Developing Local Plug-ins with the vSphere Client SDK

Update 1
16 October 2018
VMware vSphere 6.7
vSphere Client SDK 6.7

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

Copyright © 2018 VMware, Inc. All rights reserved. Copyright and trademark information.

Contents

About This Book 6

Revision History 7

1 About the vSphere Client 8

- Understanding the vSphere Client Architecture 8
- Overview of the User Interface Layer Components 10
- Understanding Extensibility in the vSphere Client 10
  - Extending the User Interface Layer 11
  - Extending the Java Service Layer 12

2 About the vSphere Client 13

- Knowledge Requirements for Using the vSphere Client SDK 13
- SDK Versions and Compatibility 13
- vSphere Client SDK Contents 14

3 vSphere Client SDK Setup 16

- Software Requirements 16
- Development Environment Requirements Overview 16
- Setting Up for HTML-Based Plug-In Development 17
  - Set Up for Java Development 17
  - Automate the Plug-In Build Process 18
  - Download the vSphere Client SDK 18
  - Set Up the Eclipse Integrated Development Environment (optional) 19
  - Install the vSphere Client Tools Eclipse Plug-In (optional) 21
  - Register Your Local vSphere Client with the vCenter Server Instance 21
  - Configure the Virgo Server in Your Eclipse IDE 23

4 Using the vSphere Client SDK Samples 25

- Location of Sample Plug-in in the vSphere Client SDK 25
- vSphere HTML SDK Sample 25
- Build and Deploy the vSphere HTML SDK Sample Plug-in 26
- Running the vSphere HTML SDK Sample 26

5 Creating a vSphere Client SDK Solution 28

- Before Creating an HTML Plug-In 28
- Creating an HTML Plug-In Project 28
- Generate an HTML Plug-In Project with a Script 29

Create an HTML Plug-In Project with Eclipse 29
Contents of the HTML Plug-In Project Template 30
Building a Plug-In Package from the Project Template 32
Testing the Generated Plug-Ins 33
Deploy the Plug-In on a Local vSphere Client 33
Deploying Your Plug-In on a Remote vSphere Client 34

6 Extension Points in the vSphere Client 36
Global Extension Points 36
Object Navigator Extension Points 38
Object Workspace Extension Points 39
Actions Extension Points 44
Extension Templates 47
Custom Object Extension Points 48
Ordering Extensions 53
Filtering Extensions 53

7 Using the vSphere Client JavaScript API 58
vSphere Client JavaScript API: Modal Interface 58
vSphere Client JavaScript API: Application Interface 59
vSphere Client JavaScript API: Event Interface 61
Example Using the modal API 61

8 Developing HTML-Based User Interface Extensions 63
Overview 63
Global View Extensions 64
Extending the vCenter Object Workspace 65
Extending an Existing Object Workspace 65
Creating an Object Workspace for a Custom Object 67
Creating Extensions to the Summary Tab 67
Creating Data View Extensions 67
Creating Actions Extensions 69
Actions Framework Overview 69
Defining an Action Set 70
Defining Individual Actions for HTML-Based Action Extensions 70
Handling Actions for HTML-Based Action Extensions 73
Handling Locales 74
Guidelines for Creating Plug-Ins Compatible with the vSphere Client 76

9 Developing for the vSphere Client Service Layer 79
Developing Extensions to the Service Layer 79
Understanding the vSphere Client Data Service 79

Overview of Data Service Queries 83
Extending the Data Service with a Data Service Adapter 86
Creating a Custom Java Service 97
Importing a Service in a User Interface Plug-In Module 99

10 Creating and Deploying Plug-In Packages 100
Plug-In Package Overview 100
XML Elements of the Plug-In Package Manifest File 101
Deploying a Plug-In Package 103
Deploying a Plug-In Package From a Remote Server 103
Register a Plug-In Package as a vCenter Server Extension 104
Creating the vCenter Server `Extension` Data Object 105
Verifying Your Plug-In Package Deployment 106
Unregister a Plug-In Package 107

11 Best Practices for Developing Extensions for the vSphere Client 108
Best Practices for Creating Plug-In Packages 108
Best Practices for Plug-In Modules Implementation 110
Best Practices for Developing HTML-Based Extensions 110
Best Practices for Extending the User Interface Layer 111
Best Practices for Extending the Service Layer 112
OSGi-Specific Recommendations 112
`DataService` -Specific Best Practices 114
Best Practices for Deploying and Testing Your vSphere Client Extensions 116

About This Book

The book *Developing Local Plug-ins with the vSphere Client SDK* provides information about developing, deploying and troubleshooting HTML-5 extensions to the vSphere Client user interface.

VMware provides many APIs and SDKs for different applications and goals. This documentation provides information about the extensibility framework of the vSphere Client for developers who are interested in extending the Web application with custom functionality using an HTML-5 user interface.

The information in this book will help you to develop local plug-ins for the vSphere Client. VMware continues to support local plug-ins, which have back ends running as services within the vSphere Client application server. However, for new plug-in development VMware recommends remote plug-ins, which offer superior functional and performance advantages. For more information about remote plug-ins, see the companion book *Developing Remote Plug-ins with the vSphere Client SDK*.

Intended Audience

This information is intended for anyone who wants to extend the vSphere Client with custom functionality in vSphere 6.7. Users typically are software developers who use 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.

Revision History

This *Developing Local Plug-ins with the vSphere Client SDK* is updated with each release of the product or when necessary.

This table provides the update history of *Developing Local Plug-ins with the vSphere Client SDK*.

Revision	Description
16OCT2018	Renamed for vSphere 6.7U1 release, since remote plug-ins are documented in a parallel book.
17APR2018	Initial release.

About the vSphere Client

1

The VMware vSphere® Client provides a means for connecting to VMware vCenter Server® systems and managing the objects in the vSphere infrastructure. The VMware vSphere Client is an HTML5-based web application with a modular architecture that allows you to extend the functionality by means of custom plug-ins.

This chapter includes the following topics:

- Understanding the vSphere Client Architecture
- Overview of the User Interface Layer Components
- Understanding Extensibility in the vSphere Client

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.

User Interface Layer The user interface layer consists of an HTML platform that provides a framework for plug-in 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 and to make changes to your vSphere infrastructure.

The vSphere Client 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.

At the user interface layer vSphere Client plug-ins use the JavaScript API to communicate with the HTML platform components.

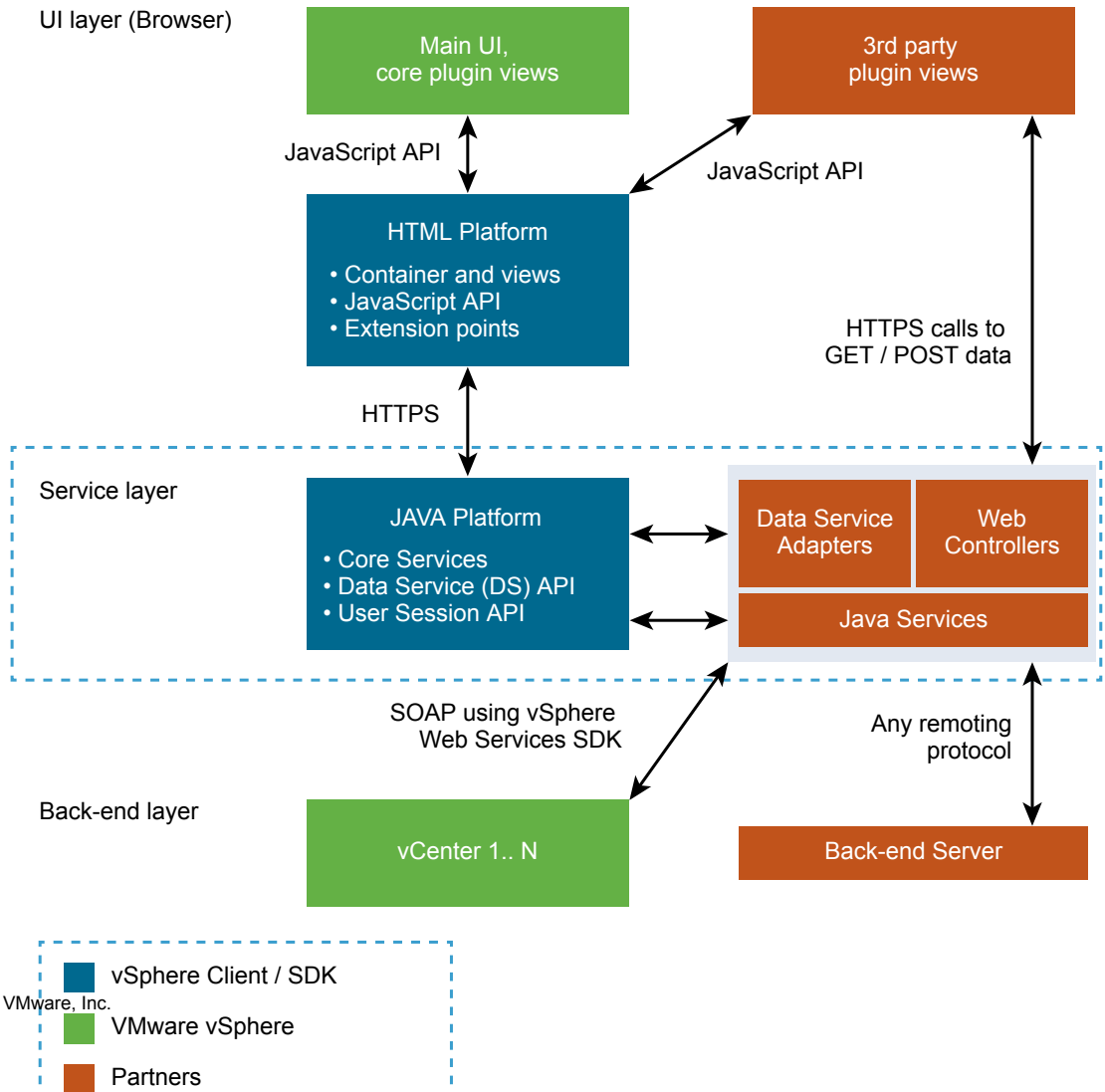
Java Service Layer

The Java service layer provides session management, a data query interface, controller components, and communication with the back end layer. HTML platform components use RESTful API calls over HTTPS to communicate with the Java platform in the service layer.

The service layer is based on the Spring MVC and the OSGI framework. Both the platform services and the vSphere Client plug-ins run in this environment.

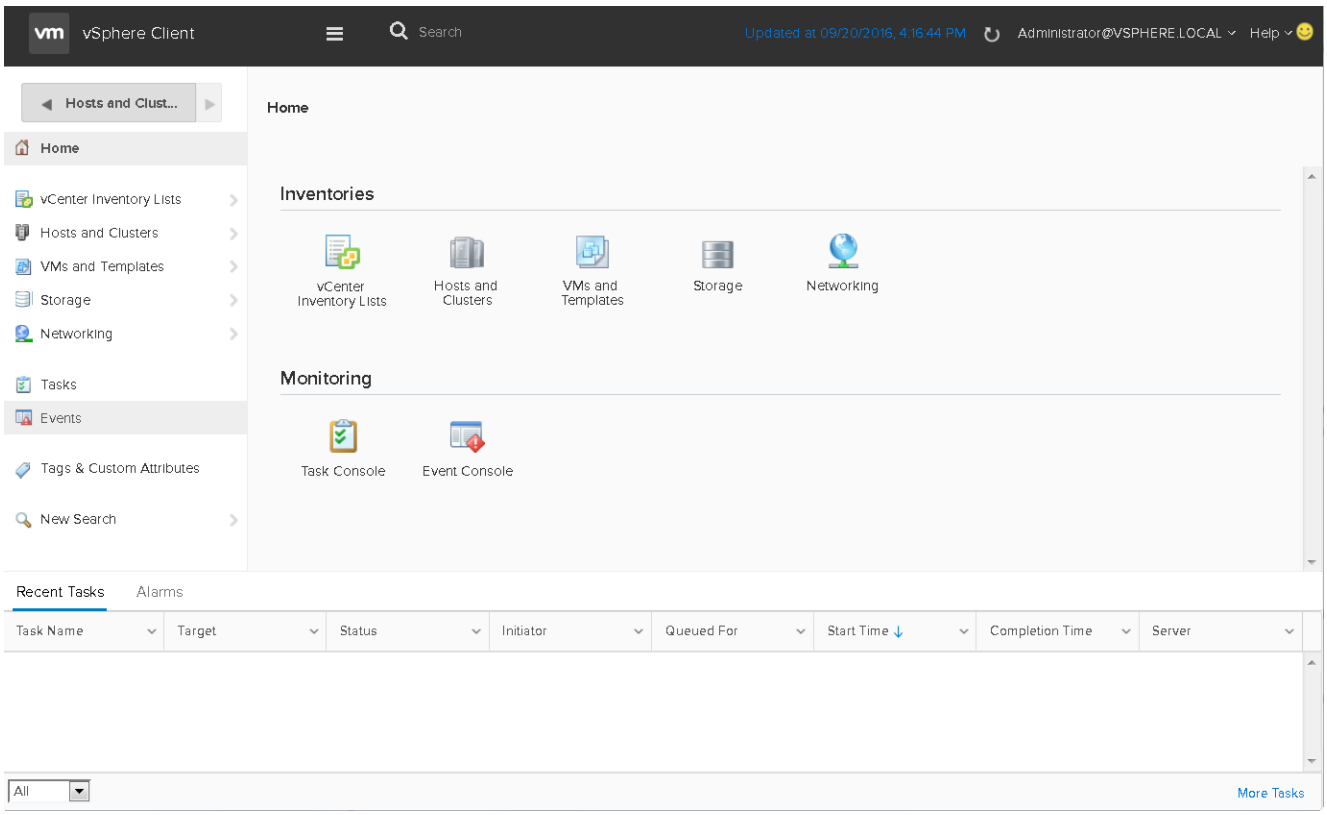
Back End Layer

The back end layer consists of services belonging to VMware vCenter Server, and of services created by third parties. Java components in the service layer use the vSphere Web Services SDK to access one or more instances of vCenter Server, or any custom or standard API to access third party services.



Overview of the User Interface Layer Components

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



Understanding Extensibility in the vSphere Client

The vSphere Client provides a modular architecture that enables plug-in developers to add new user interface elements and business logic to the VMware feature set.

When you add to the user interface layer, you create one or more extensions, which contain the HTML content that you want to visualize. This can be views, menus, or any other UI controls.

When you extend the Java service layer, you create one or more web services that provide data or perform actions on behalf of your UI extensions.

Each plug-in module extends either the user interface layer or the service layer of the vSphere 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 graphical user interfaces.

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

- You extended the vSphere environment 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 with new user interface elements that allow users to observe, monitor, and control these new objects.
- You extended the vSphere 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 can involve creating both user interface plug-in modules and service plug-in modules. For more information about the architecture of the vSphere Client, see Understanding the vSphere Client Architecture.

- Extending the User Interface Layer

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

Extensions to the user interface layer can include either global views or views that are specific to a given object. The latter are called context-specific views. When you create a data view extension, you must also create the actual GUI objects in HTML and package them in the plug-in module. These GUI objects rely on data from the vSphere Client service layers. You can use the libraries included with the vSphere 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, new relations between vSphere objects, and new actions associated with vSphere objects.

Concepts for Extending the User Interface Layer in the vSphere Client

There are three main concepts in vSphere Client UI extensibility.

Extension point	An integration point on the vSphere Client user interface where a plug-in can hook and add its own capability.
Extension	The UI content that you want to visualize. This can be views, menus or any other UI controls.
Extension ID	A unique identifier that you define to refer to your extension.

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 deliver the data to 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 application server, such as the Data Service. You can extend the built-in data service through the Java API from the vSphere Client SDK or create standalone custom Java services for data gathering.

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 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.

About the vSphere Client

2

The VMware vSphere® Client provides a means for connecting to VMware vCenter Server® systems and managing the objects in the vSphere infrastructure. The VMware vSphere Client is an HTML5-based web application with a modular architecture that allows you to extend the functionality by means of custom plug-ins.

This chapter includes the following topics:

- Knowledge Requirements for Using the vSphere Client SDK
- SDK Versions and Compatibility
- vSphere Client SDK Contents

Knowledge Requirements for Using the vSphere Client SDK

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

- The vSphere 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 sample provided within the SDK uses Angular, TypeScript, and the Clarity Design System.

SDK Versions and Compatibility

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

Table 2-1. Compatibility Between the HTML Plug-In Created with a Specific Version of the SDK and the Different Web Client Applications

Version of the SDK That Is Used to Create the HTML Plug-In	vSphere Web Client 6.0	vSphere Web Client 6.5 and vSphere Client 6.5	vSphere Web Client 6.5 U2 and vSphere Client 6.5 U2	vSphere Web Client 6.7 and vSphere Client 6.7
version 6.0	Yes	Yes*	Yes*	Yes
version 6.5	No, if the plug-in uses APIs introduced in 6.5	Yes	Yes	Yes
version 6.5 U2	No, if the plug-in uses APIs introduced in 6.5 or 6.7	No, if the plug-in uses APIs introduced in 6.7 and 6.5 U2	Yes	Yes
version 6.7	No, if the plug-in uses APIs introduced in 6.5 or 6.7	No, if the plug-in uses APIs introduced in 6.7	Yes	Yes

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.

Table 2-2. Compatibility Between JavaScript APIs and vSphere Client Versions

Version of the vSphere Client	vSphere 6.7/6.5 U2 JavaScript API	vSphere 6.0/6.5 Bridge API
version 6.0	No	Yes
version 6.5	No	Yes
version 6.5 U2	Yes	Yes
version 6.7	Yes	Yes

Note The Bridge API is deprecated in the vSphere 6.7 release and the vSphere 6.5 U2 release. VMware will discontinue support for the Bridge API in a future vSphere Client release.

vSphere Client SDK Contents

The vSphere Client SDK contains the following directories to aid developers who create plug-ins.

docs

- Detailed instructions for setting up IDEs.
- Documentation for JavaScript API used by UI components of plug-ins.

	■ FAQ with troubleshooting and development advice for plug-in developers.
libs	Run-time libraries for Spring framework and vSphere API RPC. This directory is not used for remote plug-ins.
samples	A complete remote plug-in sample that demonstrates both client-side and server-side modules, as well as accompanying metadata.
tools	Scripts to assist with development tasks.
vsphere-ui	A complete version of the vSphere Client, both client and server modules.

vSphere Client SDK Setup

3

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

This chapter includes the following topics:

- Software Requirements
- Development Environment Requirements Overview
- Setting Up for HTML-Based Plug-In Development

Software Requirements

You can set up your development environment for developing HTML-based plug-ins 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 Setup 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	For developing HTML plug-ins, download Eclipse Neon.	For more information about how to set up the Eclipse IDE, see Set up the Eclipse Integrated Development Environment.
IntelliJ IDEA	Standard Edition	You can use the IntelliJ IDEA as an alternative to the Eclipse IDE for developing your Java and JavaScript code.

Development Environment Requirements Overview

Before you start setting up your development environment, you must download the vSphere 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 Client plug-in, your development environment must include the following items.

- A development environment capable of developing Web applications by using JavaScript and HTML.
- A development environment capable of developing Java-based Web applications. You can use the Eclipse IDE or IntelliJ IDEA.
- Access to an instance of vCenter Server for Windows or a vCenter Server Appliance instance to register your plug-in. Plug-in registration allows the vSphere Client to download and install the plug-in.

You can set up the vSphere 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, and install and configure the Eclipse IDE or IntelliJ IDEA.

Setting Up for HTML-Based Plug-In Development

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

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

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 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 of 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\jdk1.8.0_10`.
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 a command like the following: `echo export "JAVA_HOME=\$(/usr/libexec/java_home –v 1.8.0_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, so that your plug-in will generate code compatible with older versions of vCenter Server .

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

To use the SDK build scripts inside Eclipse, you can use the Apache Ant version provided with the Eclipse package. The following procedure sets up 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 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.

Download the vSphere Client SDK

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

Prerequisites

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

Procedure

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

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

2. Confirm the md5sum 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 Client SDK must be short and without spaces.

4. Set the `VSPHERE_SDK_HOME` environment variable to the directory on your development machine where you extracted the vSphere Client SDK.

   For example:

   ```
   VSPHERE_SDK_HOME=C:\sdk\html-client-sdk
   ```

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-ui` as a value to the variable.

What to do next

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

Set Up the Eclipse Integrated Development Environment (optional)

You can use an IDE of your choice to develop custom plug-ins for the vSphere Client. The SDK provides an Eclipse plug-in to assist the development process for those who use the Eclipse IDE.

Procedure

1. From the Eclipse Web site at http://www.eclipse.org/downloads/eclipse-packages/, download the Eclipse IDE for Java EE Developers package.

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 increase the heap space and the maximum permanent space used by the JVM.

   You must add or edit the Eclipse initialization file to contain 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.

   a Open the Preferences dialog.

      - On a Microsoft Windows platform, choose **Window > Preferences**.
      - On a Macintosh platform, choose **Eclipse > 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 of the JDK you installed. See Set Up for Java Development.

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

      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 of your SDK.

      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 Client SDK Tools Eclipse plug-in.

Install the vSphere Client Tools Eclipse Plug-In (optional)

The vSphere Client SDK provides an Eclipse plug-in that adds tools and wizards to your Eclipse IDE to ease your HTML plug-in development process. This step is useful 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 Set Up the Eclipse Integrated Development Environment (optional).

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, then click **Open**.
5. Select the vSphere 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 **Install anyway** on the security 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. 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/`
Windows	`ds.properties`	`C:\ProgramData\VMware\vCenterServer\cfg\vsphere-client\config\`
Mac OS		`/var/lib/vmware/vsphere-client/vsphere-client/config/`

5. If you use a Mac OS development environment, edit the `webclient.properties` file and set the `keystore.jks.path` property to point to the `/var/lib/vmware/vsphere-client/store.jks` file.
6. If you use a Mac OS development environment, edit the `ds.properties` file and set the `solutionUser.keyStorePath` property to point to the `/var/lib/vmware/vsphere-client/store.jks` file.

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. Change the value of the `certificateKeystoreFile` attribute of the `<Certificate>` element to `/var/lib/vmware/vsphere-client/store.jks`.

   You can locate the file at *`your_SDK_folder`*`/vsphere-client-sdk/html-client-sdk/vsphere-ui/server/configuration`.

8 Start the local vSphere 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`.

   Note You might need to make the script executable: `chmod +x startup.sh`

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

   Your local vSphere 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 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 Set Up the Eclipse Integrated Development Environment (optional).
- 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
```

Using the vSphere Client SDK Samples

4

The vSphere Client SDK provides a sample to illustrate ways you can extend the vSphere Client. You can use the scripts provided in the SDK to rebuild and run the sample.

This chapter includes the following topics:

- Location of Sample Plug-in in the vSphere Client SDK
- vSphere HTML SDK Sample
- Build and Deploy the vSphere HTML SDK Sample Plug-in
- Running the vSphere HTML SDK Sample

Location of Sample Plug-in in the vSphere Client SDK

When you download the vSphere Client SDK, you can find the sample plug-in in the `SDK/vsphere-client-sdk/html-client-sdk/samples` directory. The sample code is already built and deployed in the Virgo server location: `SDK/vsphere-client-sdk/html-client-sdk/vsphere-ui/plugin-packages`.

The sample plug-in code demonstrates how you can create a custom object and a customized workspace for this object based on your business needs. The sample demonstrate best practices for single entry point, plug-in structure organization such as a Welcome page, a Settings page, and navigation between views. The sample uses Clarity Components and a design similar to the theme of the vSphere Client. The sample also demonstrates the usage of internationalization for different locales. For more information about the Clarity Design System, see https://vmware.github.io/clarity/.

vSphere HTML SDK Sample

The vSphere Client SDK contains a sample plug-in that demonstrates the use of the following extension points.

- vise.navigator.nodespecs
- vise.global.views
- vsphere.core.menus.solutionMenus

- vise.action.sets

Build and Deploy the vSphere HTML SDK Sample Plug-in

The vSphere Client SDK contains a sample plug-in that demonstrates the use of several extension points. Use this procedure after you modify the sample.

After you modify the sample plug-in, you must re-build and re-deploy it. To build the html-sample, execute the following steps from the command line.

Prerequisites

Before you modify the sample, see vSphere Client SDK Setup for instructions to install and configure all required components. In particular, you need the following tools installed:

- `nodejs` 6.9.x or higher
- `npm` 5.x.x
- Angular-CLI

Also you must set the following environment variables:

- Set the environment variable `ANT_HOME` to your Apache Ant folder.
- Set the environment variable `VSPHERE_SDK_HOME` to your vSphere Client SDK folder.

Procedure

1. In a command shell, change to the `html-sample-ui` directory.

   ```
   cd samples/html-sample-ui
   ```

2. Build the modified sample.

   On a MacOS system, run `./build-plugin-package.sh`.

   On a Windows system, run `build-plugin-package.bat`.

   The build output is in `samples/html-sample-ui/target`.

3. Copy the output folders to the plugin-packages directory, confirming that you want to replace existing files.

   ```
   cp -r samples/html-sample-ui/target vsphere-ui/plugin-packages
   ```

   The `vsphere-ui/plugin-packages` directory is where local plug-ins are picked up automatically when the server starts.

Running the vSphere HTML SDK Sample

Use this procedure to run the vSphere HTML SDK sample.

Procedure

1. Start or restart the Virgo server from the command line

   On a MacOS system, run `vsphere-ui/server/bin/startup.sh [-debug]` in a shell window.

   The `-debug` option allows you to specify a debug port, if desired. The default, if no port is specified, is `8000`.

   On a Windows system, run `vsphere-ui/server/bin/startup.bat -debug` in a shell window.
2. Check that there are no errors in the console or in the Virgo logs.
3. Log in to your local HTML Client at https://localhost:9443/ui/.
4. Click the vSphere HTML SDK Sample entry in the **Policies and profiles** section to explore the sample functionality.

What to do next

If you need to modify the logging level, modify the file `html-client-sdk/vsphere-ui/server/configuration/serviceability.xml`. The XML file contains instructions to modify the logging level.

To stop running samples in your local client, delete the sample folders from the `vsphere-ui/plugin-packages` directory and restart the server.

Note Do not delete other plugin packages. Do not delete the entire `plugin-packages` directory.

Creating a vSphere Client SDK Solution

5

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

This chapter includes the following topics:

- Before Creating an HTML Plug-In
- Creating an HTML Plug-In Project
- Generate an HTML Plug-In Project with a Script
- Create an HTML Plug-In Project with Eclipse
- Contents of the HTML Plug-In Project Template
- Building a Plug-In Package from the Project Template
- Testing the Generated Plug-Ins
- Deploy the Plug-In on a Local vSphere Client
- Deploying Your Plug-In on a Remote vSphere Client

Before Creating an HTML Plug-In

Before you create a plug-in, you must set up your development environment to use the vSphere Client SDK.

To set up your development machine, see Setting Up for HTML-Based Plug-In Development.

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 SDK 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. Choose an option 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 Client Tools Eclipse plug-in. For more information about how to set up the Eclipse plug-in, see Set up the Eclipse Integrated Development Environment.

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 SDK 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.

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.sh` 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 default 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. 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 Client Tools Eclipse plug-in.

Prerequisites

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

- Verify that you have the vSphere Client Tools Eclipse plug-in installed and configured in the Eclipse IDE. See Install the vSphere 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 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 for the project name. The project name is used to create 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.

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.

UI Project Template Structure

- **myplugin-id/src/**
  - **app/** Main source files for the plug-in user interface, including Javascript and Typescript files.
  - **assets/**
    - **css/** CSS files used in the plug-in. The `css` folder contains the `plugin-icons.css` file that you can use to define the external icons.
    - **i18n/** Localized resources used in the plug-in.
    - **images/** Images used in the plug-in.
  - **main/webapp/**
    - **plugin.xml** Manifest file of the plug-in. Defines extensions and resources.
    - **META-INF/**
      - **MANIFEST.MF** Manifest file of the WAR bundle.
    - **WEB-INF/spring/** Spring configuration.
    - **bundle-context.xml** Declares the service that this UI bundle uses.

Java Service Project Template Structure

- **myplugin-service/**
  - **build-java.bat** Windows script to generate the Java service bundle.

    build-java.sh MacOS script to generate the Java service bundle.

    build-java.xml `ant` script to generate the Java service bundle.
  - **src/main/**
    - **resources/META-INF/**
      - **MANIFEST.MF** The bundle manifest file.
      - **spring/** Spring configuration files.

- **java/com/mycompany/myplugin/**
  - **services/** Interfaces and implementations for plug-in services.
  - **model/** Data models for plug-in services.
  - **controllers/** Controllers for plug-in services.

Building a Plug-In Package from the Project Template

You build an HTML plug-in from the plug-in project template by using the automation scripts provided with the SDK.

To build 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 you run 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.

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="6.5.0" />
      <pluginPackage id="com.vmware.vsphere.client.html" version="6.5.0" />
   </dependencies>
   <bundlesOrder>
      <!-- Include a 3rd-party library (for example gson) -->
      <bundle id="com.google.gson" />
      <!-- Include my plug-in modules -->
      <!-- These are example IDs; prefix should match the package ID -->
      <bundle id="com.mycompany.myplugin.myplugin-service" />
      <bundle id="com.mycompany.myplugin.myplugin-ui" />
   </bundlesOrder>
</pluginPackage>
```

Follow these recommendations for the `plugin-package.xml` file to ensure that your plug-in can be deployed on the vSphere Client:

- 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 and the `com.vmware.vsphere.client.html` package, version `6.5.0`. This dependency ensures that your plug-in can be deployed on the vSphere Client 6.5.
- To specify an Update release as the minimum version supported by your plug-in, you need to use a special numbering system. For example, to specify that your plug-in supports only 6.5 Update 2 or above, use the version `6.5.0.20000` for the dependency.

Testing the Generated Plug-Ins

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

Deploy the Plug-In on a Local vSphere Client

This procedure describes how you can use the `pickup` directory to speed up your development process. You can repeat the steps for each new version of the UI and Java service components of your plug-in.

Using the `pickup` during development is convenient for debugging, but it imposes a performance penalty in production.

Prerequisites

- 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.
- Set the option `pickup.deployer=true` in the `webclient.properties` file.

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. 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`.

   The string resources are reloaded when you restart the Virgo server.

3 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.

4 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 remote vSphere Client.

Deploying Your Plug-In on a Remote vSphere Client

You can verify whether your custom plug-in runs as expected by deploying the plug-ins on a remote vSphere 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.
- Use the vCenter Server plug-in registration tool provided with the vSphere Client SDK.

vCenter Server Plug-In Registration Tool

The vSphere Client SDK provides a tool to ease the registration of custom plug-ins with the vSphere 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 plug-in extension to vCenter Server.

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 5-1. 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`** ■ **`updatePlugin`**
`-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** This option is not supported by 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 administrator -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 to verify that the plug-in is visible in the remote vSphere Client. You can also use the MOB of your vCenter Server instance to view all registered plug-ins.

Extension Points in the vSphere Client

6

The vSphere 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 chapter includes the following topics:

- Global Extension Points
- Object Navigator Extension Points
- Object Workspace Extension Points
- Actions Extension Points
- Extension Templates
- Custom Object Extension Points
- Ordering Extensions
- Filtering Extensions

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

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.
- `contentSpec`
  - `url` - relative URL to the HTML page that loads the view content.
  - `metadata` (optional)
    - `key` - "hasTitle"
    - `value` - "false" opens an empty iframe, without a title.

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>
      <contentSpec>
         <url>/ui/globalview/resources/mainView.html</url>
         <metadata><entry><key>hasTitle</key><value>false</value></entry></metadata>
      </contentSpec>
   </object>
</extension>
```

vise.home.shortcuts **deprecated**

Adds 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.

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>
```

vsphere.core.objectTypes	deprecated

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

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 view extension to navigate to when the node is selected.
- `viewOpenedUponFocus` - (optional) open a new empty object navigator for this view.
- `parentUid` - ID of the parent extension this node will be displayed in. This can be another `vise.navigator.nodespecs` extension ID defined by your plug-in or it can be "`vsphere.core.navigator.solutionsCategory`". Accessibility: Object Navigator root.

Example:

```
<extension id="com.vmware.samples.entryPoint">
   <extendedPoint>vise.navigator.nodespecs</extendedPoint>
   <object>
      <title>ChassisA Category</title>
      <parentUid>vsphere.core.navigator.solutionsCategory</parentUid>
      <navigationTargetUid>com.vmware.samples.htmlsample.welcomeView</navigationTargetUid>
   </object>
</extension>
```

vise.inventory.representationspecs	deprecated

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>
```

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

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.
- `contentSpec`
  - `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>
      <contentSpec>
         <url>/ui/vspherewssdk/resources/vm-summary.html</url>
         <dialogTitle>WSSDK Summary Sample</dialogTitle>
         <dialogSize>440,400</dialogSize>
      </contentSpec>
   </object>
</extension>
```

vsphere.core.${objectType}.monitorCategories

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

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.
- `contentSpec`
  - `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>
      <contentSpec>
         <url>/ui/vspherewssdk/resources/vm-monitor.html</url>
      </contentSyec>
   </object>
</extension>
```

vsphere.core.${objectType}.manageCategories

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

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.
- `contentSpec`
  - `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>
      <contentSpec>
         <url>/ui/vspherewssdk/resources/vm-configure.html</url>
      </contentSpec>
   </object>
</extension>
```

vise.relateditems.specs	deprecated

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	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	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	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.${objectType}.list.columns	deprecated
Creates a new column in the list of vSphere objects of type `${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 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

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.
- `className` - accepts the following classes:
  - `com.vmware.vsphere.client.HtmlPluginModalAction` - opens a modal dialog by using the JavaScript API method `modal.open()`.
  - `com.vmware.vsphere.client.HtmlPluginHeadlessAction` - initiates a function call with no associated UI view.
  - `com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate` - opens a modal dialog or initiates a headless function using the deprecated `htmlbridge` JavaScript API.

Accessibility: {object} → {menu} → {plugin sub-menu}

Example:

```
<extension id="com.vmware.samples.htmlsample.vmActionSet">
  <extendedPoint>vise.actions.sets</extendedPoint>
  <object>
    <actions>
      <com.vmware.actionsfw.ActionSpec>
        <uid>com.vmware.samples.htmlsample.vm.action</uid>
        <label>#{action1.label}</label>
        <delegate>
          <className>com.vmware.vsphere.client.HtmlPluginModalAction</className>
            <object><root>
              <actionUrl>/ui/html-sample/index.html?view=vm-action-modal</actionUrl>
              <dialogTitle>#{actionModelTitle}</dialogTitle>
              <dialogSize>500,250</dialogSize>
            </root></object>
        </delegate>
      </com.vmware.actionsfw.ActionSpec>
    </actions>
  </object>
  <metadata>
    <objectType>VirtualMachine</objectType>
  </metadata>
</extension>
```

vmware.prioritization.listActions	deprecated

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 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

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 Client user interface consistent.

vsphere.core.inventory.objectViewTemplate	deprecated

Creates a complete object workspace for a given custom object type. When you create an instance of the `objectViewTemplate`, the vSphere 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 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	deprecated

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 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	**deprecated**

Adds 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.
- `contentSpec`
  - `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>
      <contentSpec>
         <url>/ui/chassisa/resources/chassis-main.html</url>
      </contentSpec>
   </object>
</extension>
```

${namespace}.summaryViews	**deprecated**

Adds a 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.
- `contentSpec`
  - `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>
      <contentSpec>
         <url>/ui/chassisa/resources/chassis-summary.html</url>
      </contentSpec>
   </object>
</extension>
```

${namespace}.monitorViews	deprecated

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.
- `contentSpec`
  - `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>
      <contentSpec>
         <url>/ui/vspherewssdk/resources/vm-monitor.html</url>
      </contentSpec>
   </object>
</extension>
```

${namespace}.manageViews	deprecated

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.
- `contentSpec`
  - `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>
      <contentSpec>
         <url>/ui/vspherewssdk/resources/vm-configure.html</url>
      </contentSpec>
</extension>
```

${namespace}.list.columns	deprecated

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	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.
- `contentSpec` - 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	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	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	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	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	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	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	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	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	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	

Ordering Extensions

You can use the <precedingExtension> element to specify the order in which the vSphere 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 = "com.MyPluginPackage.MyPlugin.UtilityView">
   <extendedPoint>vsphere.core.vm.views</extendedPoint>
   <precedingExtension>com.MyPluginPackage.MyPlugin.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 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.

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 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.

Limitations on Filtering Based on User Privilege Level

Filtering has no effect on nodespecs and shortcut items. To avoid impacting the load time of the object navigator, home screen, and main menu, the vSphere Client does not check permissions for nodespecs and shortcut items.

Plug-ins should show the global view if the user has the appropriate privilege. If not, the user interface should display a message informing the user of the lack of privilege.

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 the vSphere Client JavaScript API

7

The vSphere Client provides several interfaces that your plug-in can use to communicate with the HTML5 platform. These JavaScript methods are documented here as if they have TypeScript signatures, but they run as pure JavaScript, and all complex types are plain old Javascript objects.

Each plug-in runs in an iframe that has the same origin as the vSphere Client.

This chapter includes the following topics:

- vSphere Client JavaScript API: Modal Interface
- vSphere Client JavaScript API: Application Interface
- vSphere Client JavaScript API: Event Interface
- Example Using the modal API

vSphere Client JavaScript API: Modal Interface

The `modal` interface enables your plug-in to manage modal dialog windows.

`modal.open` method

Signature	`modal.open(configObj:ModalConfig):void`
Description	Opens a modal dialog box specified by the `configObj` parameter.
Parameter: `configObj`	Specifies the properties of this modal dialog box.

`modal.close` method

Signature	`modal.close(data:any):void`
Description	Closes the modal dialog box in the parent iframe.
Parameter: `data`	Passes `data` to the callback function specified by `onClosed` property at dialog open.

`modal.setOptions` method

Signature	`modal.setOptions(options:DynamicModalOptions):void`
Description	Modifies some properties for a modal dialog box in the parent iframe.
Parameter: `options`	Specifies values for some dialog box properties.

`modal.getCustomData` method

Signature	`modal.getCustomData():any`
Description	Returns the `customData` object provided when a modal dialog box was opened, or `null` if no `customData` object was provided.

`modal.DynamicModalOptions` type

Description	Specifies values for some properties of a modal dialog box.

Name	Type	Required?	Notes
`title`	`string`	no	Dialog title. May not contain an icon. (If not present, no change to dialog title.)
`height`	`number`	no	Dialog size. Specified in pixels. (If not specified, no change to dialog height.)

`modal.ModalConfig` type

Description	Specifies the properties of a modal dialog box.

Name	Type	Required?	Notes
`url`	`string`	yes	Location of HTML content for the dialog.
`title`	`string`	no	Dialog title. May not contain an icon. (default=`''`)
`size`	`(width:number, height:number)`	yes	Dialog size. Specified in pixels.
`closable`	`boolean`	no	Whether the dialog displays a close button. (default=`true`)
`onClosed`	`callbackFn`	no	Function runs when the dialog closes.
`customData`	`any`	no	Data the calling module passes to the dialog.
`contextObjects`	`string[]`	no	IDs of relevant objects the calling module passes to the dialog.

vSphere Client JavaScript API: Application Interface

The *app* interface helps your plug-in navigate and control the vSphere Client user interface.

`app.getContextObjects` method

Signature	`app.getContextObjects():any[]`	
Description	Returns the current context objects.	
Return value:	**for global view**	Returns empty array. Global views have no context.
	for vSphere object	Returns a context item for the associated vSphere object.
	for dialog opened by `modal.open()`	If dialog opened by `htmlSdk.modal.open(configObj)`, returns value of `configObj.contextObjects` (or empty array, if `contextObjects` undefined)
	for dialog opened by `plugin.xml` actions	If dialog opened by action defined in `plugin.xml`, returns an array of context items.
	A context item is a JavaScript object containing a single property, `id:string`. This is the ID of the associated vSphere object.	

`app.navigateTo` method

Signature	`app.navigateTo(options:NavigationOptions):void`
Description	Navigates to a specified view, and optionally passes custom data to the view.
Parameter: `options`	Specifies the destination view and custom data.

`app.getNavigationData` method

Signature	`app.getNavigationData():any`
Description	Returns the custom data passed to the view by the `navigateTo()` method. (If no custom data passed, returns `null`.)

`app.getClientInfo` method

Signature	`app.getClientInfo():ClientInfo`
Description	Returns type and version info for the vSphere Client.

`app.getClientLocale` method

Signature	`app.getClientLocale():string`
Description	Returns the current locale of the vSphere Client.

`app.ClientInfo` type

Description	Documents type and version of vSphere Client.

Name	Type	Required?	Notes
`type`	`string`	info only	

Name	Type	Required?	Notes
`version`	`string`	info only	

`app.NavigationOptions` type

Description	Specifies a destination view and custom data for the view.

Name	Type	Required?	Notes
`targetViewId`	`string`	yes	ID of the destination view.
`objectId`	`string`	no	ID of any object associated with the view. (For a global view, this field is not required.)
`customData`	`any`	no	A custom data structure passed to the view.

vSphere Client JavaScript API: Event Interface

The `event` interface helps your plug-in with event management.

`event.onGlobalRefresh` method

Signature	`event.onGlobalRefresh(pluginCallbackFunc:function):void`
Description	Registers a global refresh handler that the vSphere Client will call when the **Global Refresh** button is clicked.
Parameter: `pluginCallbackFunc`	A reference to a global refresh handler.

Example Using the modal API

modal.html

```
<html>
  <head>
    <script src="http://code.jquery.com/jquery-latest.min.js"
            type="text/javascript"></script>
    <script type='text/javascript'>
      function handler(event)
      {
        var choice = $('input[name=heads_or_tails]:checked').val();
        htmlSdk.modal.setOptions({title: choice});
        settimeout(function(){htmlSdk.modal.close(choice);}, 3000);
      }
    </script>
  </head>
  <body>
    <form name='flip' onSubmit='return handler()'>
      <p><input type='radio' name='heads_or_tails' value='HEADS' />HEADS</p>
      <p><input type='radio' name='heads_or_tails' value='TAILS' />TAILS</p>
```

```
      <input type='submit' name='submit' value='Submit' />
    </form>
  </body>
</html>
```

modal.js

```
# Set vSphere Client JS API namespace.
htmlSdk = window.frameElement.htmlclientSdk;

# Select correct answer.
correct = ['heads', 'tails'][2*Math.random()-1];

# Create callback function.
checker = function(choice){
  var correct = htmlSdk.modal.getCustomData();
  if (choice === correct) {
    alert('You chose wisely.');
  } else {
    alert('Sorry, you lose.');
  }}

# Configure modal dialog.
var config ={
  url: "example/dialog.html",
  title: 'Choose!',
  size: { width: 490, height: 240 },
  onClosed: checker,
  customData: correct}

# Open modal dialog.
htmlSdk.modal.open(config);
```

Developing HTML-Based User Interface Extensions

8

The vSphere Client is a Web browser-based application that provides an 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 UI features by creating user interface extensions. A UI plug-in contains one or more extensions, which add UI elements to the vSphere Client user interface.

This chapter includes the following topics:

- Overview
- Global View Extensions
- Extending the vCenter Object Workspace
- Creating Data View Extensions
- Creating Actions Extensions
- Handling Locales
- Guidelines for Creating Plug-Ins Compatible with the vSphere Client

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 SDK to develop HTML plug-ins that are compatible with the vSphere Client.

The vSphere Client SDK provides the following features:

- You can use the JavaScript libraries of your choice to develop the user interface components of your extensions.
- You can examine the sample HTML plug-in provided with the vSphere Client SDK. The sample demonstrates how you can add different extensions to the vSphere Client.
- You can create extensions to the Java service layer by using Java APIs provided in the SDK.

Accessing Data

Each global view extension is an independent HTML element that must communicate with the plug-in back-end service or the vSphere environment to retrieve data, or send commands, that the view requires. The vSphere Client SDK includes a JavaScript library that you can use when creating UI extensions. The JavaScript API provides access to user interface data such as a list of objects selected in the UI, locale of the vSphere Client, and navigation to a specified view.

The JavaScript code can use REST-based Ajax queries to retrieve data from the plug-in back-end service to retrieve data that the plug-in displays in the UI view. Alternatively, a user interface plug-in can redirect to another view by submitting HTML forms.

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 brings together status information about vSphere objects from different vCenter Servers.

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 need only the view name and the content URL.

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.

Extending the vCenter Object Workspace

The vSphere Client displays a standard object workspace for each type of vSphere object. Your plug-ins can extend the object navigator with new categories, such as **Settings**, **Custom Objects List**, and so on.

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-views and detailed views within the **Monitor** and **Configure** 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 plug-in specific object navigator item or entry point.

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.

To implement a new workspace, you add an object navigator item that links to a global view extension. Within the global view extension, you have the freedom to implement any view structure you want, including a view with tabs and nested views.

When you create an object workspace, use XML extension templates as demonstrated in the sample code.

Extending an Existing Object Workspace

To add HTML extensions to the **Monitor** and **Configure** tabs in the vSphere Client, use the following generic extension points. These extension points generate a subordinate view inside that tab.

- `vsphere.core.${objectType}.monitorViews`
- `vsphere.core.${objectType}.manageViews`

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.
- Portlet - If you define the portlet extension point, a data view extension appears as a portlet in the object workspace.

Configure and Monitor Views Extensions

You can extend a vSphere object view under the **Configure** and **Monitor** tabs by using the generic extension points `vsphere.core.${objectType}.monitorViews` and `vsphere.core.${objectType}.manageViews` 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.

```
<extension id="com.vmware.samples.vspherewssdk.host.monitor">
   <extendedPoint>vsphere.core.host.monitorViews</extendedPoint>
   <object>
      <name>#{monitorHtml.label}</name>
      <contentSpec>

         <url>/ui/plugin-name/monitor-view.html</url>
      </contentSpec>
   </object>
</extension>
```

The value of the `<url>` property is a relative URL that starts with the Web context path of the plug-in, `/ui/`. 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.

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

- `objectId` - The context object ID of the view.
- `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 provided extension templates.

For more information about the extension templates, see Extension Templates.

Creating Extensions to the **Summary** Tab

To create vSphere object views, you add portlets to the **Summary** tab.

You can add a portlet to the **Summary** tab of a vSphere object by using the `vsphere.core.{objectType}.summarySectionViews.html` HTML-specific extension point.

Adding Portlets to the Summary Tab

You create portlet views at the `<namespace>.summarySectionViews.html` extension point by using the generic `HtmlView` component class.

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>
      <contentSpec>
         <url>/ui/vspherewssdk/resources/host-summary.html</url>
         <dialogTitle>WSSDK Summary Sample</dialogTitle>
         <dialogSize>440,400</dialogSize>
      </contentSpec>
   </object>
</extension>
```

Creating Data View Extensions

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 `html-sample` in the SDK 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 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 might need to 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.

The <delegate> object requires a `<className>` property and an `<object>` element that contains only an embedded `<root>` element. The `<className>` property specifies one of the following, depending on whether the action is modal or headless:

- `com.vmware.vsphere.client.HtmlPluginHeadlessAction` for a headless action
- `com.vmware.vsphere.client.HtmlPluginModalAction` for a modal (UI) 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.
`<closable>`	boolean	Hides the top right close button for the dialog. The default value of this property is `true`.

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 specifying `com.vmware.vsphere.client.HtmlPluginHeadlessAction` as the delegate class name. of the action extension point.

- The value of the `actionUrl` parameter has the following form.

```
/ui/html-sample/rest/vm-headless-ction
```

- 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.

```
<!-- Plugin Action set -->
<extension id="com.vmware.samples.htmlsample.vm.actionSet">
  <extendedPoint>vise.actions.sets</extendedPoint>
   <object>
      <actions>
...
         <!-- Plugin Headless Action -->
         <com.vmware.actionsfw.ActionSpec>
            <uid>com.vmware.samples.htmlsample.vm.headless.action</uid>
            <label>#{vmHeadlessActionLabel}</label>
            <delegate>
               <className>com.vmware.vsphere.client.HtmlPluginHeadlessAction</className>
               <object><root>
                  <actionUrl>/ui/html-sample/rest/vm-headless-action</actionUrl>
               </root></object>
            </delegate>
         </com.vmware.actionsfw.ActionSpec>
...
      </actions>
   </object>
   <metadata>
      <objectType>VirtualMachine</objectType>
   </metadata>
</extension>
```

When the headless action is invoked the JavaScript API 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.

When you define a UI action, you can supply an additional property for the `<delegate>` class to specify whether the dialog displays an **X** button to close the dialog. The default is to display an **X** button. To suppress the **X** button, add the `closable` property with a value of `false`.

Example: HTML-Based UI Action Extension Definition

The following example shows an extension definition for an HTML-based UI action extension.

```
<!-- Plugin Action set -->
<extension id="com.vmware.samples.htmlsample.vm.actionSet">
  <extendedPoint>vise.actions.sets</extendedPoint>
   <object>
      <actions>
...
         <!-- Plugin UI Action -->
         <com.vmware.actionsfw.ActionSpec>
            <uid>com.vmware.samples.htmlsample.vm.modal.action</uid>
            <label>#{vmUiActionLabel}</label>
            <delegate>
               <className>com.vmware.vsphere.client.HtmlPluginModalAction</className>
               <object><root>
                  <actionUrl>/ui/html-sample/index.html?view=vm-modal-action</actionUrl>
                  <dialogTitle>#{vmActionModalTitle}</dialogTitle>
                  <dialogSize>600.250</dialogSize}
                  <closable>false</closable>
               </root></object>
            </delegate>
         </com.vmware.actionsfw.ActionSpec>
...
      </actions>
   </object>
   <metadata>
      <objectType>VirtualMachine</objectType>
   </metadata>
</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.

- `locale` - The current locale that is used.

After the dialog form is submitted or the operation is canceled, the code calls `modal.close(data)`.

Handling Actions for HTML-Based Action Extensions

When you create an HTML-based action extension to the vSphere 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

```
...
/**
  * Perform headless action on a VirtualMachine vSphere object.
  */
   @RequestMapping( value="vm-headless-action",
                    method = RequestMethod.POST)
   @ResponseBody
public void vmHeadlessAction() {
   // Implement your logic here to trigger an action on a virtual machine.
            }
...
```

Handling Locales

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 `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 retrieve the current client locale by using the app.getClientLocale() method from the JavaScript API. You can use the locale information to localize your plug-in UI with a framework of your choice. For an example of localizing a plug-in UI, see the HTML sample plug-in included with the SDK.

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:

- 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 `/ui` root path to the URLs.
- Use the `ui` 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 HTML sample plug-in is defined in the manifest file as follows:

```
Web-ContextPath: ui/html-sample
```

The root path for resources and data requests for the vSphere Client starts with `ui`.

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 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>
```

Developing for the vSphere Client Service Layer

9

User interface elements in 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.

Developing Extensions to the Service Layer

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 and user interface layer, such as 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 Client Architecture.

Understanding the vSphere 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.

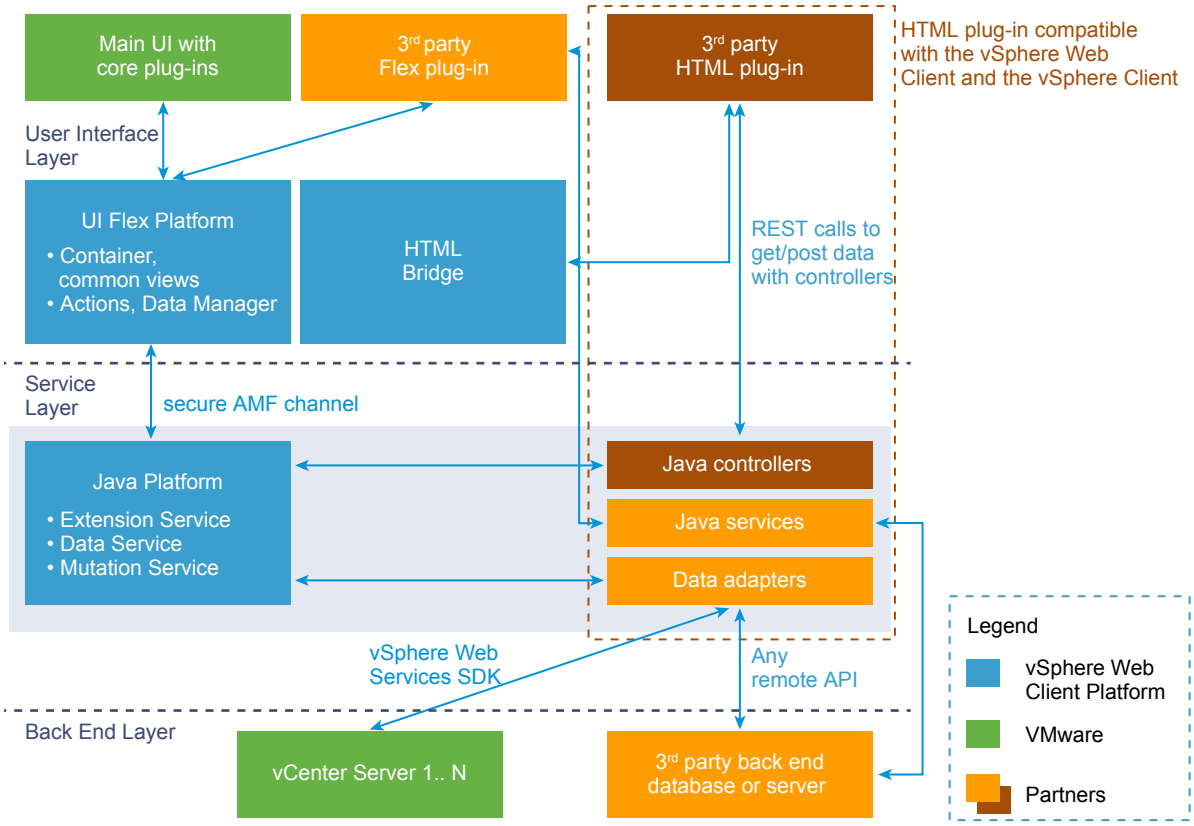
User interface components, such as Flex data views, act as Data Service clients. These clients retrieve information by creating Data Service queries. The Data Service processes each query and returns a set of result objects.

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.

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 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.

Creating and Deploying Plug-In Packages

10

Each plug-in package contains 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 chapter includes the following topics:

- Plug-In Package Overview
- XML Elements of the Plug-In Package Manifest File
- Deploying a Plug-In Package

Plug-In Package Overview

A plug-in package is a ZIP archive file that contains all of the plug-in modules in your solution along with a package manifest.

The package manifest describes deployment information for each plug-in module using XML metadata. The vSphere 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 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. The source code that corresponds with this manifest file is available in the HTML sample in the SDK, at `html-client-sdk/vsphere-ui/plugin-packages/`.

```
<pluginPackage id = "com.MyCompany.myPackage"
version="1.0.0"
type="html"
name="My Plugin Name"
description="Demo package version 1"
vendor="My Company"
iconUri="assets/packageIcon.png">

<dependencies>
<pluginPackage id = "com.vmware.vsphere.client" version="6.7.0" />
<pluginPackage id = "com.vmware.vsphere.client.html" version="6.7.0" />
</dependencies>

<bundlesOrder>

<bundle id="com.mySolution.myUI" />
<bundle id="com.mySolution.myService" />
</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 plug-in package.

Attribute Name	Description
`id`	The unique package identifier that you define. A best practice is to use namespace notation, such as `com.myCompany.MyPluginID`. Must match the vCenter Server extension key.
`version`	A dot-separated string containing the plug-in version number, such as `1.0.0`. Must match the vCenter Server extension version.
`type`	Must be `html`.
`description`	A short description of the plug-in.

Attribute Name	Description
vendor	The name of the plug-in 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 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 Client plug-in depends on packages with specific versions and might not be compatible with later versions of these 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.

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 Client:

```
...
<dependencies>
   <pluginPackage id="com.vmware.vsphere.client"
                  version="6.7.0" match="greaterOrEqual" />
   <pluginPackage id="com.vmware.vsphere.client.html"
                  version="6.7.0" match="greaterOrEqual" />
</dependencies>
...
```

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

<bundlesOrder> Element

The <bundlesOrder> element specifies the order in which locally hosted plug-in modules are deployed to the vSphere Client. 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.

Deploying a Plug-In Package

You deploy a plug-in package to the vSphere 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 Client that connects to your vCenter Server.

You must register your plug-in on every vCenter Server where you need to use it. When a vSphere Client connects to a vCenter Server where your plug-in is not registered, the plug-in is not visible to the vSphere Client.

When a vSphere Client establishes a user session to a vCenter Server instance, the vSphere 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 Client application server are downloaded and deployed.

The vSphere 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 with the newer version.

Deploying a Plug-In Package From a Remote Server

The plug-in package ZIP file that represents a vSphere Client plug-in 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 vCenter Server extension definition. When you register a plug-in as an extension with a vCenter Server instance, the plug-in package ZIP file is downloaded from the remote URL.

The vSphere 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 Client Service, VMware Directory Service, Security Token Service and Syslog Collector Service. The TLS protocol versions that you configure for the vCenter Server service must be the same as the protocol versions for all other services.

The default TLS protocol version for vSphere 6.7 is 1.2. For more information about supported TLS versions and configurations, see https://kb.vmware.com/s/article/2145796.

Note Make sure that the Web server that hosts your vSphere Client plug-ins supports the same TLS protocol versions that are configured for the vSphere services. If this requirement is not met, the vSphere 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. For more information about the vCenter Server plug-in registration tool, see Register Your Local vSphere Client with the vCenter Server Instance.
- 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 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 Client plug-in.
6. Click **Invoke Method** to register the plug-in as a vCenter Server 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:

- Log in to the vSphere Client, go to **Administration**, and under **Solutions**, select **Client Plug-Ins**.

- Log out and log in again to the vSphere Client. The vSphere 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 Client plug-ins, you might need to restart the vSphere 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>MyCompany</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>MyCompany</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>MyCompany</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 Client. You can verify that the deployment procedure is successful by using the log files of the Virgo server, which are available at `/var/log/vmware/vsphere-ui`.

You can verify that your plug-in package is deployed correctly by searching the log file of the vSphere 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 Client caches the downloaded plug-in package in `/etc/vmware/vsphere-ui/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 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 deployed locally on the vSphere 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 Client Plug-In Management view. This behavior is a known issue, and a workaround is to restart the Virgo server.

Best Practices for Developing Extensions for the vSphere Client

11

You can improve your extension solutions by understanding the process of extending the user interface layer and service layer of the vSphere 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 Client extensions.

This chapter includes the following topics:

- Best Practices for Creating Plug-In Packages
- Best Practices for Plug-In Modules Implementation
- Best Practices for Developing HTML-Based Extensions
- Best Practices for Extending the User Interface Layer
- Best Practices for Extending the Service Layer
- Best Practices for Deploying and Testing Your vSphere Client Extensions

Best Practices for Creating Plug-In Packages

To meet the requirements of your virtual environment, you must extend the capabilities of the vSphere 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 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 Client SDK to develop your vSphere 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 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 `html-client-sdk/server/repository` and `html-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 Client.
- For best performance, when designing your vSphere 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. Otherwise, after the vSphere 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 Client.

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

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

  Note If your plug-in package is only compatible with a specific version of the vSphere 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 Client, your plug-in package is not deployed, and does not cause any errors.

  Note If the `match` attribute is not provided, the default value is `greaterOrEqual`.

- To avoid deployment failures, you must create a ZIP archive file for your vSphere Client extension. Moreover, if you want to complete successfully the certification for your vSphere 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.

- Your plug-in package must be OS agnostic. You must avoid reading and writing on the file system from the vSphere Client service layer. In case you need to temporarily store files, you must use the browser cache or your own back-end server.
- To provide a consistent end-user experience in case your vSphere 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.
- Avoid using deprecated or private APIs and extension points. Using deprecated APIs in your vSphere Client extensions will prevent them from working with future versions of the vSphere Client.
- To prevent performance problems in the vSphere 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 Client remains stateless. To ensure the scalability of the vSphere Client, you must use your backend server to cache data.
- 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.
- To achieve optimal scalability and performance for your vSphere 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 Client SDK to create HTML-based extensions.

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 SDK is provided to developers that want to create HTML5-based extensions for the vSphere Client.

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`.

Best Practices for Extending the User Interface Layer

When developing extensions for the user interface layer of the vSphere 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 Client.
- When you create 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.
- Use the REST API for retrieving data from the service layer. 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 request through the proxy.
- If you 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 Client to display the plug-in by using the current vSphere Client locale.

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

    ```
    <plugin id="com.vmware.samples.htmlsample"
       defaultBundle="com_vmware_samples_htmlsample">

       <resources baseUrl="locales/">
          <resource>com_vmware_samples_htmlsample</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. Use your company name and product name as part of the resource bundle name to make it unique.
  - 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 your environment has 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 Client bundles, set the package version to `0` in the `MANIFEST.MF` file. When you update the vSphere Client platform, your bundle might stop working if you specified a 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 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 11-1 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 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 `ClassNotFoundExceptions` 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 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 extension, you must not use any custom properties defined by the vSphere Client modules. These properties are prone to change in the future. You must use only the properties defined in the vSphere Web Services 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 Client Extensions

After you develop your vSphere Client extension, you can follow these recommendations to ensure that your extension is successfully deployed to 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 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 vSphere Client 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. You must also manually delete the cached files of the plug-in package that are stored on the Virgo server in one of the following locations:

Virgo Server Environment	Location of Cached Packages
vCenter Server Appliance	/etc/vmware/vsphere-ui/vc-packages/vsphere-client-serenity/
Windows OS local development environment	%PROGRAMDATA%\VMware\vCenterServer\cfg\vsphere-client\vc-packages\vsphere-client-serenity\
Mac OS local development environment	/var/lib/vmware/vsphere-client/vsphere-client/vc-packages/vsphere-client-serenity/

- To avoid performance issues, 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 11-1. 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`
vCenter Server Appliance 6.5 installation vSphere Client	`/var/log/vmware/vsphere-ui/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 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.