding Portlets to the Summary Tab

After you create a **Summary** tab for your custom object workspace by using the **Summary** tab extension point, you can add portlet data views to the **Summary** tab. You create portlet views at the `<namespace>.summarySectionViews.html` extension point by using the generic `HtmlView` component class. For more information about the properties of this generic class, see Properties of the HtmlView Extension Object.

If you use the default `<namespace>.summarySectionViews` extension point that the **Summary** tab template creates, the HTML-based portlets are displayed inside a separate pop-up modal dialog. The link to the dialog is generated in the portlet area of the **Summary** tab. You can use the mouse to move around the portlets but you cannot change their size.

Starting with vSphere 6.5, the HTML platform of the vSphere Client allows you to creates portlets that display their content directly in the **Summary** tab.

Example: Adding Portlet Views to the Summary Tab

The following example creates a portlet in the **Summary** view of a host.

```
<extension id="com.vmware.samples.vspherewssdk.host.summary2">
   <extendedPoint>vsphere.core.host.summarySectionViews.html</extendedPoint>
   <object>
      <name>#{summaryView.title}</name>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/vspherewssdk/resources/host-summary.html</url>
               <dialogTitle>WSSDK Summary Sample</dialogTitle>
               <dialogSize>440,400</dialogSize>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

Creating Data View Extensions

You can select the methods used by your HTML plug-ins to access data from the vCenter Server system. You can use the Ajax Web development techniques and the Spring Web MVC framework integration, or another solution of your choice.

When you create data view extensions for the vSphere Client user interface layer, follow these general recommendations:

- You do not need to change the Data Adapter services running in the service layer.
- You can use the generic `DataAccessController` Java class provided with each generated plug-in project to handle HTTP JSON `GET` data requests.
- You must access data through the vSphere Client server and avoid calling directly your back end services or database.

Common Data Access Pattern

You can use the pattern demonstrated in the SDK samples to access data from the vCenter Server system from your plug-ins:

- The Ajax `GET` request created in your JavaScript code has the following format:

  ```
  /plugin_context_path/rest/data/properties/objectId?properties=properties-list
  ```

  , where *objectId* is the object ID of the currently selected vSphere object, and *properties-list* is the comma-separated list of properties that must be retrieved for that object.

- The `web.xml` deployment descriptor located in the `WEB-INF` folder of the UI bundle of your plug-in contains the `<servlet-mapping>` element that defines the `/rest/*` URL pattern for invoking the `springServlet` servlet.

  ```
     <servlet-mapping>
        <servlet-name>springServlet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
     </servlet-mapping>
  ```

- The `bundle-context.xml` file located in the `WEB-INF\spring` folder declares the `dataAccessController` bean for the `DataAccessController` class that is available in the Java service bundle of your plug-in.

  ```
  <bean name="dataAccessController"
       class="com.vmware.samples.vspherewssdk.mvc.DataAccessController" />
  ```

- The `DataAccessController` class included in the Java service bundle of your plug-in has the `@RequestMapping` annotation set to process the HTTP JSON `GET` for the `/data` endpoint. The `getProperties()` generic method has the `@RequestMapping` annotation set to the `/properties/{objectId}` value to handle the Ajax `GET` requests created in your JavaScript code.

  ```
      ...
  @Controller
  @RequestMapping(value = "/data", method = RequestMethod.GET)
  public class DataAccessController {

      ...

     @RequestMapping(value = "/properties/{objectId}")
     @ResponseBody
     public Map<String, Object> getProperties(
              @PathVariable("objectId") String encodedObjectId,
              @RequestParam(value = "properties", required = true) String properties)

      ...
  ```

- The `getProperties()` generic method uses the `QueryUtil` class to create a Data Service query for the requested list of vSphere object properties. The query results are returned to the Web browser as JSON data.

```
    ...
     Object ref = getDecodedReference(encodedObjectId);
     String objectId = _objectReferenceService.getUid(ref);

     String[] props = properties.split(",");
     PropertyValue[] pvs = QueryUtil.getProperties(_dataService, ref, props);
     Map<String, Object> propsMap = new HashMap<String, Object>();
     propsMap.put(OBJECT_ID, objectId);
     for (PropertyValue pv : pvs) {
        propsMap.put(pv.propertyName, pv.value);
     }
     return propsMap;
  }
```

- The JavaScript code can display the data returned by the Ajax `GET` request as needed.

Creating Object List View Extensions

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 custom vSphere object by using the standard template to create an object workspace.

The vSphere 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 one of the categorized relations tabs in an object 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 create list view extensions for the vSphere objects in the same way as for the Flex-based plug-ins.

Example: HTML List Extension

Following is an example of how to create a list extension for an HTML plug-in. The values of the `<requestedProperties>` element are retrieved directly from the vCenter Server instance. You must provide implementations of the `DataProviderAdapter` and `PropertyProviderAdapter` that handle the requests and return the data.

```
<extension id="com.vmware.samples.chassisa.list.sampleColumns">
   <!-- This extension point is created by objectViewTemplate above -->
   <extendedPoint>com.vmware.samples.chassisa.list.columns</extendedPoint>
   <object>
      <!-- XML representation of com.vmware.ui.lists.ColumnSetContainer -->
      <items>
         <!-- Chassis name column -->
```

```
            <com.vmware.ui.lists.ColumnContainer>
               <uid>com.vmware.samples.chassisa.column.name</uid>
               <dataInfo>
                  <com.vmware.ui.lists.ColumnDataSourceInfo>
                     <!-- Column header -->
                     <headerText>#{name}</headerText>
                     <!-- Object property whose text value will be displayed (array of 1 element) -->
                     <requestedProperties>
                        <String>name</String>
                     </requestedProperties>
                     <!-- Use sortProperty to allow column to be sorted with header click -->
                     <sortProperty>name</sortProperty>
                     <!-- Use exportProperty to allow column data to be exported -->
                     <exportProperty>name</exportProperty>
                  </com.vmware.ui.lists.ColumnDataSourceInfo>
               </dataInfo>
            </com.vmware.ui.lists.ColumnContainer>

            ...
```

Creating Actions Extensions

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

In the vSphere 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 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.

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

Use Cases

You can extend the vSphere 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 Client service layer. This Java service is used to perform the action operation on the target vSphere object.

Actions Framework Overview

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

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.

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 Client must define an action set. You add each action set extension to a specific extension point in the vSphere 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 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.

Note 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 custom objects.

Defining Individual Actions for HTML-Based Action Extensions

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

HTML-based extensions use delegated actions instead of the command classes used by Flex-based extensions. The `HtmlActionDelegate` object requires a `<className>` property and an `<object>` element that contains only an embedded `<root>` element.

The <className> property must specify the HtmlActionDelegate object for HTML-based extensions.

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

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

The following table lists the properties that you can use in the <root> element.

Property	Type	Description
<actionURL>	string	Identifies the HTML resource to be displayed. The value can be an absolute HTTPS URL or a bundle context path. If the value is a bundle context path, the relative URL must end with the .html extension to enable session authentication. For absolute URLs, the framework does not use session authentication.
<dialogTitle>	string	Specifies the title of the dialog box. Add this property to the <root> element, or the action is treated as headless. Can be localized.
<dialogSize>	string	Indicates the width and height of the dialog box, in pixels, separated by commas.
<dialogIcon>	string	Specifies an optional icon resource for the dialog.
<showCloseButton>	boolean	Hides the top right close button for the dialog. The default value of this property is false.

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.

Invoking Headless HTML Actions

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

- The value of the url parameter has the following form.

  ```
  /vsphere-client/chassis/rest/actions.html?actionUid=<actionUid>
  ```

- The value of the jsonData parameter is a JSON map of parameters passed to the actions controller, or null if no parameters are needed.

Example: HTML-Based Headless Action Extension Definition

The following example shows an 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 Chassis.

```
<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
```

UI Actions

You can implement a UI action that displays a modal dialog in response to a menu click or a toolbar button. You can implement also other types of pop-up dialogs that are specific to an object view or a global view. To open such modal dialogs, invoke the `WEB_PLATFORM.openModalDialog(title, url, width, height, objectId, scrollPolicy, showCloseButton)` JavaScript function. Note that such modal dialogs cannot be nested and the view from which they are invoked disappears temporarily until they are closed.

Example: HTML-Based UI Action Extension Definition

The following example shows an 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 Chassis.

```
<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 table contains the parameters that 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 that is 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 by 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()`.

Handling Actions for HTML-Based Action 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 the 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: Example Java Actions Controller Class

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 vSphere 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.

```
...
   @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();
}
...
```

Handling Locales

Your HTML plug-ins must support the same set of locales as the Flex-based plug-ins. The default locale is the locale that is set by the Web browser of the user, or the English (United States) locale if the vSphere Client does not support the set locale.

In the vSphere Client locales are usually handled on the user interface layer. In some cases, the HTML plug-in must return text from the Java service layer such as the properties of a vSphere object adm error messages.

Handling Resources in the `plugin.xml` Manifest File

The localized resources for your plug-in are located in the `locales` directory of the WAR file. The Ant build scripts for WAR files provided with the vSphere Client development kit compile the plug-in source and the resource files into the `.swf` file format to keep your plug-in compatible with the vSphere Web Client.

The `plugin.xml` manifest file contains the `<resources>` element that you must use to specify the location of plug-in resources such as images and localization data. The `defaultBundle` attribute of the `<plugin>` element specifies the name of the main `.properties` file of the plug-in and is added automatically by the Ant build scripts.

To instruct the vSphere Client to use the locale that your Web browser specifies at runtime, set `{locale}` as a value to the `locale` attribute of the `<resource>` element in the `plugin.xml` manifest file. You must avoid hard-coding a specific locale as a value to the `locale` attribute.

The `plugin.xml` manifest file contains the names of views, dialogs, action menus, icons, and other localizable objects. These strings and icons must be localized and not hard-coded in a particular language. If the string or icon is defined in the main properties file specified with the `defaultBundle` attribute, you must use the `#{RESOURCE_KEY}` syntax for the element and attribute values. If the string or icon is defined in a different `.properties` file, use the `#{BUNDLE_NAME:RESOURCE_KEY}` syntax for the element and attribute values.

Example: Localizing Strings and Icons in the `plugin.xml` Manifest File

The following code snippet demonstrates how you can specify the values for strings and icons that must be localized in the vSphere Client depending on the settings of the Web browser. The main properties file of the plug-in is `locale/en_US/com_vmware_samples_chassisa.properties` which is reflected with the value of the `defaultBundle` attribute.

```
<plugin id="com.vmware.samples.chassisa"
   defaultBundle="com_vmware_samples_chassisa">

   <resources>
      <resource locale="{locale}">
         <module uri="locales/chassisa-{locale}.swf"/>
      </resource>
   </resources>
  ...
   <templateInstance id="com.vmware.samples.lists.allChassis">
      <templateId>vsphere.core.inventorylist.objectCollectionTemplate</templateId>
      <variable name="namespace" value="com.vmware.samples.chassisa_collection"/>
      <variable name="title" value="#{chassisLabel}"/>
      <variable name="icon" value="#{chassis}"/>
  ...
```

The English locales for the `chassisLabel` string and the `chassis` icon are defined in the `com_vmware_samples_chassisa.properties` file in the following way:

```
# ------- String properties --------

chassisLabel = ChassisA
summary.title = Chassis main info
...

# ------------- Images -------------

chassis = Embed("../../assets/images/chassis.png")
localizedImage.url = assets/images/localizedImage-en_US.png
...
```

Handling Resources in the HTML and JavaScript Code

You can directly retrieve and use specific string resources at runtime in your HTML views or JavaScript code.

For example, you can use the `ns.getString()` function defined in the `web-platform.js` JavaScript API. The following example demonstrates how this API is defined in the `web-platform.js` file for the ChassisA sample included in the vSphere Client development kit.

```
       var ns = com_vmware_samples_chassisa;
       // Get a string from the resource bundle defined in plugin.xml
       function getString(key, params) {
          return WEB_PLATFORM.getString("com_vmware_samples_chassisa", key, params);
       }
       ns.getString = getString;
```

The following example demonstrates how you can use localized strings from the plug-in resource files inside the HTML code of your plug-in.

```
   // insert the localized value of "summary.title" in the #title node
   <script>
       $(document).ready(function() {
          var ns = com_vmware_samples_chassisa;
          $("#title").text(ns.getString("summary.title"));
       });
    </script>
</head>
<body>
   <h3 id="title">Chassis main info</h3>
</body>
...
```

Handling Resources at the Service Layer

In some cases your plug-in might return strings from the service layer that must be displayed in the vSphere Client. For example, the service layer can return the properties of a vSphere object that must be displayed in a human-readable format, or an error message that comes from the back end. You must retrieve the current locale of the user and return the translated text for that locale in your Java code.

In case of error messages, your back end server might have the messages localized. In other cases, you can use the standard Java localization APIs and add `.properties` files inside your JAR files. These properties files are used to load the correct strings based on the locale.

Following is an example of how to use the `UserSession` class to access the locale of the current client session.

```
    // see the vsphere-wssdk-service sample for injecting  _userSessionService in your class
    UserSession userSession = _userSessionService.getUserSession();
    String locale = userSession.locale;
    ...
```

Guidelines for Creating Plug-Ins Compatible with the vSphere Client

You can use the plug-in generation scripts provided with the vSphere Client SDK to create a plug-in that is compatible with both Web browser-based applications.

To develop an HTML plug-in, you must first create a plug-in project that has the required by the plug-in resources and directory structure. Use one of the generation scripts that are available in the `tools\Plugin generation scripts` folder under `html-client-sdk`.

After you create the HTML plug-in project, follow these guidelines to ensure that your plug-in is compatible with the vSphere Client and the vSphere Web Client:

- Use relative URLs to set the location to the resources inside your plug-in inside your HTML and JavaScript code. For example, you must avoid adding the `/vsphere-client` root path to the URLs.
- Use the `<plugin_namespace>.webContextPath` variable defined in the `web-platform.js` JavaScript API to specify the Web context path for the Virgo server requests.
- Use the `vsphere-client` root path only inside the `MANIFEST.MF` and `plugin.xml` files.
- Add Cascading Style Sheets (CSS) classes to the `plugin-icons.css` file for the icons that are displayed outside the views, such as Home screen shortcut icons, menu icons, and vSphere objects list icons. See Handling Icons Outside the HTML Views.
- When you add an extension to an existing object menu or a custom object menu, you must define a custom menu extension referencing the `vsphere.core.menus.solutionMenus` extension point in addition to the actions referencing the `vise.actions.sets` extension point. See Defining Menus and Sub-Menus.

Using the Web Context Path in HTML Plug-Ins

Each HTML plug-in is a separate Web application that has a specific context path defined in the `MANIFEST.MF` file of the WAR bundle. The context path of your application specifies where the Web content is hosted and which requests must be handled by your application. For example, the Web context path for the ChassisA sample is defined in the manifest file as follows:

```
Web-ContextPath: vsphere-client/chassisa
```

The root path for resources and data requests for the vSphere Client starts with `ui` while for the vSphere Web Client, the root path starts with `vsphere-client/`. If you leave the `vsphere-client/` root path in the `MANIFEST.MF` and the URLs declared in the `plugin.xml` files of an HTML plug-in, you can still deploy the plug-in on both Web browser-based applications. The root path URL is transformed at runtime.

Handling Icons Outside the HTML Views

External icons are the icons displayed outside the HTML views and handled directly by the vSphere Client. Examples of such icons are the Home view shortcut icons, menu icons, and the vSphere object list icons. If you use the generation scripts or the wizard provided with the vSphere Web Client Tools Eclipse plug-in to generate your HTML plug-in, the `plugin-icons.css` CSS file is added to the plug-in project. The example CSS file contains the definitions of two external icons.

To declare that your plug-in depends on external icons, in the `plugin.xml` manifest file add the `<dependency>` element inside the `<dependencies>` element. The following attributes of the `<dependency>` element contain information about the external icons:

- `type` - The resource type such as `css`.
- `uri` - The URI of the CSS file that contains the external icon declarations.

Following is an example of dependency declaration in the `plugin.xml` file:

```
    <dependencies>
       <!-- Allow HTML Client to display icons in menus, shortcuts, lists -->
       <dependency type="css" uri="myplugin/assets/css/plugin-icons.css" />
    </dependencies>
```

Defining Menus and Sub-Menus

When you add a custom vSphere object menu or extend an existing object menu, you must define each individual action and add a custom solution menu under the existing menu which might include sub-menus and separators. Use the `vise.actions.sets` extension point to define each action, and the `vsphere.core.menus.solutionMenus` extension point to add the custom solution menu.

The following example demonstrates how you can define custom actions for `VirtualMachine` objects and then add custom solution menus under the existing `VirtualMachine` menu.

```
    <extension id="com.vmware.samples.vspherewssdk.vmActionSet">
       <extendedPoint>vise.actions.sets</extendedPoint>
       <object>
          <actions>
             <com.vmware.actionsfw.ActionSpec>
                <!-- UI action: show dialog -->
                <uid>com.vmware.samples.vspherewssdk.myVmAction1</uid>
                <label>#{action1.label}</label>
                <delegate>
<className>com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate</className>
                   <object><root>
                      <!-- execute the action on client-side (html view in a modal dialog) -->
                      <actionUrl>/vsphere-client/vspherewssdk/resources/vm-action-
dialog.html</actionUrl>
                      <dialogTitle>#{action1.label}</dialogTitle>
                      <dialogSize>500,250</dialogSize>
                   </root></object>
```

```
               </delegate>
            </com.vmware.actionsfw.ActionSpec>
      </object>
      <metadata>
         <!-- Filter this extension only for VirtualMachine objects -->
         <objectType>VirtualMachine</objectType>
      </metadata>
   </extension>
            ...

   <extension id="com.vmware.samples.vspherewssdk.vmMenu">
      <extendedPoint>vsphere.core.menus.solutionMenus</extendedPoint>
      <object>
         <!-- <label> is required here because it is an extension to an existing menu -->
         <label>#{solution.label}</label>
         <children>
            <Array>
               <com.vmware.actionsfw.ActionMenuItemSpec>
                  <!-- UI action example -->
                  <type>action</type>
                  <uid>com.vmware.samples.vspherewssdk.myVmAction1</uid>
               </com.vmware.actionsfw.ActionMenuItemSpec>

            ...
       </object>
       <metadata>
         <!-- Filter creates this extension only for VirtualMachine objects -->
         <objectType>VirtualMachine</objectType>
       </metadata>
   </extension>
```

Hybrid Plug-Ins

Hybrid plug-ins are plug-ins that include both Flex and HTML implementations of the user interface extensions. This type of plug-ins can be deployed to the vSphere Client and to the vSphere Web Client but the functionality that they provide works with some limitations.

Use Cases

You can create hybrid plug-ins that either include a Flex implementation of a portlet in an HTML plug-in, or convert an existing Flex plug-in to HTML without removing the Flex views.

To be able to use the same plug-in containing portlets with both Web applications, you can include a Flex and HTML implementations of the same portlet in the `plugin.xml` manifest file. For the Flex extension you must use the `<namespace>.summarySectionViews` extension points and for the HTML portlet you must use the `<namespace>.summarySectionViews.html` extension points. Make sure that you specify different extension IDs for both portlets in the manifest file. For more information about adding portlets to the **Summary** tab, see Creating Extensions to the Summary Tab for HTML portlets and Using the Summary Tab Template for Flex-based portlets.

You can also convert an existing Flex-based plug-in to HTML, but leave some of the Flex extension views. In this case the hybrid plug-in is in a transition mode. The plug-in functions correctly in the vSphere Web Client, but has limited functionality in the vSphere Client that cannot display the Flex-based extensions.

The SDK provides you with two options for combining HTML and Flex views in the same plug-in:

- Adding two different UI implementation bundles in the same plug-in package, one that uses Flex, and one that uses HTML.
- Including both Flex and HTML views in the same plug-in project.

Combining Two UI Bundles in the Same Plug-In

You can use the wizard provided with the vSphere Web Client Tools Eclipse plug-in to create separate Flex and HTML projects. When you build both WAR files that contain the Flex and HTML UI modules, you must reference the bundles in the `plugin-package.xml` manifest file and include them in the same plug-in package ZIP file.

Note Make sure that you specify different values to the `Web-ContextPath` header for each bundle `MANIFEST.MF` manifest file.

Combining the Flex with the HTML Plug-In Projects

If you already have a Flex plug-in project and you have also an HTML project for developing HTML plug-ins, follow these guidelines to combine both projects into one:

- In the Flex project, replace the `war/src/main/webapp/WEB-INF/web.xml` file with the `web.xml` configuration from the HTML project.
- Edit the `webapp/META-INF/MANIFEST.MF` manifest file to include the packages listed in the `Import-Package` header in the HTML project.
- Add the `webapp/WEB-INF/spring/bundle-context.xml` file from the HTML plug-in project to the Flex project, if you use Java controllers.
- Copy all CSS files, images, and JavaScript libraries under the `webapp/assets` folder in the Flex project.
- Copy the HTML and JavaScript code under the `webapp/resources` folder by including the `web-platform.js` JavaScript APIs.
- Add the HTML view extensions to the `webapp/plugin.xml` manifest file.
- Modify your project build script to take into account the new files.

vSphere Client JavaScript APIs

You can access the Web platform APIs through the `WEB_PLATFORM` variable that is defined in the `web-platform.js` generated for each plug-in.

Following is a list of the JavaScript APIs provided with the vSphere Client development kit.

Table 7-1. vSphere Client JavaScript APIs

JavaScript Function	Description
`callActionsController(url, jsonData)`	Use this function to invoke a headless action from within a user interface action dialog. Pass to the function the following parameters: ■ `url` - The value of this parameter has the following form: `/<plugin_context_path>/rest/actions.html?actionUid=<action_Uid>`. ■ `jsonData` - The optional data that is used by the headless action. The target `objectId`s are added automatically to the `url` parameter. For more information about how to create headless actions, see Creating Actions Extensions.
`closeDialog()`	Close a dialog that was opened through a user interface action such as when a form is submitted.
`getActionUid()`	Get the ID of the action that is invoked to open a user interface action or wizard. You can use the returned action ID in the `url` parameter of the `callActionsController(url, jsonData)` function.
`getActionTargets()`	Retrieve a comma-separated list of object IDs selected for an action. This function returns `null` for a global action. Use this function only within a user interface action dialog.
`getObjectId()`	Get the context object ID within an object view or a modal dialog. You can use this object ID to retrieve the vSphere object data from the vCenter Server system.
`getString(bundleName, key, params)`	Retrieve the localized value of a string that is defined in the plug-in resource bundle. Pass to the function the following parameters: ■ `bundleName` - The name of the plug-in resource bundle. ■ `key` - The string resource key. ■ `params` - Optional array of values that must replace the placeholders in the string.
`getVcSelectorInfo()`	Retrieve information that is provided within a global view by using the vCenter Server selector. The retrieved properties are `serviceGuid`, `sessionId`, and `serviceUrl`. For more information about how to enable the vCenter Server selector option, see Adding a vCenter Server Selector.
`getUserSession()`	Retrieve information about the current user session. The function returns the following properties: ■ `userName` ■ `locale` ■ `serversInfo` You can use also the `getUserSession()` method of the `UserSessionService` Java service to retrieve details about the user session.

Table 7-1. vSphere Client JavaScript APIs (Continued)

JavaScript Function	Description
`setGlobalRefreshHandler(callback)`	Invoke this function to refresh your HTML views when the user clicks on the global Refresh button. Pass as argument the function that refreshes your HTML view.
`openModalDialog(title, url, width, height, objectId, scrollPolicy, showCloseButton)`	Invoke this function to open a modal dialog from within an HTML view, such as a wizard. The modal dialog blocks the rest of the user interface and is not limited to the HTML view. Because the dialog is modal, the content of the HTML view disappears temporarily until the dialog is closed. Pass to the function the following parameters: ■ `title` - The title of the dialog. ■ `url` - The URL to the HTML content of the dialog. ■ `width` - The width of the dialog in pixels. ■ `height` - The height of the dialog in pixels. ■ `objectId` - Optional. The ID of the target object. ■ `scrollPolicy` - Optional. The scroll policy of the dialog that can be one of the following values: `yes`, `no`, or `auto`. ■ `showCloseButton` - Optional. The option to display a close button on the modal dialog. You can pass as a parameter to the function one of the following values: `true` or `false`.
`sendModelChangeEvent(objectId, opType)`	Trigger an event that results in the update of the object model when something in the vSphere inventory changes. The function accepts the following parameters: ■ `objectId` - The ID of the vSphere object if the change must be applied to an object view. ■ `opType` - The type of the triggered event. You can pass as parameter one of the following values: `add`, `change`, `delete`, or `relationshipChange`.
`sendNavigationRequest(targetViewId, objectId)`	Invoke this function to open a global view or an object view. The function accepts the following parameters: ■ `targetViewId` - The ID of the extension view. ■ `objectId` - The object ID in case the function is invoked on an object view.
`setDialogTitle(title)`	Change the title of the dialog at runtime. This function overrides the value of the `<dialogTitle>` property defined in the `plugin.xml` manifest file. The function takes as parameter the `title` of the dialog.

JavaScript Utility APIs

Following is a list of the additional utility APIs that are defined in the plug-in namespace variable inside the `web-platform.js` file.

Table 7-2. Utility JavaScript APIs

JavaScript Function	Description
`buildDataUrl(objectId, propList)`	Invoke this function to create the REST URL that is used to retrieve a set of object properties through the `DataAccessController` class. The function accepts the following parameters: ■ `objectId` - The object ID that is passed as a parameter to the view. You can use the `WEB_PLATFORM.getObjectId()` function to retrieve the value of the `objectId` parameter. ■ `propList` - An array of property names that must be retrieved.
`getString(key, params)`	Retrieve the localized value of a `key` defined in the plug-in resource bundle. The function accepts the following parameters: ■ `key` - The string value of the resource key. ■ `params` - Optional array of values that must replace the placeholders in the string.

Mapping the JavaScript to the Flex APIs

The Flex APIs provided with the vSphere Web Client SDK have an equivalent HTML and JavaScript APIs provided with the vSphere Client development kit.

The following table represents the mapping between the Flex and HTML and JavaScript APIs. You can use this information in case you want to start creating HTML plug-ins but you have used only the Flex APIs so far.

Table 7-3. Mapping Between the Flex and HTML APIs

Flex APIs	HTML or JavaScript APIs
`ActionContext`	Actions are invoked by making a REST call with the `targets` parameter that contains a list of the object IDs of the involved vSphere objects.
`ActionInvocationEvent`	Use the `com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate` class.
`ActionMenuItemSpec`	Use this class to define actions extensions in the `plugin.xml` file in the same way as for a Flex-based extension.
`ActionSpec` and `ActionSetSpec`	Use these classes to define actions extensions in the `plugin.xml` file in the same way as for a Flex-based extension.
`BaseProxy`	Use Java controllers to support the REST calls to services running on the Virgo server instead of the `BaseProxy` class. For more information about how you can create controllers, you can take a look at the `DataAccessController` and `ActionsController` classes provided with the HTML sample plug-ins.

Table 7-3. Mapping Between the Flex and HTML APIs (Continued)

Flex APIs	HTML or JavaScript APIs
`ColumnContainer`, `ColumnDataSourceInfo`, `ColumnSetContainer`	These classes are deprecated since vSphere 5.5. You can define object list view extensions by using the `${namespace}.list.columns` extension point in the `plugin.xml` file.
`DataRefreshInvocationEven`	Use the `WEB_PLATFORM.setGlobalRefreshHandler(callback)` function to handle the user-initiated refreshes to the UI.
`Dialogs`	Use your preferred JavaScript library to handle the different types of pop-up dialogs such as error, warning, and confirmation dialogs.
`IconLabelSpec`	Use this class in the `plugin.xml` manifest file in the same way as for a Flex-based extension.
`IContextObjectHolder`	Use the `WEB_PLATFORM.getObjectId()` function to retrieve the context object ID.
`IResourceReference`	Use the object ID in your JavaScript code.
`NavigationRequest`	Use the `WEB_PLATFORM.sendNavigationRequest(targetViewId, objectId)` function to navigate to a specific view or part of the application.
`UserSession`	Use the `WEB_PLATFORM.getUserSession()` function to retrieve details about the currently logged in user.
`Wizard`	This class has no equivalent in the JavaScript APIs.

Developing Extensions to the Service Layer

8

User interface elements in the vSphere Web Client and the vSphere Client` interact with Java services that run in the application server, called the Virgo server. The Java services on the Virgo server communicate with vCenter Server, ESXi hosts, and other data sources within and outside of the vSphere environment.

The principal Java service included in the 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 and user interface layer, such as Flex and HTML data views, send queries to the Data Service for specific objects or attributes. The Data Service processes each query and returns responses.

When you create an extension in the user interface layer that requires data not provided by the Data Service, you must extend the service layer with new providers for the data. This chapter explains how to create Data Service extensions, how to create a custom Java service, how to access data using the vSphere Web Services API or the Data Services interface, and how to import services in a user interface module.

For more information about the relationships between the components in the different layers, see Understanding the vSphere Web Client Architecture and Understanding the vSphere Client Architecture.

This section includes the following topics:

- Understanding the vSphere Web Client Data Service
- Overview of Data Service Queries
- Extending the Data Service with a Data Service Adapter
- Creating a Custom Java Service
- Importing a Service in a User Interface Plug-In Module

Understanding the vSphere Web Client Data Service

The default Data Service provides a stateless, query-based interface to retrieve information about vSphere objects, as defined by the vSphere Web Client 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.

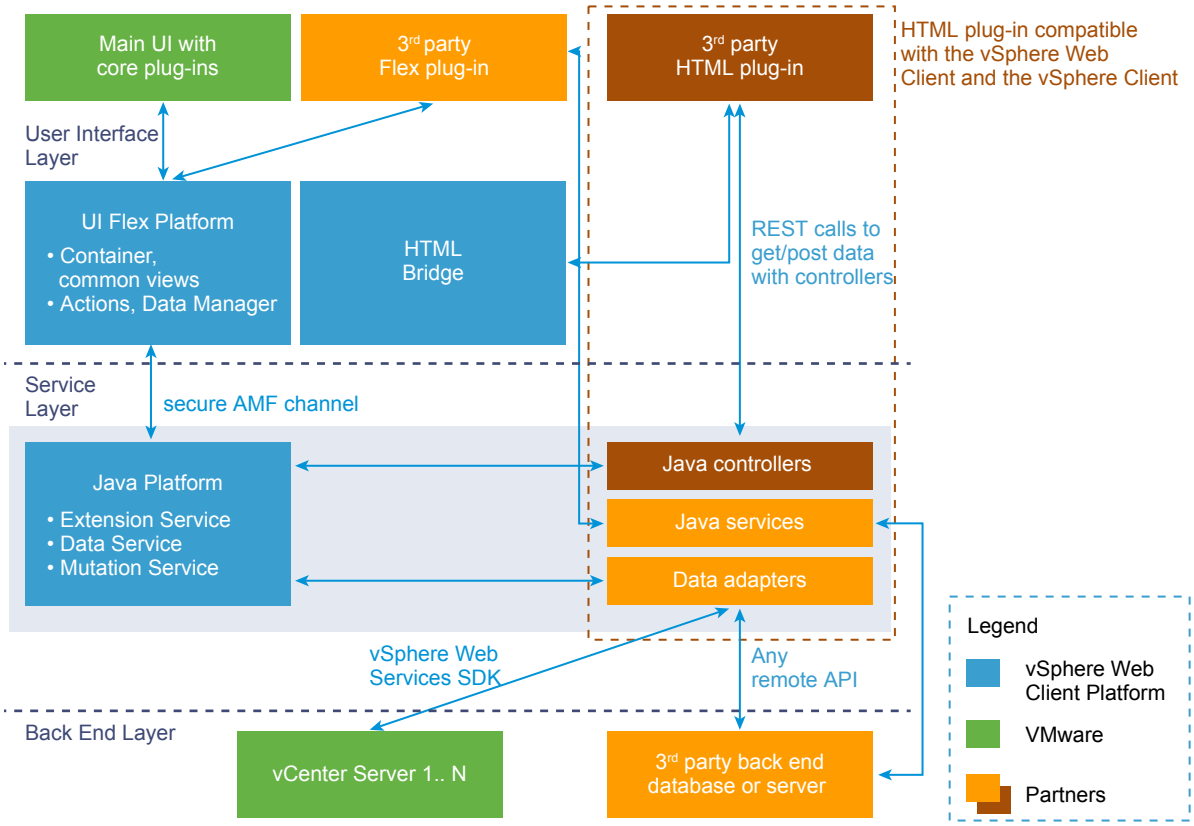
If your vSphere Web Client or vSphere Client extensions require data from a different source, either within vCenter Server or outside vCenter Server, you can extend the Data Service by creating a Data Service Adapter. A Data Service Adapter provides a way for you to use Data Service queries to retrieve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

Extending the Service Layer with Custom Components

The Web Client SDK provides several ways to extend the service layer. Each kind of extension is best suited for certain functions.

- To manage back-end operations in the Virgo service layer, you create custom Java plug-ins, which can be of two kinds:
  - Plug-ins that implement REST services that act on behalf of user interface plug-ins written in HTML.
  - Plug-ins that implement custom RPC interfaces on behalf of Flex proxy components.
- To retrieve data from vCenter Server or from external sources, you create custom data adapters in Java. Your data adapters can be of two kinds:
  - Property Provider Adapters can retrieve data from vSphere managed objects.

- Data Provider Adapters can retrieve data from external sources as well as vSphere managed objects.



Custom Component Types

The following types of custom components belong to or communicate with the Virgo service layer.

HTML UI components	HTML components display the visual components of the vSphere Client interface. You can create custom HTML components to add new features to the user interface.
Flex UI components	Flex user interface components display the visual components of the vSphere Web Client interface. You can create custom Flex components to add new features to the user interface.
Data Service Adapters	Data Service Adapters implement query service interfaces designed by VMware for data requests from user interface components. Property Provider Adapters and Data Provider Adapters are the two kinds of Data Service Adapters.
Data Provider Adapters	Data Provider Adapters implement the `DataProviderAdapter` interface. They respond to requests for data from custom vSphere objects or from objects that are not managed by vSphere.
Property Provider Adapters	Property Provider Adapters implement the `PropertyProviderAdapter` interface. They respond to requests for properties of vSphere objects. Property Provider Adapters cannot provide properties for custom objects.
Custom Java services	Custom Java services provide operations on vSphere managed objects or external data sources. Custom services usually dispatch requests to vCenter Server or to external processes that perform extensive operations.

Interfaces to the Service Layer

Components communicating in the service layer can use the following types of interfaces:

Data Service	The Data Service is an interface accessible to the Data Access Manager or to controller services used by HTML UI components.
Data Access Manager	The Data Access Manager is a Flex library provided by VMware to simplify communications between Flex UI components and the service layer.
PropertyProviderAdapter	Property Provider Adapters implement the `PropertyProviderAdapter` interface of the Data Service. This interface is designed to provide properties of VMware managed objects.
DataProviderAdapter	Data Provider Adapters implement the `DataProviderAdapter` interface of the Data Service. This interface is designed to provide properties of custom objects.
Web Services API	The Web Services API is supported by vCenter Server and ESXi systems. It provides access to vSphere managed objects using an XML SOAP protocol.
Custom Service Interfaces	You can design your own service interfaces to use in custom Java services.

Communications with the Virgo Service Layer

The service layer contains several providers from VMware and you can extend it with custom providers that you create in Java. Custom providers collect and package data used either by custom user interface components or by existing user interface components.

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 Creating Data View Extensions.

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

You can extend the 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 data sources. When you extend the 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 from 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.

Note A best practice is to limit your Java service to dispatching requests from the vSphere Web Client and the vSphere Client, without passing on requests to other services. You can implement extensive or resource-intensive logic on your own external server.

Overview of Data Service Queries

Data Service is an API used to query data in the vSphere Virgo server. You can use the Data Service either from user interface components or from providers in the service layer.

When To Use Data Service Queries

The Data Service is primarily intended for queries from user interface components. However, your service providers also have access to the Data Service.

You can initiate Data Service queries in the Java code of your service providers to fetch data from vCenter Server or from custom service providers. A best practice is to use the vSphere Web Services API to fetch data from vCenter Server, because it is more efficient than Data Services. However, you must use Data Services in the following cases:

- You need to join data from more than one vCenter Server.
- Your query includes properties that are available only from a custom provider.
- Your query includes data objects (complex properties) from vCenter Server, and the client is a UI component from VMware that understands data object encoding.

RequestSpec Data Structure in Data Service Queries

A Data Services client sends a request in the form of a `RequestSpec` object, which contains a list of `QuerySpec` objects.

QuerySpec Structure

The `name` field of a `QuerySpec` is optional. You can assign a name of your choosing, to help you identify the corresponding results. The `name` field is also useful to troubleshoot custom data providers.

A `QuerySpec` also contains a `ResourceSpec` and a `ResultSpec`.

ResourceSpec

The `ResourceSpec` specifies what properties and what objects are to be returned. It contains a list of `PropertySpec` objects and a tree of `Constraint` objects. The `PropertySpec` objects select resources and their properties, while the `Constraint` objects enable you to construct Boolean combinations of conditions to filter the set of resources from which properties are returned.

ResultSpec

The `ResultSpec`, which is optional, enables you to sort the results and to specify a chunk length and a starting index for the `ResultSet`.

The `OrderingCriteria` is a list of `OrderingPropertySpec`. Each list entry specifies the name of a sortable property and whether to sort the values in ascending or descending order.

`OrderingPropertySpec` is a subclass of `PropertySpec`. The subclass adds a `SortType` field.

Note Sorting on custom properties can degrade performance in the client.

PropertySpec

A `PropertySpec` object is used to identify the properties to return in the `ResultSet`, or the properties used for sorting the results. In the latter usage, you can specify an optional sort order for the property by supplying an instance of `OrderingPropertySpec`, which is a subclass of `PropertySpec`. A `PropertySpec` is required in the `ResourceSpec`, but `OrderingPropertySpec` is optional.

A `PropertySpec` begins with a `type` field which contains the name of a resource type. This is typically the URI of a custom resource type, or the name of a managed object type in the case of a query that joins data across vCenter Servers. For example, to request properties of a `VirtualMachine` managed object, you must set the `type` field of a `PropertySpec` object to `"VirtualMachine"`.

A `PropertySpec` contains an array of strings identifying properties to be returned in the `ResultSet`. To identify nested properties, such as properties of nested data objects, use a period as delimiter. For example, the name of a virtual machine config file is a property of the `files` data object, which is a property of the `config` data object, which is a property of the `VirtualMachine` managed object, so you identify the chosen property with the string `"config.files.vmPathName"`.

To access properties of related resources or managed objects, such as the name of the host on which a virtual machine is currently running, use a `Constraint` object to do a join operation between the two managed object types.

Note The `relation` field and the `ParameterSpec` array contained in the `propertySpec` object are reserved for internal use.

Constraint

`Constraint` objects enable you to specify arbitrary Boolean expressions that filter the results of your query. You can limit the results by placing conditions on property values and object identities. Your query must include a `Constraint` object.

`Constraint` is an abstract class with four subclasses. You can supply a simple constraint of object identity or property value by using an `ObjectIdentityConstraint` object or a `PropertyConstraint` object. You can use a `RelationalConstraint` object to join data across resource types.

You can use a `CompositeConstraint` wherever a `Constraint` object is allowed. A `CompositeConstraint` enables you to combine a list of other constraint objects, joined by a Boolean operator. You can nest a `CompositeConstraint` within another `CompositeConstraint`, which enables you to create arbitrarily complex Boolean expressions.

A query can contain 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 a managed object type or any custom type that implements the `IResourceReference` interface. The identifier in this constraint includes the server GUID.
- `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`. This constraint is not bound to a specific server, and can be used to retrieve results from all vCenter Servers known to the client.
- `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. The identifier in this constraint includes the server GUID.
- `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.

Each constraint operates relative to a resource type that you specify in its `targetType` field. For instance, if you want to query the names of all virtual machines running on a given host, one way is to create a `PropertyConstraint` that specifies a `targetType` of `"HostSystem"` and a value for the `name` property, then nest that `PropertyConstraint` in the `Constraint` field of a `RelationalConstraint` that specifies a `targetType` of `"VirtualMachine"` and a `relation` field of `"runtime.host"`.

ResultSet Data Structure in Data Service Queries

The response to a `RequestSpec` is a list of `ResultSet` objects. Each `ResultSet` corresponds to a `QuerySpec` object in the `RequestSpec`, with a one-to-one mapping.

The `queryName` field of a `ResultSet` is used to identify the `QuerySpec` that corresponds to the `ResultSet`. If you assigned a name to a query in the `RequestSpec`, the Response contains a `ResultSet` with a matching value in its `queryName` field. If you submitted a `QuerySpec` without a name, the corresponding `ResultSet` has an empty string in the `queryName` field. A best practice is to assign a unique name to each `QuerySpec` whenever you submit a request that contains more than one query.

When you process a `ResultSet`, first check the `error` field. If the `error` is non-empty, the query failed, and the `queryName` field has a valid value but other fields have indeterminate values. If the `error` is empty, the other fields are meaningful.

The `totalMatchedObjectCount` tells you the number of items the query can return. If the query did not specify a chunk size in the `maxResultCount` field, then `totalMatchedObjectCount` is the size of the `ResultSet.items` list. If the query did specify a chunk size, then the `items` list size is the minimum of `QuerySpec.maxResultCount` and `ResultSet.totalMatchedObjectCount – QuerySpec.offset`.

The data payload is `ResultSet.items`, which is a list of `ResourceItem` objects. Each `ResourceItem` object contains a single `resourceObject` field, which holds the identifier of the resource whose properties are returned in this `ResourceItem`. The `ResultItem.properties` field contains a list of name-value pairs for properties requested by the `QuerySpec`.

Extending the Data Service with a Data Service Adapter

You extend the Data Service by creating a Data Service Adapter to provide data to the components in your user interface extensions that require data that is not available through the Data Service.

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 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.
- 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.

- 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 Chassis. You might also use a Data Provider Adapter to display data in the vSphere Web Client or the vSphere 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 `com.vmware.vise.data.query` package.

After you create the adapter service, you must add the adapter service to the Virgo Server framework and register the adapter with the Data Service. You register an adapter by using the `DataServiceExtensionRegistry` service, typically within your adapter constructor method. See Registering a Property Provider Adapter and Registering a Data Provider Adapter.

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.

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

A query target is a resource type for which your `getData()` method must retrieve properties. A `QuerySpec` can specify a number of targets within its `ResourceSpec`, by including an array of objects of type `com.vmware.data.query.PropertySpec`. Each target type is represented as a string in the field `ResourceSpec.PropertySpec[x].type`.

Your `getData()` method can determine what information it must retrieve by using the values in the `PropertySpec` objects. If the target is a VMware managed object, the value of the string is the name of the managed object type. For custom objects, see Resolving a Custom Target Object.

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 a managed object type or any custom type that implements the `IResourceReference` interface. The identifier in this constraint includes the server GUID.
- `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`. This constraint is not bound to a specific server, and can be used to retrieve results from all vCenter Servers known to the client.
- `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. The identifier in this constraint includes the server GUID.
- `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.

Note When a Data Service query requests a vSphere data object as a whole, rather than its properties, the response contains the data object in an unsupported format that VMware user interface elements understand. If your provider needs to use the Data Service to request a data object on behalf of a client, your provider should copy the data object from its query results into the result set that your provider is building in response to the client, without doing any kind of processing on the data object portion of the results.

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.

Note You may not register a provider for an existing VMware property or object type. For example, if your solution needs to identify a host by an alternate name, you may create an adapter to implement a property such as `alt_name`, but it may not modify the original `name` property.

`PropertyProviderAdapter` Interface

A Property Provider Adapter must implement the `PropertyProviderAdapter` interface of the `com.vmware.vise.data.query` 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 the `getProperties()` method of your adapter 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 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 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 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

The following example shows a Property Provider Adapter class. The class constructor method registers the adapter with the Data Service.

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`.

```
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
    ...
  }
}
```

Data Provider Adapters

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.

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.

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.

Note You may not register a provider for an existing VMware property or object type. For example, if your solution needs to identify a host by an alternate name, you may create an adapter to implement a property such as `alt_name`, but it may not modify the original `name` property.

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

- You must be able to represent the external data by using the same object and property model as the 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.
- In general, you should not use a Data Provider Adapter to add properties to an existing resource. if you register a Data Provider Adapter to service a request for any properties of the resource, your provider must be able to provide all properties for the resource. A best practice is to use a Property Provider Adapter to add properties to an existing resource.

`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 the `getData()` method of your adapter 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.

Resolving a Custom Target Object

A custom target object for a query is identified by a Uniform Resource Identifiers (URI) string, which is a unique identifier for a specific custom object type. 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 within the service layer of the vSphere Web Client and vSphere Client. You must use the 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.

Data Provider Adapter Example 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 and the vSphere Client detect new OSGi bundles as they are added and register 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 Java service of your adapter as a Java Bean in the `bundle-context.xml` file. You can find the `bundle-context.xml` file in the `src/main/resources/META-INF/spring` folder of your plug-in module.

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 the `src/main/resources/META-INF/spring` folder of your plug-in module.

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 SDK that your Data Provider Adapter imports. You add the imported packages to the `Import-Package` manifest header of the `MANIFEST.MF` file.

In Data Provider Adapter Example, the example Data Provider Adapter imports the packages `com.vmware.vise.data.uri` and `com.vmware.data.query`. The packages are listed by using the `Import-Package` OSGi manifest header in the `MANIFEST.MF` file.

```
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 and the vSphere Client use these annotations to route queries for the specific types to the correct adapters. You use the `@type` annotation to define the vSphere object type for which the adapter processes queries.

For example, if you have a custom object of type `WhatsIt`, you annotate the class in the following way.

```
@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 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 Data Service. To include the Data Service as an argument to your Data Provider Adapter class constructor, you add the following element to the `bundle-context-osgi.xml` file of your service.

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

Note Making Data Service queries from within a Data Service provider can impact the performance of your provider. A best practice is to use the vSphere Web Services API to fetch data from vCenter Server, because it is more efficient than Data Services.

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 the `bundle-context-osgi.xml` file of your service.

```
<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 the `bundle-context-osgi.xml` file of your service.

```
<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 the `bundle-context-osgi.xml` file of your service.

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

If you pass OSGi services to your Data Provider Adapter 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.

For each service your Data Provider Adapter includes, you must add a `<constructor-arg>` element to the Bean definition of your adapter. 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

The following example 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 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 SDK.

Example: Example Data Provider Adapter Class

The `getData()` method is called by the 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.

```
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;
  }
}
```

Creating a Custom Java Service

You can extend the Java 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 or the vSphere 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. Inside the JAR bundle, you must add an XML configuration file that declares all of the Java objects that the service adds to the Virgo server framework. The Virgo server uses Spring as the application server framework.

Make Java Services Available to the UI Components in the vSphere Web Client and the vSphere Client

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

Procedure

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.

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 data requests.

5 Establish a communication between your service and the user interface layer.

   - 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 by 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.

The following example shows a basic interface class and an implementation class.

```
package com.vmware.myService;

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

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

Persisting Data from Your Plug-Ins to the vCenter Server Appliance and the vCenter Server System

You can store persistently small data files such as configuration changes on the vCenter Server Appliance and the vCenter Server system.

You can use the default data directory on the vCenter Server Appliance and the vCenter Server on Windows for storing small files. If the data you want to persist is complex or requires more storage space, you must use a separate back end server or database.

For more information, you can refer to the `GlobalServiceImpl.getGlobalViewDataFolder()` method from the Global View sample. The sample code demonstrates how you can use your Java services to create folders for storing the data persistently on the vCenter Server Appliance and vCenter Server instances.

Note Make sure that the directories that you use for storing your data are accessible by the processes running on the Virgo server.

Packaging and Exposing the Service

To make your Java service available for use with the vSphere Web Client and the vSphere 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 shown in the following example.

```
<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 by 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 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 Creating Data View Extensions.

Example: Example ActionScript Proxy Class

The following example 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.

```
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 =
       ServiceUtil.getDefaultChannelUri(GlobalviewModule.contextPath);

    /**
    * 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. The proxy constructor sets the destination parameter to the service ID `myService`, as defined in the `/war/src/main/webapp/WEB-INF/spring/bundle-context.xml` WAR bundle configuration file.

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`. The proxy class uses the `callService` method to call the echo method in the Java service.

Creating and Deploying Plug-In Packages

9

In the vSphere Web Client and vSphere 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 and the vSphere Client extensibility frameworks can perform live hot deployment of the plug-in modules in a package.

This section includes the following topics:

Plug-In Package Overview

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.

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 third 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.

XML Elements of the Plug-In 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.

The following example shows an example of a `plugin-package.xml` manifest 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="6.0.0" />
</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.

Attribute Name	Description
`id`	The unique package identifier that you define. 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.

Attribute Name	Description
id	The unique identifier of the package that your package depends on.
version	The version number of the package that your package depends 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.

Important If your vSphere Web Client extension depends on plug-in packages with specific versions and might not be compatible with later versions of these plug-in packages, make sure that you define correctly the dependencies by using the match parameter. Otherwise, your plug-in package will not work and might cause errors in the vSphere Web Client.

For example, you can use the following lines in the manifest file of your plug-in package to define the minimum and maximum supported versions of the vSphere Web Client:

```
...
<dependencies>
   <pluginPackage id="com.vmware.vsphere.client" version="5.5.0" match="greaterOrEqual" />
   <pluginPackage id="com.vmware.vsphere.client" version="6.0.0" match="lessThan" />
</dependencies>
...
```

If your plug-in package is only compatible with a specific version of the vSphere Web Client, you must use the *equal* value of the match attribute to specify the version. This ensures that when the vSphere Web Client is upgraded, your plug-in package will not be deployed, and will not cause any errors.

<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 id attribute of the <bundle> element 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.

Plug-Ins Compatibility Matrix

In vSphere 6.5, you can prevent plug-ins that are not compatible with the current version of the vSphere Client and the vSphere Web Client from deploying in your environment.

You can use the compatibility matrix feature in the following use cases:

- You can mark as incompatible plug-ins that have compatibility issues with the current version of the vSphere Client or the vSphere Web Client. For example, the plugin-package.xml manifest file might contain incorrect definitions of the plug-in dependencies. For more information, see <dependencies> Element.
- You can mark as compatible in your environment, plug-ins that are by default considered as incompatible for the current version of the vSphere Client or the vSphere Web Client.
- You can blacklist several plug-ins that you suspect are causing issues in your environment and in this way you can narrow down the list of problematic plug-ins.

To disable and enable plug-ins from deploying on the vSphere Client or the vSphere Web Client, you must edit the compatibility-matrix.xml configuration file. You can locate the file in one of the following directories:

Table 9-1. Location of the compatibility-matrix.xml Configuration File

vSphere Environment Setup	vSphere Client	vSphere Web Client
vCenter Server for Windows	C:\ProgramData\VMware\vCenterServer\cfg\vsphere-ui	C:\ProgramData\VMware\vCenterServer\cfg\vsphere-client
vCenter Server Appliance	/etc/vmware/vsphere-ui	/etc/vmware/vsphere-client

You can define your compatibility rules by inserting a <PluginPackage> element inside the <pluginsCompatibility> element for each plug-in. The <PluginPackage> element contains the following attributes:

- id - The ID of the plug-in that you want to prevent from deploying or explicitly declare as compatible with current version of the vSphere Client or the vSphere Web Client.
- version - The version of the plug-in that you want to prevent from loading.
- status - The rule that you want to be applied for the plug-in. The value of this attribute can be either incompatible or compatible.

Example: Plug-Ins Compatibility Matrix

The following example of the `compatibility-matrix.xml` file content demonstrates how you can control the plug-ins that are deployed on your vSphere Client or vSphere Web Client.

```
<!--
    This file lets you define a WHITE LIST and a BLACK LIST of plugins to control your own setup.
    It overrides the internal black and white lists that are hard-coded in this release.
-->
<Matrix>
    <pluginsCompatibility>
        <!--
            WHITE LIST:
            Add this to enable all plugins whose plugin-package id is com.acme.example.myplugin:
                <PluginPackage id="com.acme.myplugin" status="compatible"/>
            Or this to specify all versions greater or equal to 2.1.0:
                <PluginPackage id="com.acme.myplugin" version=[2.1.0,] status="compatible"/>
            Or this to enable all plugins starting with com.acme:
                <PluginPackage id="com.acme.*" status="compatible"/>
        -->

        <!--
            BLACK LIST:
            Add this to disable a plugin whose plugin-package id is com.acme.example.myplugin:
                <PluginPackage id="com.acme.myplugin" status="incompatible"/>
         -->

    </pluginsCompatibility>
</Matrix>
```

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 a vCenter Server instance, the vSphere Web Client application server queries vCenter Server for a list of all available plug-in packages that are registered as vCenter Server extensions. Plug-in packages that are not present on the vSphere Web Client application server are downloaded and installed.

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 registered vCenter Server extension, the registered vCenter Server extension replaces the older plug-in package.

Deploying a Plug-In Package From a Remote Server

The plug-in package ZIP file that represents a vSphere Web Client extension is typically hosted on a remote Web server. A vCenter Server extension can reference a remotely hosted plug-in package by specifying the Web sever URL in the extension definition. If you register such an extension with a vCenter Server instance, the plug-in package ZIP file is downloaded from the remote URL.

The vSphere Web Client establishes a secure HTTPS connection with the remote Web server that hosts the plug-in packages. Starting with vSphere 6.0 Update 2, you can configure the TLS protocol versions for the vCenter Server Service, VMware vSphere Web Client Service, VMware Directory Service, Security Token Service and Syslog Collector Service. You can choose between the following configurations:

- TLS protocol version 1.2
- TLS protocols versions 1.1 and 1.2
- TLS protocols versions 1.0, 1.1 and 1.2

In vSphere 6.0 Update 2, you have all three TLS protocol versions enabled by default. Note that the TLS protocol versions that you configure for the vCenter Server Service must be the same as the protocol versions for all other services.

Note Make sure that the Web server that hosts your vSphere Web Client plug-ins supports the same TLS protocol versions that are configured for the vSphere services. If this requirement is not met, the vSphere Web Client fails to download the extension plug-ins.

Register 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`.

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 programmatically, and register that data object using the vSphere API. You can use the `ExtensionManager.registerExtension()` method to register the data object.
- Use the Managed Object Browser (MOB) application for your vCenter Server system. For more information about how to use the MOB to register your extension, see the procedure bellow.

Procedure

1. Create the `vim.Extension` data object in an XML file, and place that file in a file system available to the vSphere Web Client.
2. In a Web browser, navigate to the Managed Object Browser of your vCenter Server.

   `https://<vcenter_server_ip_address_or_fqdn>/mob/?moid=ExtensionManager`
3. Log in with your vCenter Server credentials.

4 On the `ManagedObjectReference:ExtensionManager` page, under Methods, click **RegisterExtension**.

5 On the `void RegisterExtension` page, in the text box inside the Value column, enter the XML data of your vSphere Web Client extension.

6 Click **Invoke Method** to register the extension.

For an example about how to define your `Extension` data object, see Creating the vCenter Server Extension Data Object.

What to do next

Check whether your extension is registered successfully with the vCenter Server instance by using one of the following approaches:

- In the vSphere Web Client, go to **Administration** and under **Solutions**, select **Client Plug-Ins** and click **Check for New Plug-Ins**.
- Log out and log in again to the vSphere Web Client. The vSphere Web Client checks for new plug-ins for each new user session.

  Note If you try to upgrade an existing plug-in with a new version and you do not follow the best practices and recommendations for developing vSphere Web Client plug-ins, you might need to restart the vSphere Web Client service to see your plug-in. This additional step is required in the following two cases:

  - The new version of your plug-in has a different plug-in ID.
  - The `plugin-package.xml` manifest file and the vCenter Server extension data object have different plug-in IDs or versions specified.

For more information about verifying the deployment of your plug-in package, see Verifying Your Plug-In Package Deployment.

Creating the vCenter Server `Extension` Data Object

Regardless of the registration method you choose, you must set the properties of the `Extension` data object.

You use the following properties to define the `Extension` data object.

Property Name	Description
<key>	The plug-in package ID that you defined in your plug-in package manifest file, plugin-package.xml file.
<client>	This property must contain one ExtensionClientInfo data object, with the following properties.

Property Name	Description
<version>	The dot-separated version number of the plug-in package that is defined in plugin-package.xml.
<type>	Must be set to vsphere-client-serenity.
<url>	The location of the plug-in package ZIP file that is accessible on a Web server.
<server>	Optional. If the URL uses HTTPS, you must define a <server> property in your extension data object. The <server> property must contain the SHA1 thumbprint for the server where your plug-in package ZIP file is stored. For information about the <server> property, see the following example.

Example: Example vim.Extension XML Definition

The following example shows an example Extension object defined in an XML file.

```
<extension>
   <description>
      <label>My plugin</label>
      <summary>My first vSphere Client plugin</summary>
   </description>
   <key>com.mycompany.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>
</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.

The following example shows an example <server> property.

```
<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

Once you register your plug-in package extension, the plug-in is downloaded and deployed on vSphere Web Client startup. You can verify that the deployment procedure is successful by using the log files of the Virgo server.

You can verify that your plug-in package is deployed correctly by searching the log file on the vSphere Web Client Virgo server for your plug-in package ID. If the package is deployed correctly, the plug-in package ID is included in a message about a successful package deployment.

On startup, the vSphere Web Client caches the downloaded plug-in package in one of the following directories on the Virgo server.

Table 9-2. Locations for the Cached Downloaded Plug-In Packages

Virgo Location	Cache Location
vCenter Server for Windows	`C:\ProgramData\VMware\vCenterServer\cfg\vsphere-client\vc-packages\vsphere-client-serenity\`
vCenter Server Appliance	`/etc/vmware/vsphere-client/vc-packages/vsphere-client-serenity/`
Windows OS	`%PROGRAMFILES%/VMware/vSphere Web Client/vc-packages/vsphere-client-serenity/`
Mac OS	`/var/lib/vmware/vsphere-client/vsphere-client/vc-packages/vsphere-client-serenity/`

Note If you want to update the content of a plug-in package, you must register the plug-in with a new version or remove the plug-in package from the cache.

Unregister a Plug-In Package

You can unregister a plug-in package that you previously registered with a vCenter Server instance.

You can unregister the extension in the following ways:

- Use the vSphere API and invoke the `unregisterExtension()` method of the `ExtensionManager` managed object to unregister programmatically your extension.
- Use the vCenter Managed Object Browser (MOB) interface in your Web browser to delete manually the extension. For more information about how to use the MOB to unregister your extension, see the procedure bellow.

Procedure

1. In a Web browser, navigate to the Managed Object Browser of your vCenter Server.

   `https://<vcenter_server_ip_address_or_fqdn>/mob/?moid=ExtensionManager`

2. Log in with your vCenter Server credentials.
3. On the `ManagedObjectReference:ExtensionManager` page, under Methods, click **UnregisterExtension**
4. On the `void UnregisterExtension` page, in the text box inside the Value column, enter the value for the `key` property of the `Extension` data object of your vSphere Web Client extension.
5. Click **Invoke Method** to unregister 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 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 behavior is a known issue, and a workaround is to restart the Virgo server.

10 Sample Plug-Ins Overview

The vSphere Web Client and the vSphere Client SDKs provide samples for both Web browser-based applications. You can use the scripts provided in both SDKs to rebuild and run the samples.

HTML Sample Plug-Ins

When you download the vSphere Web Client SDK, you can find the HTML sample plug-ins in the `SDK\vsphere-client-sdk\html-client-sdk\samples` directory.

Table 10-1. HTML Sample Plug-Ins

Name	Description
ChassisA Client	The ChassisA Client sample demonstrates how you can use extension templates to create a standard vSphere object workspace for custom objects. The sample code is provided within the `chassisA-html` and `chassisA-service` folders. The `chassisA-html` folder contains the UI extensions that add Chassis objects to the vSphere inventory. The sample uses a custom jQuery UI CSS which is similar to the theme of the vSphere Web Client. The sample also demonstrates the usage of internationalization for different locales. The `chassisA-service` folder contains the implementation of the `DataProviderAdapter` that handles all UI requests about the `samples:ChassisA` object type. The ChassisA Client sample extends the following extension points: ■ `vise.navigator.nodespecs` ■ `${namespace}.gettingStartedViews` ■ `${namespace}.summaryViews` ■ `${namespace}.monitorViews` ■ `${namespace}.manageViews` ■ `${namespace}.list.columns` ■ `vise.actions.sets` ■ `vsphere.core.menus.solutionMenus` ■ `vsphere.core.objectTypes` ■ `vmware.prioritization.listActions` ■ `vise.inventory.representationspecs` ■ `vsphere.core.host.summarySectionViews` The sample uses the following extension templates: ■ `vsphere.core.inventorylist.objectCollectionTemplate` ■ `vsphere.core.inventory.objectViewTemplate`
ChassisB Client	The ChassisB Client sample demonstrates how you can use extension templates to add custom Rack objects, relations between Chassis and Rack objects, and also between Chassis and vSphere hosts. The sample code is provided within the `chassisB-html` and `chassisB-service` folders. The `chassisB-html` folder contains the UI extensions that add the Chassis and Rack objects to the vSphere inventory. The sample uses a custom jQuery UI CSS which you can modify if you want. The `chassisB-service` folder contains the implementation of the `DataProviderAdapter` interface to define the `ChassisRackVSphereDataAdapter` to the Data Service. The data adapter handles all UI requests about the `HostSystem`,`samples:Rack`, and `samples:ChassisB` object types. The `mvc` folder contains controllers that handle the HTML requests and dispatch actions and queries. The ChassisB Client sample extends the following extension points: ■ `vise.navigator.nodespecs` ■ `${namespace}.summaryViews`

Table 10-1. HTML Sample Plug-Ins (Continued)

Name	Description
	■ `${namespace}.monitorViews` ■ `${namespace}.manageViews` ■ `${namespace}.list.columns` ■ `vise.relateditems.specs` ■ `vise.actions.sets` ■ `vsphere.core.objectTypes` ■ `vmware.prioritization.listActions` ■ `vmware.prioritization.actions` ■ `vise.inventory.representationspecs` ■ `vsphere.core.menus.solutionMenus` The sample uses the following extension templates: ■ `vsphere.core.inventory.objectViewTemplate` ■ `vsphere.core.inventorylist.objectCollectionTemplate`

Table 10-1. HTML Sample Plug-Ins (Continued)

Name	Description
Global View	The Global View sample demonstrates how to add global views to the vSphere Client and how to make Java service calls. The sample also shows how you can store data in a local file and how to use the vCenter Server Selector functionality. The sample code is provided within the `globalview-html` and the `globalview-html-service` folders. The `globalview-html` folder contains the global views extensions created by using the jQuery JavaScript library. The `globalview-html-service` folder contains the `GlobalService` and `EchoService` interfaces and their implementations. The sample shows how you can expose OSGi services and make them available to other bundles on the Virgo server. The Global View sample extends the following extension points: ■ `vise.global.views` ■ `vise.home.shortcuts` ■ `vise.navigator.nodespecs`
vSphere WSSDK	The vSphere WSSDK sample demonstrates how to use the vSphere Web Services SDK to access the vSphere management objects which may not be accessible through the Data Service. The sample shows how to create hybrid plug-ins that use the HTML views to open specific Flex views in the vSphere Web Client. The sample code id provided within the `vsphere-wssdk-html` and `vsphere-wssdk-service` folders. The `vsphere-wssdk-html` folder contains the HTML and Flex **Summary** portlets and **Monitor** views extensions for both virtual machines and hosts. The **Monitor** view of the host also demonstrates how to display data from different objects in the same view . The `vsphere-wssdk-service` folder contains the `DataProviderImpl` property provider adapter that uses the vSphere Web Services SDK to access data from the vCenter Server system. The sample also demonstrates how to handle multiple properties and build complex queries. The vSphere WSSDK sample extends the following extension points: ■ `vsphere.core.${objectType}.monitorViews` ■ `vsphere.core.${objectType}.monitorViews.html` ■ `vsphere.core.${objectType}.summarySectionViews` ■ `vsphere.core.${objectType}.summarySectionViews.html` ■ `vsphere.core.menus.solutionMenus` ■ `vise.actions.sets`

Best Practices for Developing Extensions for the vSphere Web Client

11

You can improve your extension solutions by understanding the process of extending the user interface layer and service layer of the vSphere Web Client, and packaging and deploying your extension solutions. Follow best practices to ensure optimal performance and scalability, and to improve the security of your vSphere Web Client extensions.

This section includes the following topics:

Best Practices for Creating Plug-In Packages

To meet the requirements of your virtual environment, you must extend the capabilities of the vSphere Web Client by creating plug-in modules. Depending on your extension solution, you can extend the user interface layer and the service layer of the vSphere Web Client.

Incorrect structure of the plug-in package leads to deployment errors. To avoid these errors, consider the following best practices when creating your plug-in packages.

- Use the generation tools provided with the vSphere Web Client SDK to develop your vSphere Web Client extensions and create plug-in packages.
- Verify that the structure of the plug-in package is as follows:
  - `plugin-package.xml` - The file describes general information about the plug-in package, the deployment order of the plug-in modules, and any dependencies upon other plug-in packages.
  - `plugins` folder - The folder contains one or more JAR and WAR files that represent the user interface and Java services components. Limit the number of third-party libraries that you add to this folder.

- To avoid installation errors, make sure that all third-party libraries that you use are added inside the JAR and WAR files of the plug-in package and not inside the `plugins` folder. If you add third-party libraries to the `plugins` folder, the bundles must be OSGi-compliant. Because the vSphere Web Client resides on the Virgo Web Server, which is based on the SpringSource dm Server and is built on top of the Equinox OSGi framework, third-party libraries must be packaged as OSGi bundles. OSGi bundles must include an OSGi manifest file that contains correct and thorough OSGi metadata.
- To avoid deployment errors generated by the Virgo server, make sure that you do not include third-party libraries that are already available on the server. You can navigate to the `vsphere-client-sdk/server/repository/usr` and `vsphere-client-sdk/server/pickup` directories to view the available libraries.
- If your plug-in package contains both user interface and Java service components, place the Java service components before the user interface components in the plug-in package manifest file. Use the `<bundlesOrder>` element to specify the order in which the bundles are deployed to the vSphere Web Client.
- For best performance, when designing your vSphere Web Client extension, limit the number of files included in the `plugins` folder of your plug-in package. Ideally, your plug-in package must contain only one WAR file, which contains the user interface plug-in modules, and one JAR file, which contains the Java service plug-in modules. Fragmenting your code into many bundles might significantly increase the deployment time and memory consumption.
- To avoid compatibility issues in case your plug-in package depends on other plug-in packages with specific versions, make sure that you define correctly the plug-in dependencies by using the `match` parameter of the `dependencies` element in your `plugin-package.xml` manifest file. You must specify not only the minimum required plug-in package version compatible with your plug-in, but also the maximum one. Otherwise, after the vSphere Web Client deploys your plug-in package, the plug-in will not work because the plug-in dependencies cannot be resolved and may cause errors in the vSphere Web Client.

  For example, you can use the following lines in the manifest file of your plug-in package to define the minimum and maximum supported versions of the vSphere Web Client:

```
...
  <dependencies>
   <pluginPackage id="com.vmware.vsphere.client" version="5.5.0" match="greaterOrEqual" />
   <pluginPackage id="com.vmware.vsphere.client" version="6.0.0" match="lessThan" />
  </dependencies>
...
```

  Note If your plug-in package is only compatible with a specific version of the vSphere Web Client, you must use the `equal` value of the `match` attribute to specify the version. In this way, you ensure that when you upgrade the vSphere Web Client, your plug-in package is not deployed, and does not cause any errors.

- To avoid deployment failures, you must create a ZIP archive file for your vSphere Web Client extension. Moreover, if you want to complete successfully the certification for your vSphere Web Client plug-in, know that the plug-in signing tool signs only plug-ins that have the ZIP file format.

Best Practices for Plug-In Modules Implementation

Following general design and development recommendations is the first step in creating high-performance and secure vSphere Web Client extensions. You can then move on to special areas, such as developing HTML-based extension solutions.

- When you develop and test your extension solution, make sure that the plug-in package installs and functions properly on both the vCenter Server Appliance and the vCenter Server running on Windows. Your plug-in package must be OS agnostic. You must avoid reading and writing on the file system from the vSphere Web Client service layer. In case you need to temporarily store files, you must use the Tomcat temp directory. To achieve platform independent read/write file operations, use the Java APIs for handling file paths.
- To provide a consistent end-user experience in case your vSphere Web Client extension migrates server workloads, make sure that your extension migrates only to vSphere environments that are hosted by a VMware vCloud Air Network Service Provider. For more information about the available service providers, see http://vcloudproviders.vmware.com/find-a-provider.
- When you create a Flex-based extension for the vSphere Web Client, you must use the Data Access Manager API. The Data Access Manager communicates with the Data Service on the service layer. The Data Service acts as the intermediary between the user interface layer and the objects in the vSphere environment.

  Using the Data Access Manager library has the following advantages:

  - Provides a unified data channel
  - Enables sending multiple queries in a batch request
  - Provides a simple event-based API
  - Enables receiving of aggregated responses to data requests
  - Provides mechanisms for extending the existing vSphere managed objects and creating custom ones.
- To achieve greater security and scalability, you must use the Java service layer for requesting data from the vSphere environment. The communication between the Flex GUI objects and the Java service layer is achieved by using the BlazeDS Java remoting technology and creating secure AMF channels. This approach eliminates security breaches to data sent to and received from the vSphere environment.
- Avoid using deprecated or private APIs and extension points. Using deprecated APIs in your vSphere Web Client extensions will prevent them from working with future version of the vSphere Web Client.

- To prevent performance problems in the vSphere Web Client and vCenter Server instances, use your Java services only for communication between the vCenter Server instances, or other remote data sources, and the user interface layer. You must not create thread pools in your Java services. Consider implementing any complex business logic in your own backend servers.
- Avoid caching data in the Java service layer. Make sure that the vSphere Web Client remains stateless. To ensure the scalability of the vSphere Web Client, you must use your backend server to cache data.
- Use the data refresh mechanisms, which are provided by the Data Access Manager, to limit the time for refreshing the data of your view objects. For further information about how to use the Data Access Manager refresh mechanism, see Data Refresh and Data Update Specifications.
- To increase the security of your extensions, you must limit the access to your plug-ins to specific users. Use the `plugin.xml` extension definition to control the user access to your extensions based on their privileges. For example, you can make your extensions available only to users who have privileges to create or delete Datastore objects. For more information about how to filter extensions, see Filtering Extensions.
- To achieve optimal scalability and performance for your vSphere Web Client plug-ins, your Java services must not require any significant heap allocation.

Best Practices for Developing HTML-Based Extensions

You can use the vSphere Web Client SDK and the vSphere Client development kit to create HTML-based extensions.

Starting with vSphere 5.5 Update 1, an HTML Bridge infrastructure is added to the vSphere Web Client that provides support for HTML-based extensions. The vSphere Web Client SDK provides APIs, tools, and samples that can help you extend the vSphere Web Client.

Starting with vSphere 6.5, you can use the vSphere Client to connect to vCenter Server systems and manage vSphere inventory objects. The vSphere Client development kit is provided to developers that want to create HTML5-based extensions for both Web browser applications. For backward compatibility, the vSphere Client development kit contains the same APIs as the HTML Bridge.

Follow these best practices when you create your HTML-based solutions.

- Make sure that your HTML and JavaScript code is fully functional on different Web browsers and provides the same user experience.
- You must not send calls to the topmost browser window `window.top` or to the parent object of your current window `window.parent`.
- You must include in your HTML-based extensions the latest version of the `web-platform.js` JavaScript file provided with the vSphere Web Client SDK and added to each extension during generation. If you use an older version of this file, your HTML-based extensions might not work in the vSphere Web Client and might cause other HTML-based extensions to stop working.

- To minimize future maintenance work and prevent incompatibility problems, do not change the `web-platform.js` JavaScript file on your own initiative. The file depends on the vSphere Web Client version and is updated with each major release of the SDK. If the file changes between major releases, you must see whether the release notes contain any instructions for manual changes that you must apply to the file before generating your plug-in packages.
- To ensure the integrity of future versions of the vSphere Web Client running HTML-based extensions, do not modify the `WEB_PLATFORM` object. All HTML-based extensions use this global variable to access the vSphere Web Client platform APIs. For example, if you change this variable, other HTML-based extensions that use the `WEB_PLATFORM = self.parent.document.getElementById("container_app")` variable initialization might stop working.

Best Practices for Extending the User Interface Layer

When developing extensions for the user interface layer of the vSphere Web Client, follow these best practices.

- Create pointer node extensions on the Object Navigator home page only for major applications and solutions. This approach provides consistent and meaningful user experience for the customized vSphere Web Client.
- When you create a view for your Flex extension, use the vSphere Web Client SDK Model View Controller framework (Frinje) and the Data Access Manager APIs to achieve better performance and scalability.
- When creating action set extensions for a particular type of vSphere object, you must use the extensions filtering mechanism. The defined action sets must be visible only when the user selects the relevant vSphere object type. For more information about how to filter extensions, see Filtering Extensions.
- If your Flex-based solution extends the `com.vmware.flexutil.proxies.BaseProxy` class, make sure that you are not hard-coding the channel URI. Use the `getDefaultChannelUri()` method of the `com.vmware.flexutil.ServiceUtil` Flex class.
- Use the Data Access Manager API for retrieving data from the service layer. You must use proxies only for adding, editing, and deleting issued data requests.
- For better performance, avoid making proxy calls that require more than several seconds to return a response. A best practice is to design your extensions to submit a task that returns immediately, and to track the task progress.
- If you use proxies for data requests, verify that you receive the request response before sending another one through the proxy.
- If you want to use localization data for your plug-in package, follow these recommendations:
  - Set the `locale` attribute in the `<resource>` element of the `plugin.xml` manifest file to the value `{locale}`. Using the `{locale}` value instructs the vSphere Web Client and the vSphere Client to use the locale that the user's Web browser specifies at runtime.

The following XML fragment shows how the `<resource>` element can be used in the plug-in module manifest file of a Flex-based extension.

```
<plugin id="com.vmware.samples.helloworld_i18nui"
   moduleUri="Helloworld.swf" defaultBundle="com_vmware_samples_helloworld">

   <resources>
      <resource locale="{locale}">
         <!-- relative path of the .swf generated by the build script.
            {locale} will be set at runtime to the current vSphere Web Client locale
         -->
         <module uri="locales/helloworld_{locale}.swf"/>
      </resource>
   </resources>

....

</plugin>
```

- To avoid collisions with other localized plug-in packages, set a unique resource bundle name to the `defaultBundle` attribute of the `<plugin>` element in the plug-in manifest file.
- If you develop HTML-based plug-ins, use the `getString()` API defined in the `web-platform.js` JavaScript file.
- If you develop HTML-based plug-ins, make sure that the filenames of your resource files end with *_en_US* instead of *-en_US*

Best Practices for Extending the Service Layer

Following these recommendations and best practices for creating extensions of the vSphere Web Client service layer, can help you improve the security, scalability, and performance of your extension solutions.

- To avoid deployment errors, add your services to the Spring configuration by using the `bundle-context.xml` Spring configuration file. Do not create alternative Spring contexts.
- To increase the deployment speed of your extensions, make sure that you optimize your Spring context initialization. You must use as little source code as possible in the constructor and the initialization method of your Spring beans.
- Avoid using timers for pooling data from the vSphere environment. In case there is no other way to retrieve the required data, you must make sure that data queries are not overlapping.
- If you use a tool to automatically generate the manifest file of your service layer extension, make sure that no third-party packages are added to the `Package-Export` manifest header.

OSGi-Specific Recommendations

Following these OSGi-specific recommendations, helps you improve the performance and scalability of your Java service layer extensions.

- To avoid deployment errors in case your plug-in depends on a third-party library with a different version than the ones available on the Virgo server, you can embed the library inside your bundle. You must also specify the library in your bundle manifest file by using the `Bundle-Classpath` manifest header. In this way, the bundle class loader looks for required classes among the classes from your plug-in and also from the embedded third-party library.

  For example, if your bundle uses classes from the `thirdPartyLibrary.jar`, add the JAR to the root of the bundle and add the following line to the bundle manifest file:

  ```
  Bundle-Classpath: .,thirdPartyLibrary.jar
  ```

  As a result, when you deploy your plug-in on the Virgo server, your bundle dependencies are resolved using the embedded third-party library and not the one that is already on the server.

- To avoid future compatibility issues, make sure that you follow the recommendations of the OSGi Alliance for wiring bundles. Use the `Import-Package` manifest header to declare your package dependencies and not the `Require-Bundle` header.

- To avoid deployment failures in case your bundle imports packages that are exported from `vim25.jar`, remove any packages exported by the `vim25.jar` bundle from the package imports of your `MANIFEST.MF` file. You must add the following line to your `MANIFEST.MF` file:

  ```
  Require-Bundle: com.vmware.vim25;bundle-version=1.0.0
  ```

  You might have deployment issues, if you have in your environment a plug-in package that contains the `vijava-osgi.jar` bundle.

- To improve the future maintenance of your bundles, you must export as few packages as possible. Remember that every exported package is considered a public API that must be versioned and maintained. If you export packages that contain implementation classes, your specific implementation becomes harder to evolve and to be maintained in the future. Ideally, you must export APIs by using a dedicated API bundles. Other bundles must import the APIs and provide implementation classes that use and publish services. The implementation classes must not export packages.

- To avoid deployment errors, you must not export packages that do not belong to your own code. If you include a third-party bundle in your bundle, do not export any classes from the third-party bundle.

- To avoid future compatibility issues in case you import a package from the vSphere Web Client bundles, set the package version to `0` in the `MANIFEST.MF` file. When you update the vSphere Web Client platform, your bundle might stop working if you specified concrete package version that is not available after the update. If you do not specify a version, the OSGi validation utility logs a warning message in the `plugin-medic.log` file.

For example, if you import the `com.vmware.vise.data` and `com.vmware.vise.data.query` packages, you must add the following line to your `MANIFEST.MF` file:

```
Import-Package: com.vmware.vise.data;version="0", com.vmware.vise.data.query;version="0"
```

- To improve the performance of your plug-in package, avoid using the `DynamicImport-Package` manifest header unless necessary. If you use the `DynamicImport-Package` header in your bundle and the packages you want to import are not known in advance, the Virgo framework switches to searching mode for a publicly available package that satisfies the requirement. The use of wildcards is discouraged.
- To improve the deployment time of your plug-in packages, you must add as few bundles as possible to the `<bundlesOrder>` element of your `plugin-package.xml` manifest file. All bundles that are not included in the ordered bundles list are deployed in parallel.

  For example, you can deploy the OSGi bundles from your plug-in package in a parallel manner. This deployment is achieved, if you move all APIs exported by bundle A and imported by bundle B to a separate `my_api.jar` bundle. Include the `my_api.jar` bundle to the ordered bundles list of your plug-in package. In this way, the dependencies of bundle A and B are satisfied in advance and these bundles can start in parallel.
- To improve the deployment time of your plug-in package, do not perform Spring bean initialization in the bundles from the ordered bundles list. The deployment of bundles is blocked until the Spring bean initialization is completed for each bundle that is part of the ordered bundles list. This behavior slows down the startup of the Virgo server. You must use the bundles from the ordered bundles list only to export APIs and data transfer objects, if possible. For more information, see the previous recommendation.
- To speed up the deployment of your plug-in package, you must use as few Web application ARchive (WAR) files as possible, ideally only one WAR file per plug-in package. WAR files are deployed slower that the other bundles, especially when the Web application has OSGi dependencies. For example, the deployment process can be slowed down when the Web application registers a message broker.
- To avoid runtime errors, you can specify the versions of the packages that you import and export for your OSGi bundle.
- Starting with vSphere 6.5, an OSGi validation utility is added to the vSphere Web Client which ensures that the deployed plug-ins follow the OSGi-specific best practices. The results from the validation checks are logged to the `plugin-medic.log` file which is located in the same folder as the Virgo server log file, `vsphere_client_virgo.log`. For more information about the location of the Virgo server log files, see the Table 11-2 table.

Once the deployment of all plug-ins completes, the validation for the whole set of OSGi bad practices begins. Any issues detected are logged as `INFO` and `WARN` messages to the `plugin-medic.log` file. For example, following are some of the warning messages that can be seen in the log file after you deploy your plug-ins:

- `DynamicImport-Package should be avoided` - To prevent performance issues during plug-in deployment, you must avoid using the `DynamicImport-Package` manifest header to declare packages that must be looked up at runtime. Using dynamic imports might cause instability issues with the vSphere Web Client. To complete successfully the certification of your plug-in, use wildcards with caution and avoid using declarations such as the following: `DynamicImport-Package: com.vmware.*`.
- `Don't use 'com.vmware' prefix for bundle symbolic names and packages` - The warning message is logged when a third-party bundle exports packages with the `com.vmware` prefix and the bundle's symbolic name starts with a different prefix.
- `Conflicting package exports` - The warning message is logged when two or more plug-ins contain bundles that export the same package. This violation of the recommendations of the OSGi Alliance leads to `ClassNotFoundException`s at runtime that are difficult to troubleshoot. For example, in production environments, this warning message is logged in case two plug-ins contain bundles that export Hibernate or another third-party library with the same version number.

`DataService` -Specific Best Practices

Following these recommendations and best practices for writing Data Service queries, helps you improve the performance and scalability of your extensions.

- To increase the performance of your extension, you must avoid creating constraints, such as `ObjectIdentityConstraints`, `PropertyConstraints`, and `RelationalConstraints`, and defining `OrderingPropertySpec` objects that have multi-valued properties such as collections and arrays.

  For example, when you create a `PropertyConstraint` object that filters all `VirtualMachine` objects based on their `network` property, the filtering process is slowed down. This situation occurs because the back end Data Provider does not support such requests. In such cases, the Data Service fetches the entire data set and then filters the received data.

- To improve the performance of your extension, you must avoid creating constraints and defining `OrderingPropertySpec` objects by using the length of multi-valued properties such as collections and arrays.

  For example, when you create a `PropertyConstraint` object that filters query results by using the property `network._length` for all `VirtualMachine` objects, the filtering process is slowed down. This situation occurs because the back end Data Provider does not support such requests or does not maintain a separate index for property length. In such cases, the Data Service fetches the entire data set and then proceeds with filtering the received data.

- To improve the performance of your extensions, you can use `QuerySpec.resultSpec.maxResultCount` field to limit the returned result set.

- To improve the performance of your extensions in case you use PropertyConstraints, you must use the com.vmware.vise.data.query.Comparator.EQUALS comparator instead of a text-matching comparator such as com.vmware.vise.data.query.Comparator.CONTAINS and com.vmware.vise.data.query.Comparator.TEXTUALLY_MATCHES for the PropertyConstraint queries. Text-matching operations require a specific database indexing which only a few properties, such as name, have. If you need to use a text-matching comparator, you can use CONTAINS instead of TEXTUALLY_MATCHES, because TEXTUALLY_MATCHES requires more complex processing.
- To improve the performance of your extensions, you can set a value to the targetType field of each com.vmware.vise.data.PropertySpec and com.vmware.vise.data.query.OrderingPropertySpec object. The Data Service uses the targetType field to optimize the execution of the queries.
- To avoid future compatibility issues with your vSphere Web Client extension, you must avoid using multi-valued properties, such as collections and arrays, as the middle nodes in the property paths.

  For example, you must not use the property path configurationEx.drsVmConfig.key for ClusterComputeResource objects because the drsVmConfig property of the vim.cluster.ConfigInfoEx data object is a collection. In this case, you must request the whole vim.cluster.ConfigInfoEx data object.
- To avoid future compatibility issues with your vSphere Web Client extension, you must not use any custom properties defined by the vSphere Web Client modules. These properties are prone to change in the future. You must use only the properties defined in the vSphere Web Service API for the managed objects and data objects.
- To avoid future compatibility issues with your extension, you must avoid using the com.vmware.vise.data.query.Conjoiner.EXCEPT operator in your CompositeConstraints. Instead you must use negation and De Morgan's laws.
- To avoid future incompatibility, avoid using the relation field of the com.vmware.vise.data.PropertySpec objects.
- To avoid future incompatibility, avoid using the facets field of the com.vmware.vise.data.query.ResultSpec objects.
- The Data Service uses the value of the targetType field to optimize query execution. To improve the performance of your extensions, set the targetType field on every constraint except for the following cases:
  - com.vmware.vise.data.query.ObjectIdentityConstraint - You must not specify the targetType field because the type is already present in the object reference. You can set the type by using the target field of the ObjectIdentityConstraint class.
  - com.vmware.vise.data.query.RelationalConstraint with hasInverseRelation field set to true - The targetType field is ignored for such constraints.
- To avoid performance issues with your extension in case you use constraints, you must use a specific managed object type as a value for the targetType field. For example, if you use an abstract base type such as the ManagedEntity managed object type, the execution of the query is slowed down.

- To ease the future optimization of your extensions, you must limit the size of each `CompositeConstraint` by limiting the number of child constraints in the `nestedConstraints` field of the `CompositeConstraint` class, and you must avoid also nesting multiple `CompositeConstraint`.
- Make sure that your Data Provider Adapter takes less than 3 seconds to process a query. If your adapter takes too long to process a request, the Data Service cuts the adapter from the result.

Best Practices for Deploying and Testing Your vSphere Web Client Extensions

After you develop your vSphere Web Client extension, you can follow these recommendations to ensure that your extension is successfully deployed to the vSphere Web Client or the vSphere Client .

- To improve the performance of your plug-in package, the initial download and deployment time after the first time the user logs into the vSphere Web Client, must be less than a minute.
- To ease the testing and debugging of your plug-in package, you must include the build number in the dot-separated version number of the plug-in package when you register the plug-in as a vCenter Server extension.
- To prevent deployment issues when you try to deploy a new version of a registered plug-in package, make sure that you modify the version property of your plug-in package in the `plugin-package.xml` manifest file.
- To prevent deployment issues when you try to deploy a plug-in package with the same version, make sure that you unregister the plug-in package by removing the plug-in as a vCenter Server extension point. You must also manually delete the cached files of the plug-in package that are stored on the Virgo server from one of the following locations:

  Table 11-1. Locations for the Cached Downloaded Plug-In Packages

Virgo Location	Cache Location
vCenter Server for Windows	`C:\ProgramData\VMware\vCenterServer\cfg\vsphere-client\vc-packages\vsphere-client-serenity\`
vCenter Server Appliance	`/etc/vmware/vsphere-client/vc-packages/vsphere-client-serenity/`
Windows OS	`%PROGRAMFILES%/VMware/vSphere Web Client/vc-packages/vsphere-client-serenity/`
Mac OS	`/var/lib/vmware/vsphere-client/vsphere-client/vc-packages/vsphere-client-serenity/`

- To avoid issues with the vSphere Web Client performance, make sure that your plug-in package has only one version registered with the vCenter Server. You must not change the value of the `key` property of the vCenter Server `Extension` data object between releases.

- To verify easily the deployment of your plug-in package and monitor for any issues related to your plug-in, you must know how to work with the Virgo server log files. You can find these log files in one of the following locations:

  Table 11-2. Log Files Location

Environment	Virgo Log Files Location
vSphere Client development environment (Windows or Mac OS)	`html-client-sdk/vsphere-ui/server/serviceability/logs/vsphere_client_virgo.log`
vSphere Web Client SDK development environment (Windows or Mac OS)	`flex-client-sdk/server/serviceability/logs/vsphere_client_virgo.log`
vCenter Server Appliance 6.5 installation vSphere Client	`/var/log/vmware/vsphere-ui/logs/`
vCenter Server Appliance 6.5 and 6.0 installation vSphere Web Client	`/var/log/vmware/vsphere-client/logs/`
vCenter Server for Windows 6.5 and 6.0 vSphere Web Client	`C:\ProgramData\VMware\vCenterServer\logs\vsphere-client\logs\`

  The `vsphere_client_virgo.log` file contains the log information that the Virgo server generates. Problems usually start with the `[ERROR]` tag. Use your plug-in package name or the bundle symbolic name to detect errors caused by your plug-in.

- To log information about your plug-in package, you must use the default logging mechanisms of the vSphere Web Client. Use the Apache Log4j logging framework to provide debugging information for your plug-in package. The Virgo server uses the Simple Logging Facade for Java (SLF4J) logging API.

List of Extension Points

12

The vSphere Web Client publishes extension points that you can use to create your extensions. The following sections contain a list of the currently supported extension points, including a brief description of each extension point and the required extension definition type.

This section includes the following topics:

Global Extension Points

Global extension points allow you to extend the home screen, to add a global view to the main workspace, or to control application-wide settings.

vise.global.views		Flex Client	HTML Client
	Compatibility	yes	yes

Adds a global UI view to the main area that is not related to vSphere objects.

Requires a data object of type `GlobalViewSpec` with available properties:

- `name` - user-visible name of the global view.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.

Accessibility: can be a target of any navigation request.

Example:

```
<extension id="com.vmware.samples.h5.globalview.mainView">
   <extendedPoint>vise.global.views</extendedPoint>
   <object>
      <name>My Global View</name>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/globalview/resources/mainView.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

vise.home.shortcuts		Flex Client	HTML Client
	Compatibility	yes	yes

Add a home screen shortcut to a global view or other data view.

Requires a data object of type `ShortcutSpec` with available properties:

- `name` - user-visible name of the shortcut.
- `icon` - (optional) resource ID of 32x32 shortcut icon.
- `categoryUid` - ID of the category this shortcut will be displayed in. Supported values are "`vsphere.core.controlcenter.inventoriesCategory`" and "`vsphere.core.controlcenter.monitoringCategory`".
- `targetViewUid` - identifier of the extension to navigate to when the shortcut is clicked.

Accessibility on vSphere Client: Shortcuts.

Accessibility on vSphere Web Client: Home.

Example:

```
<extension id="com.vmware.samples.h5.globalview.shortcut">
   <extendedPoint>vise.home.shortcuts</extendedPoint>
   <object>
      <name>My Shortcut</name>
      <icon>#{appIcon}</icon>
      <categoryUid>vsphere.core.controlcenter.monitoringCategory</categoryUid>
      <targetViewUid>com.vmware.samples.h5.globalview.mainView</targetViewUid>
   </object>
</extension>
```

vise.dispose.namespace.inclusions		Flex Client	HTML Client
	Compatibility	yes	no

Specifies a list of namespaces to the Dispose Manager that must be garbage-collected. All vSphere object types that have the specified namespaces will be garbage-collected.

Accessibility: Not displayed.

vsphere.core.objectTypes		Flex Client	HTML Client
	Compatibility	yes	yes

Declares UI information that is associated with a custom object type.

Requires a data object of type `com.vmware.core.specs.ObjectTypeSpec` with available properties:

- `types` - list of type names applicable to the same type info.
- `icon` - resource ID of a 18x18 icon associated with this object type.
- `label` - localized type name.
- `labelPlural` - plural of the localized type name.
- `listViewId` - (optional) ID of the list view extension used to display multiple objects of this object type. If missing or null, the default ${namespace}.list is used.

Accessibility: Not directly displayed, just declares the new object type.

Example:

```
<extension id="com.vmware.samples.chassisa.objectType">
   <extendedPoint>vsphere.core.objectTypes</extendedPoint>
   <object>
      <types>
         <String>samples:ChassisA</String>
      </types>
      <label>Chassis</label>
      <labelPlural>ChassisA's</labelPlural>
      <icon>#{chassis.icon}</icon>
   </object>
</extension>
```

Object Navigator Extension Points

You can extend the object navigator by creating new nodes and categories on each page. You can customize also any object collection node that you create by adding a new icon and label.

vise.navigator.nodespecs		Flex Client	HTML Client
	Compatibility	yes	yes

Adds an object collection node, category, or pointer node extension to the object navigator.

Requires a data object of type `ObjectNavigatorNodeSpec` with available properties:

- `title` - user-visible node title.
- `icon` - (optional) 18x18 node icon resource ID.
- `navigationTargetUid` - (optional) ID of the extension to navigate to when the node is selected.
- `parentUid` - ID of the parent extension this node will be displayed in. Supported values are:
  - "`vsphere.core.navigator.solutionsCategory`". Accessibility: Object Navigator root.
  - "`vsphere.core.navigator.virtualInfrastructure`". Accessibility: Object Navigator → Global Inventory Lists.
  - "`vsphere.core.navigator.administration`". Accessibility: Object Navigator → Administration.

Example:

```
<extension id="com.vmware.samples.chassisACategory">
   <extendedPoint>vise.navigator.nodespecs</extendedPoint>
   <object>
      <title>ChassisA Category</title>
      <parentUid>vsphere.core.navigator.virtualInfrastructure</parentUid>
   </object>
</extension>
```

vise.inventory.representationspecs		Flex Client	HTML Client
	Compatibility	yes	yes

Defines 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 `ObjectRepresentationSpec` with available properties:

- `objectType` - type of objects to which the specs apply.
- `specCollection` - array of IconLabelSpec objects, each of which contains:
  - `iconId` - (optional) 18x18 icon resource ID.
  - `labelId` - (optional) label or its resource ID.
  - `conditionalProperties` - (optional) array of property names. The icon and label are applicable only if the values of all properties evaluate to "true". Note: To test for "false" use the negation operator "!" in front of the property name.
  - `conditions` - (optional) array of PropertyConstraint-s. The icon and label are applicable only if all constraints are satisfied.

Accessibility: Object Navigator → Global Inventory Lists.

Example:

```
<extension id="com.vmware.samples.chassisa.iconLabelSpecCollection">
   <extendedPoint>vise.inventory.representationspecs</extendedPoint>
   <object>
      <objectType>samples:ChassisA</objectType>
      <specCollection>
         <com.vmware.ui.objectrepresentation.model.IconLabelSpec>
            <iconId>#{chassis}</iconId>
         </com.vmware.ui.objectrepresentation.model.IconLabelSpec>
      </specCollection>
   </object>
</extension>
```

vsphere.core.objectTypes		Flex Client	HTML Client
	Compatibility	yes	yes

Declares UI information that is associated with a custom object type.

Requires a data object of type `com.vmware.core.specs.ObjectTypeSpec` with available properties:

- `types` - list of type names applicable to the same type info.
- `icon` - resource ID of a 18x18 icon associated with this object type.
- `label` - localized type name.
- `labelPlural` - plural of the localized type name.
- `listViewId` - (optional) ID of the list view extension used to display multiple objects of this object type. If missing or null, the default ${namespace}.list is used.

Accessibility: Not directly displayed, just declares the new object type.

Example:

```
<extension id="com.vmware.samples.chassisa.objectType">
   <extendedPoint>vsphere.core.objectTypes</extendedPoint>
   <object>
      <types>
         <String>samples:ChassisA</String>
      </types>
      <label>Chassis</label>
      <labelPlural>ChassisA's</labelPlural>
      <icon>#{chassis.icon}</icon>
   </object>
</extension>
```

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 **Configure** tab view. Every object workspace extension point requires a data object of type `com.vmware.ui.views.ViewSpec`.

Most 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 **Configure** 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
- `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}.summarySectionViews.html		Flex Client	HTML Client
	Compatibility	no	yes

Adds an HTML portlet to the **Summary** tab view.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the global view.
- `icon` - (optional) 18x18 portlet icon resource ID.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.
  - `dialogTitle` - portlet title.
  - `dialogSize` - portlet width and height.

Accessibility: {vSphere object} → Summary page.

Example:

```
<extension id="com.vmware.samples.vspherewssdk.vm.summary2">
   <extendedPoint>vsphere.core.vm.summarySectionViews.html</extendedPoint>
   <object>
      <name>#{summaryView.title}</name>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/vspherewssdk/resources/vm-summary.html</url>
               <dialogTitle>WSSDK Summary Sample</dialogTitle>
               <dialogSize>440,400</dialogSize>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

vsphere.core.${objectType}.summarySectionViews		Flex Client	HTML Client
	Compatibility	yes	yes

Adds a Flex portlet to the **Summary** tab view.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the global view.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.
  - `dialogTitle` - portlet title.
  - `dialogSize` - portlet width and height.
  - `dialogIcon` - (optional) 18x18 portlet icon resource ID.

Accessibility: {vSphere object} → Summary page.

Example:

```
<extension id="com.vmware.samples.vspherewssdk.vm.summary">
   <extendedPoint>vsphere.core.vm.summarySectionViews</extendedPoint>
   <object>
      <name>#{summaryView.title}</name>
      <componentClass
  className="com.vmware.samples.wssdkui.views.VsphereWsSdkVmMonitorView"/>
   </object>
</extension>
```

vsphere.core.${objectType}.monitorCategories		Flex Client	HTML Client
	Compatibility	no	yes

Adds a sub-view category to the **Monitor** tab view.

Requires a data object of type `CategorySpec` with available properties:

- `label` - user-visible name of the Monitor view category.

Accessibility: {vSphere object} → Monitor page

Example:

```
<extension id="com.vmware.samples.vspherewssdk.vm.monitor.category">
   <extendedPoint>vsphere.core.vm.monitorCategories</extendedPoint>
   <object>
      <label>WSSDK Category</label>
   </object>
</extension>
```

vsphere.core.${objectType}.monitorViews		Flex Client	HTML Client
	Compatibility	yes	yes

Adds a sub-view to the **Monitor** tab view.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the Monitor view.
- `categoryUid` - (optional) ID of the category this Monitor view belongs to.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.

Accessibility: {vSphere object} → Monitor page

Example:

```
<extension id="com.vmware.samples.vspherewssdk.vm.monitor">
   <extendedPoint>vsphere.core.vm.monitorViews</extendedPoint>
   <object>
      <name>Monitor view</name>
      <categoryUid>com.vmware.samples.vspherewssdk.vm.monitor.category</categoryUid>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/vspherewssdk/resources/vm-monitor.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

vsphere.core.${objectType}.manageCategories		Flex Client	HTML Client
	Compatibility	no	yes

Adds a sub-view category to the **Configure** tab view.

Requires a data object of type `CategorySpec` with available properties:

- `label` - user-visible name of the Configure view category.

Accessibility: {vSphere object} → Configure page

Example:

```
<extension id="com.vmware.samples.vspherewssdk.vm.manage.category">
   <extendedPoint>vsphere.core.vm.manageCategories</extendedPoint>
   <object>
      <label>WSSDK Category</label>
   </object>
</extension>
```

vsphere.core.${objectType}.manageViews		Flex Client	HTML Client
	Compatibility	yes	yes

Adds a sub-view to the **Configure** tab view.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the Configure view.
- `categoryUid` - (optional) ID of the category this Configure view belongs to.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.

Accessibility: {vSphere object} → Configure page

Example:

```
<extension id="com.vmware.samples.vspherewssdk.vm.manage">
   <extendedPoint>vsphere.core.vm.manageViews</extendedPoint>
   <object>
      <name>Configure view</name>
      <categoryUid>com.vmware.samples.vspherewssdk.vm.manage.category</categoryUid>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/vspherewssdk/resources/vm-configure.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

vise.relateditems.specs		Flex Client	HTML Client
	Compatibility	yes	yes

Creates a new relation between object types, either vSphere objects or custom objects

Requires a data object of type `ObjectRelationSetSpec` with available properties:

- `type` - vSphere/Custom object type.
- `relationViewId` - ID of a view that can display object relations.
- `conditionalProperty` - (optional) property name to introduce additional constraints on the object type for a relation.

  Note: To test for "false" use the negation operator "!" in front of the property name.
- `relationSpecs`
  - `id` - relation ID
  - `label` - user-visible label of the relation.
  - `icon` - 18x18 relation icon resource ID.
  - `listViewId` - ID of a view that can display relation items.
  - `relation` - (optional) property name wrapped into a `RelationalConstraint`.
  - `inverseRelation` - (optional) property name used to check if an object applies to the relation.
  - `conditionalProperty` - (optional) property name wrapped into a `PropertyConstraint`.

    Note: To test for "`false`" use the negation operator "`!`" in front of the property name.
  - `targetType` - (optional) target type name used in any kind of Constraint.
  - `constraint` - (optional) general constraint used in case of relations that cannot be expressed in terms of `targetType`, `relation` and `conditionalProperty`

Accessibility: {vSphere object} → {related object type} in case of single relation; {vSphere object} → More objects in case of multiple relations.

Example:

```
<extension id="com.vmware.samples.relateditems.specs.host">
   <extendedPoint>vise.relateditems.specs</extendedPoint>
   <object>
      <type>HostSystem</type>
      <relationsViewId>vsphere.core.host.related</relationsViewId>
      <relationSpecs>
         <com.vmware.ui.relateditems.model.RelationSpec>
            <id>chassisForHost</id>
            <icon>#{chassis}</icon>
            <label>Chassis relation</label>
            <relation>chassis</relation>
            <targetType>samples:ChassisB</targetType>
            <listViewId>com.vmware.samples.chassisb.list</listViewId>
         </com.vmware.ui.relateditems.model.RelationSpec>
      </relationSpecs>
   </object>
</extension>
```

vsphere.core.${objectType}.monitor.performanceViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a view under the **Performance** second-level tab of the **Monitor** tab view.

Accessibility: {vSphere object} → Monitor → Performance

vsphere.core.${objectType}.manage.settingsViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a view under the **Settings** second-level tab of the **Configure** tab view.

Accessibility: {vSphere object} → Configure → Settings

vsphere.core.${objectType}.manage.alarmDefinitionsViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a view to the Alarm Definitions element in the Issues second-level tab of the Configure tab view.

Accessibility: {vSphere object} → Monitor → Alarm Definitions

vsphere.core.${objectType}.list.columns		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Creates a new column in the list of vSphere objects of type `${objectType}` in the object workspace.

Requires a data object of type `com.vmware.ui.lists.ColumnSetContainer`.

Note: Only the XML representation is supported.

Accessibility: {vSphere object list}

Actions Extension Points

Actions are invoked in the vSphere Web Client and the vSphere Client from menus or toolbars. The actions extension points allow you to add actions to global or contextual menus, and to prioritize the placement of actions within menus and toolbars.

vise.actions.sets		Flex Client	HTML Client
	Compatibility	yes	yes

Defines a set of actions, each of which is represented by the class `ActionSpec`.

Requires a data object of type `ActionSpec` with available properties:

- `uid` - action ID.
- `label` - user-visible action label.
- `actionUrl` - URL of the action target.
- `dialogTitle` - target dialog title.
- `dialogSize` - target dialog width and height.

Accessibility: {object} → {menu} → {plugin sub-menu}

Example:

```
<extension id="com.vmware.samples.vspherewssdk.vmActionSet">
   <extendedPoint>vise.actions.sets</extendedPoint>
   <object>
      <actions>
         <com.vmware.actionsfw.ActionSpec>
            <uid>com.vmware.samples.vspherewssdk.myVmAction1</uid>
            <label>#{action1.label}</label>
            <delegate>
               <className>com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate</className>
               <object><root>
                  <actionUrl>/vsphere-client/vspherewssdk/resources/vm-action-dialog.html</actionUrl>
                  <dialogTitle>#{action1.label}</dialogTitle>
                  <dialogSize>500,250</dialogSize>
               </root></object>
            </delegate>
         </com.vmware.actionsfw.ActionSpec>
      </actions>
   </object>
   <metadata>
      <objectType>VirtualMachine</objectType>
   </metadata>
</extension>
```

vmware.prioritization.actions		Flex Client	HTML Client
	Compatibility	yes	no

Defines and prioritizes actions in object menus and the object list toolbar.

Requires a data object of type `com.vmware.actionsfw.model.ActionPriorityGroup`.

Accessibility: {object list} → {action button bar} and {list menu}

vmware.prioritization.listActions		Flex Client	HTML Client
	Compatibility	yes	yes

Defines and prioritizes global list actions (not related to a particular object).

Requires a data object of type `ActionPriorityGroup` with available properties:

- `prioritizedIds` - list of action IDs to declare as global.

  Note: the order of global action IDs is taken into account only in the vSphere Web Client (Flex). The vSphere HTML Client does not support action prioritization.
- `regionId` - ID of the extension that contains the global actions.

Accessibility: {object list} → {action button bar} and {list menu}

Example:

```
<extension id="com.vmware.sample.chassis.listAction">
   <extendedPoint>vmware.prioritization.listActions</extendedPoint>
   <object>
      <prioritizedIds>
         <String>com.vmware.samples.chassisa.createChassis</String>
      </prioritizedIds>
      <regionId>com.vmware.samples.chassisa.list</regionId>
   </object>
</extension>
```

vsphere.core.menus.solutionMenus		Flex Client	HTML Client
	Compatibility	yes	yes

Defines a custom sub-menu including actions, separators, and nested menus.

Requires a data object of type `ActionMenuItemSpec` with available properties:

- `uid` - menu item ID.
- `type` - type of menu item. Supported values are "`menu`", "`action`" and "`separator`".
- `label` - (optional) user-visible label of the menu item.

  If omitted and the type is "`action`", the label defined in the action declaration will be used.
- `icon` - (optional) 18x18 icon resource ID.

  If omitted and the type is "`action`", the icon defined in the action declaration will be used.
- `children` - (optional) array of child menu items (`ActionMenuItemSpec`) if the type is "`menu`".

Accessibility: {object} → {menu}

Example:

```
<extension id="com.vmware.samples.vspherewssdk.vmMenu">
   <extendedPoint>vsphere.core.menus.solutionMenus</extendedPoint>
   <object>
      <label>WSSDK menu</label>
      <children>
         <Array>
            <com.vmware.actionsfw.ActionMenuItemSpec>
               <type>action</type>
               <uid>com.vmware.samples.vspherewssdk.myVmAction1</uid>
            </com.vmware.actionsfw.ActionMenuItemSpec>
            <com.vmware.actionsfw.ActionMenuItemSpec>
               <type>separator</type>
            </com.vmware.actionsfw.ActionMenuItemSpec>
            <com.vmware.actionsfw.ActionMenuItemSpec>
               <type>action</type>
               <uid>com.vmware.samples.vspherewssdk.myVmAction2</uid>
            </com.vmware.actionsfw.ActionMenuItemSpec>
         </Array>
      </children>
   </object>
   <metadata>
      <objectType>VirtualMachine</objectType>
   </metadata>
</extension>
```

Extension Templates

When you add custom vSphere objects, use the extension templates to make the vSphere Web Client user interface consistent.

vsphere.core.inventory.objectViewTemplate		Flex Client	HTML Client
	Compatibility	yes	yes

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, second-level tabs, and views.

Requires the following variables:

- `namespace` - plugin-specific prefix to use in all extension IDs of the template. A best practice is to use reverse domain naming, such as `com.myCompany`, to start the namespace name, followed by a unique extension name. For example, if your company name is Acme, and you create a plug-in for a custom object called Rack, you could use the namespace `com.acme.plugin01.rack`.
- `objectType` - custom object type of the instance. Should be qualified with its own namespace to avoid collisions.

The `objectViewTemplate` creates extension points in the format *namespace.extension-point-name*. To continue the previous example, one extension point might be `com.acme.plugin01.rack.monitorViews`.

For the full list of object workspace extension points, see Custom Object Extension Points. A given tab does not appear in the vSphere Client or vSphere Web Client user interface unless you explicitly create an extension that references that tab's extension point.

Example:

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

vsphere.core.inventorylist.objectCollectionTemplate		Flex Client	HTML Client
	Compatibility	yes	yes

Creates an object collection node in the object navigator for a given custom object type.

Requires the following variables:

- `namespace` - plugin-specific prefix to use in all extension IDs of the template. It must be different than the one in `objectViewTemplate`.
- `title` - custom object title or its resource ID.
- `icon` - 18x18 custom object icon resource ID.
- `objectType` - custom object type of the instance. Should be qualified with its own namespace to avoid collisions.
- `listViewId` - container view ID for the current object collection.
- `parentUid` - extension ID of the category node which the current node belongs to.

Example:

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

Custom Object Extension Points

When you instantiate a `objectViewTemplate` for your custom object, the template creates a number of extension points that you can use to fill out the user interface for the object.

The extension points that are created for a custom object include some of the listed extension points in Object Workspace Extension Points. In addition, the `objectViewTemplate` creates the following list of extension points at runtime for a particular namespace.

You can use the extension points to define views and tabs for the custom object workspace. If you want a specific view or tab to appear in the vSphere Web Client or the vSphere Client user interface for a custom object, you must explicitly create an extension that references the extension point of the view or tab.

${namespace}.views		Flex Client	HTML Client
	Compatibility	yes	yes

Add a top-level tab view for custom objects.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the Getting Started view.
- `categoryUid` - (optional) ID of the category this Getting Started view belongs to.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.

Accessibility: {custom object root}

Example:

```
<extension id="com.vmware.samples.chassisa.MainView">
   <extendedPoint>com.vmware.samples.chassisa.views</extendedPoint>
   <object>
      <name>Chassis Main</name>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/chassisa/resources/chassis-main.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

${namespace}.summaryViews		Flex Client	HTML Client
	Compatibility	yes	yes

Adds an HTML portlet to the Summary tab view for custom objects.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the global view.
- `icon` - (optional) 18x18 portlet icon resource ID.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.
  - `dialogTitle`- portlet title.
  - `dialogSize`- portlet width and height.

Accessibility: {custom object root}

Example:

```
<extension id="com.vmware.samples.chassisa.SummaryView">
   <extendedPoint>com.vmware.samples.chassisa.summaryViews</extendedPoint>
   <object>
      <name>Chassis Summary</name>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/chassisa/resources/chassis-summary.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

${namespace}.monitorViews		Flex Client	HTML Client
	Compatibility	yes	yes

Adds a sub-view to the Monitor tab view for custom objects.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the Monitor view.
- `categoryUid` - (optional) ID of the category this monitor view belongs to.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.

Accessibility: {custom object root} → Monitor page

Example:

```
<extension id="com.vmware.samples.chassisa .monitor">
   <extendedPoint>com.vmware.samples.chassisa.monitorViews</extendedPoint>
   <object>
      <name>Monitor view</name>
      <categoryUid>com.vmware.samples.chassisa.monitor.category</categoryUid>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/vspherewssdk/resources/vm-monitor.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

${namespace}.manageViews		Flex Client	HTML Client
	Compatibility	yes	yes

Adds a sub-view to the Configure tab view for custom objects.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the Configure view.
- `categoryUid` - (optional) ID of the category this Configure view belongs to.
- `componentClass`
  - `url` - relative URL to the HTML page that loads the view content.

Accessibility: {custom object root} → Configure page

Example:

```
<extension id="com.vmware.samples.chassisa.manage">
   <extendedPoint>com.vmware.samples.chassisa.manageViews</extendedPoint>
   <object>
      <name>Configure view</name>
      <categoryUid>com.vmware.samples.chassisa.manage.category</categoryUid>
      <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
         <object>
            <root>
               <url>/vsphere-client/vspherewssdk/resources/vm-configure.html</url>
            </root>
         </object>
      </componentClass>
   </object>
</extension>
```

${namespace}.list.columns		Flex Client	HTML Client
	Compatibility	yes	yes

Creates a new column in the list of custom objects.

Requires a data object of type `com.vmware.ui.lists.ColumnSetContainer` which is a collection of columns with available properties:

- `headerText` - column header text.
- `requestedProperties` - object properties whose value representation will be displayed in the column (commonly a 1-element array).
- `requestedParameters` - parameters of the requested object properties.
- `sortProperty` - enables column sorting by header selection.
- `exportProperty` - enables exporting column data.

Note Only the XML representation is supported.

Accessibility: {custom object list}

Example:

```
<extension id="com.vmware.samples.chassisa.list.sampleColumns">
   <extendedPoint>com.vmware.samples.chassisa.list.columns</extendedPoint>
   <object>
      <items>
         <com.vmware.ui.lists.ColumnContainer>
            <uid>com.vmware.samples.chassisa.column.name</uid>
            <dataInfo>
               <com.vmware.ui.lists.ColumnDataSourceInfo>
                  <headerText>Name</headerText>
                  <requestedProperties>
                     <String>name</String>
                  </requestedProperties>
                  <sortProperty>name</sortProperty>
                  <exportProperty>name</exportProperty>
               </com.vmware.ui.lists.ColumnDataSourceInfo>
            </dataInfo>
         </com.vmware.ui.lists.ColumnContainer>
         ...
      </items>
   </object>
</extension>
```

${namespace}.gettingStartedViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a Getting Started tab view for custom objects.

Requires a data object of type `ViewSpec` with available properties:

- `name` - user-visible name of the Getting Started view.
- `categoryUid` - (optional) ID of the category this Getting Started view belongs to.
- `componentClass` - parameters of the requested object properties.
  - `url` - relative URL to the HTML page that loads the view content.

Accessibility: {custom object root}

${namespace}.monitor.issuesViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view under the Issues second-level tab of the Monitor tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Monitor → Issues

${namespace}.monitor.performanceViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view under the Performance second-level tab of the Monitor tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Monitor → Performance

${namespace}.monitor.performance.overviewViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view under the Performance/Overview section of the Monitor tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Monitor → Performance → Overview

${namespace}.monitor.performance.advancedViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view under the Performance/Advanced section of the Monitor tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Monitor → Performance → Advanced

${namespace}.monitor.taskViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view under the Tasks second-level tab of the Monitor tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Monitor → Tasks

${namespace}.monitor.eventsViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view under the Events second-level tab of the Monitor tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Monitor → Events

${namespace}.manage.settingsViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view under the Settings second-level tab of the Configure tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Configure → Settings

${namespace}.manage.alarmDefinitionsViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view under the Issues/Alarm Definitions section of the Configure tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Configure → Alarm Definitions

${namespace}.manage.permissionsViews		Flex Client	HTML Client
	Compatibility	yes	yes (deprecated)

Adds a sub-view to the Permissions tab view for custom objects.

Requires a data object of type `ViewSpec`.

Accessibility: {custom object root} → Configure → Permissions