Developing Remote Plug-ins with the vSphere Client SDK

Update 3
VMware vSphere 8.0
vSphere Client SDK 8.0

vmware by Broadcom

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

https://docs.vmware.com/

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

Contents

About This Book

Developing Remote Plug-ins with the vSphere Client SDK provides information about developing and deploying HTML-5 extensions to the VMware vSphere® Client user interface.

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

The vSphere Client SDK

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

VMware once offered local plug-ins for the vSphere Client SDK, but these are replaced by remote plug-ins. Remote plug-ins are compatible with VMware cloud, easier to upgrade, more secure, perform better, and support a wide variety of configurations.

Intended Audience

This information is intended for anyone who wants to extend the vSphere Client with custom functionality. Users typically are software developers who use HTML and JavaScript to create graphical user interface components that work with VMware platforms.

Revision History

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

The table below provides an update history of this book.

Revision	Description
18 JUN 2024	Release with vSphere 8.0 U3: ■ Dynamic actions can be hidden or disabled. ■ Add vCenter version parameters and per-vCenter extensions. ■ Improvements to extension registration tool. ■ Fix functionality of multi-manifest and describe its backward compatibility. ■ New feature to define the size of summary view cards.
21 SEP 2023	Release with vSphere 8.0 U2: ■ Users can monitor progress of solution installation in the Tasks console. ■ No longer necessary to have `plugin.json` for multi-manifest support. ■ Additional properties for vCenter Servers injected into OVF. ■ Developers without `vim25.jar` must obtain it from vSphere Management SDK. ■ Plug-in servers can be registered with certificate in addition to or instead of thumbprint.
18 APR 2023	Release with vSphere 8.0 U1: ■ New type "separator" can define dividers between menu actions. ■ Ability to define multiple summary cards (three later increased to five). ■ New property `requiredVcVersion` available with integrated solution installer.
29 MAR 2023	Update: ■ Added `solutioninstall` API to integrated solution installer chapter.
14 JAN 2023	Corrections: ■ Corrected specification for app.ClientViewNavigationOptions. ■ Added vCenter Server thumbprint to registration example. ■ Removed obsolete material in Best Practices and Troubleshooting chapters. ■ Corrected statement about resizing Summary views.

Revision	Description
11 OCT 2022	vSphere 8.0 release. ■ Change occurrences of 'disable' to 'deactivate'. ■ Update documentation to reflect the change to Clarity cards in place of portlets. ■ Add advice to label the primary server with type MANIFEST_SERVER in the registration record. ■ Add a chapter about the integrated solution installer. See Chapter 9 Integrated Solution Installer for the vSphere Client. ■ Add material about dynamic Summary views. See Chapter 6 Dynamic Extensions for Remote Plug-ins. ■ Add material about using auxiliary servers for dynamic filter queries. See Chapter 6 Dynamic Extensions for Remote Plug-ins. ■ Refresh screenshots in Chapter 5 Choosing Extension Points for vSphere Client Plug-ins and Viewing the vSphere Client Remote Plug-in Sample.
14 OCT 2021	Streamlined description of remote plug-in sample directory structure.
05 OCT 2021	vSphere Client SDK 7.0 Update 3 release. ■ Expanded material on server back end authentication to explain more flexible token exchange and access feature. ■ Improved diagram of authentication communicatons. ■ Added MacOS examples to Remote Sample Starter. ■ Updated information about SHA-1 deprecation. ■ Removed troubleshooting topic about OSGi (inapplicable for remote plug-ins). ■ Removed obsolete reference to `updateOnBrowserRefresh`. ■ Noted Administrator role is required for Redeploy feature. ■ Removed obsolete references to external Platform Services Controller.
09 MAR 2021	vSphere Client SDK 7.0 Update 2 release. ■ Expand information about auxiliary plug-in servers. ■ Expand material about plug-in instance handling. ■ Corrected final step in using `cloneSession()` method. ■ Improvements to the chapter about using the Remote Plug-in Sample. Added Remote Plug-in Sample Starter instructions. ■ Added screen shots to illustrate UI context for extension points.
06 OCT 2020	vSphere Client SDK 7.0 Update 1 release. ■ Dynamic extension support. ■ Multiple targets for plug-in actions. ■ Add plug-in server logging configuration to server startup command. ■ Add production build option to SDK sample. ■ Add chapter for best practices. ■ Multi-manifest capability to support different versions. ■ Expanded material concerning extension points. ■ At VMware, we value inclusion. To foster this principle within our customer, partner, and internal community, we are replacing some of the terminology in our content. We have updated this guide to remove instances of non-inclusive language.
04 MAY 2020	Cosmetic improvements.

Revision	Description
02 APR 2020	vSphere Client SDK 7.0 release. ■ Extended and updated JavaScript API. ■ SHA-256 thumbprint support. ■ Support for auxiliary plug-in servers. ■ Support for theme-dependent icons. ■ Improved discovery of new plug-ins or updates. ■ Plug-in compatibility support. ■ Redeploy button in development mode.
20 AUG 2019	vSphere Client SDK 6.7U3 release. ■ Corrected MOB URL in Troubleshooting chapter. ■ Added Additional Resources chapter.
20 JUN 2019	Minor corrections to Troubleshooting chapter.
30 MAY 2019	Minor updates and corrections. Added Troubleshooting chapter.
11 APR 2019	Changes for vSphere Client SDK 6.7U2 release. ■ Replaced Virgo server with Tomcat server. ■ Updated JavaScript API to handle UI themes. ■ Added chapter about UI themes. ■ Added information about plug-in caching. ■ Added information about plug-in handling in linked mode. ■ Expanded and improved chapter about running remote plug-in sample. ■ Improved information about plug-in deployment.
12 FEB 2019	Minor updates and corrections. Expanded material about Deployment.
16 OCT 2018	Initial release.

Using the vSphere Client Remote Plug-in Samples

1

For an introduction to the vSphere Client Remote Plug-in SDK, you can install and run the remote plug-in sample and the remote plug-in sample starter. The samples illustrate several key features that you can adapt to develop your own plug-ins.

This chapter assumes that you have access to a vCenter Server instance and a development machine where you build and run the sample plug-in server.

Read the following topics next:

Try the Remote Plug-in Sample Starter

Build and run the remote plug-in sample starter for a quick introduction to the process of deploying and viewing a remote plug-in for the vSphere Client.

The plug-in sample starter demonstrates a very simple plug-in that serves only static HTML content. This is the easiest way to be introduced to the remote plug-in environment, including only a few components that extend the vSphere Client user interface with a full-screen view available for plug-in content.

What To Do First

Check the dependencies on your development machine:

- Java Development Kit 8
- Maven 3

Install the vSphere Client SDK:

1 Download the SDK from https://code.vmware.com/sdk/client

2 In a shell window, unzip the SDK file:

PowerShell:

```
PS C:\> Get-Item vsphere-client-sdk-*zip |
 Foreach-Object { Expand-Archive -Path $_ -DestinationPath C:\ }
```

MacOS:

```
Downloads % unzip vsphere-client-sdk-*zip -d ~
```

Generate a Plug-in Server Certificate

1 Generate an SSL certificate and a JKS key store for the remote plug-in server using Java's `keytool` certificate management utility.

PowerShell example:

```
PS C:\> keytool -genkeypair -alias plugin-server -keyalg RSA -keysize 4096 `
-storetype JKS -keystore keystore.jks -storepass "remote.plugin" -keypass "remote.plugin" `
-dname "CN=mydevbox"
```

MacOS example:

```
Downloads % keytool -genkeypair -alias plugin-server -keyalg RSA -keysize 4096 \
-storetype JKS -keystore keystore.jks -storepass "remote.plugin" -keypass "remote.plugin" \
-dname "CN=mydevbox"
```

Note Do not change the values of arguments to the `keytool` utility, except the value *mydevbox* in the `-dname` "CN=mydevbox" argument. The value *mydevbox* must be replaced by the IP address or the DNS name of the machine on which the remote plug-in server will run. The value is significant because vCenter Server will check whether the Common Name (CN) field value of the remote plug-in server certificate matches the host through which remote plug-in resources are served.

2 Replace the default *keystore* in the Remote Plug-in Sample Starter code with the newly generated one:

PowerShell example:

```
PS C:\> Move-Item -Path "keystore.jks" -Destination `
 "html-client-sdk\samples\remote-plugin-sample-starter\src\main\resources\keystore.jks" `
 -Force
```

MacOS example:

```
Downloads % mv -f keystore.jks \
 ~/html-client-sdk/samples/remote-plugin-sample-starter/src/main/resources/keystore.jks
```

Build the Remote Plug-in Sample Starter

1 Navigate to the sample starter directory:

PowerShell example:

```
PS C:\> cd html-client-sdk/samples/remote-plugin-sample-starter
```

MacOS example:

```
Downloads % cd ~/html-client-sdk/samples/remote-plugin-sample-starter
```

2 Issue the Maven build command:

PowerShell example:

```
PS C:\html-client-sdk\samples\remote-plugin-sample-starter> mvn clean install
```

MacOS example:

```
remote-plugin-sample-starter % mvn clean install
```

Start the Remote Plug-in Sample Starter

1 Navigate to the `target` directory:

PowerShell example:

```
PS C:\html-client-sdk\samples\remote-plugin-sample-starter> cd target
```

MacOS example:

```
remote-plugin-sample-starter % cd target
```

2 Issue the Java command to run the plug-in server:

PowerShell example:

```
PS C:\html-client-sdk\samples\remote-plugin-sample-starter\target> `
java -jar (Get-ChildItem remote-plugin*.jar -Name | Select-Object -first 1)
```

MacOS example:

```
target % java -jar remote-plugin-sample-starter-*.jar
```

3 Use a browser to verify that the remote plug-in server can supply its manifest. Enter the following URL:

```
https://localhost:8443/sample-ui/plugin.json
```

Leave the browser window open for the next step.

Save the Certificate or Fingerprint of the Plug-in Server

1 Find the security certificate and/or fingerprint of the plug-in server.

In Chrome browser for example, click the padlock icon next to the URL at the top of the browser window. Then click **Connection is secure** > **Certificate is valid**. You can copy the SHA-256 Fingerprint or Thumbprint (depending on the browser) from there. For the certificate, click **Details** > **Export**.

2 Paste the SHA-256 fingerprint into a file, or save the certificate as a PEM file with `.cer` suffix.

vCenter Server expects the fingerprint (thumbprint) to contain colon-separated character pairs. As of vSphere 8.0 U3, the registration script changes unseparated or space-separated character pairs into the expected format.

For PEM, select base64-encoded ASCII, single certificate. In the plug-in server certificate, check that the server host name is specified according to subjectAltName extension (RFC 2818, Section 3.1 Server Identity).

Register the Remote Plug-in Sample Starter

1 Navigate to the directory containing the registration script:

PowerShell example:

```
PS C:\html-client-sdk\samples\remote-plugin-sample-starter\target> `
cd 'C:\html-client-sdk\tools\vCenter plugin registration\prebuilt'
```

MacOS example:

```
target % cd ~/html-client-sdk/tools/vCenter\ plugin\ registration/prebuilt
```

2 Run the registration script for the plug-in. Use the name of your vCenter Server instance in the URL and use valid credentials to authenticate. Enter the password for `administrator@vsphere.local` when prompted.

PowerShell example:

```
PS C:\html-client-sdk\tools\vCenter plugin registration\prebuilt> `
 .\extension-registration.bat `
 -action registerPlugin -remote `
 -url https://myvcenter.example.com/sdk `
 -username administrator@vsphere.local `
 -key com.vmware.sample.remote.starter -version 1.0.0 `
 -pluginUrl https://mydevbox:8443/sample-ui/plugin.json `
 -serverThumbprint 19:FD:2B:0E:62:5E:0E:10:FF:24:34:7A:81:F1:D5:33:`
19:A7:22:A0:DA:33:27:07:90:0F:8E:8D:72:F1:BD:F1 `
 -serverCertificateFile C:\plugin-server.cer `
 -c 'Example, Inc.' -n 'Remote Plug-in' -s 'This is a sample plug-in'
```

MacOS example:

```
prebuilt % ./extension-registration.sh \
 -action registerPlugin -remote \
 -url https://myvcenter.example.com/sdk \
 -username administrator@vsphere.local \
 -key com.vmware.sample.remote.starter -version 1.0.0 \
 -pluginUrl https://mydevbox:8443/sample-ui/plugin.json \
 -serverThumbprint 19:FD:2B:0E:62:5E:0E:10:FF:24:34:7A:81:F1:D5:33:\
19:A7:22:A0:DA:33:27:07:90:0F:8E:8D:72:F1:BD:F1 \
 -serverCertificateFile /tmp/plugin-server.cer \
 -c 'Example, Inc.' -n 'Remote Plug-in' -s 'This is a sample plug-in' -insecure
```

When registering the plug-in, it's highly recommended to provide the SSL certificate of the plug-in server in addition to the thumbprint of the SSL certificate of the plug-in server. Performing a full SSL certificate check during SSL handshake is more secure than performing an SSL certificate thumbprint check. Furthermore, in future vSphere releases, support for SSL certificate thumbprints will be dropped and full SSL certificates will be required.

View the Plug-in User Interface

1. Connect a web browser to the vCenter Server instance and log in to the vSphere Client. Example URL:

   https://myvcenter.example.com/ui/

2. Navigate to the Home screen (for example, use **ctrl-alt-home**) or the Shortcut screen.
3. In the object navigator pane on the left of the home screen, click on the remote plug-in sample starter.

   The vSphere Client displays a plug-in welcome screen.

Remote Plug-in Sample UI and Server

The remote plug-in sample demonstrates a secure, efficient, remote plug-in that was developed according to recommended practices. The sample supports a Global View extension to the

vSphere Client. The user interface component creates a modal dialog, views that extend vSphere inventory objects, and examples of using a card to extend a Summary view.

The user interface code also shows how to:

- Initialize the Client API.
- Retrieve session authentication information and pass it to the plug-in server.
- Perform a data retrieval request to the plug-in server.
- Define a context action for a VirtualMachine object.

The plug-in server code shows how to:

- Respond to a data retrieval or modification request.
- Clone and cache a user session with a vCenter Server.

The remote plug-in sample does the following actions:

- Creates a global view.
- Opens a modal dialog.
- Authenticates to vCenter Server.
- Performs a data retrieval call.
- Creates a relation between a Chassis object and a HostSystem object.
- Creates several views that extend context object views.
- Defines an action on a VirtualMachine context object.

Components in the Remote Plug-in Sample

The remote plug-in sample in the vSphere Client SDK shows how to design and implement, deploy and register a remote plug-in. The sample is functionally simple, to focus on displaying the infrastructure rather than the business logic.

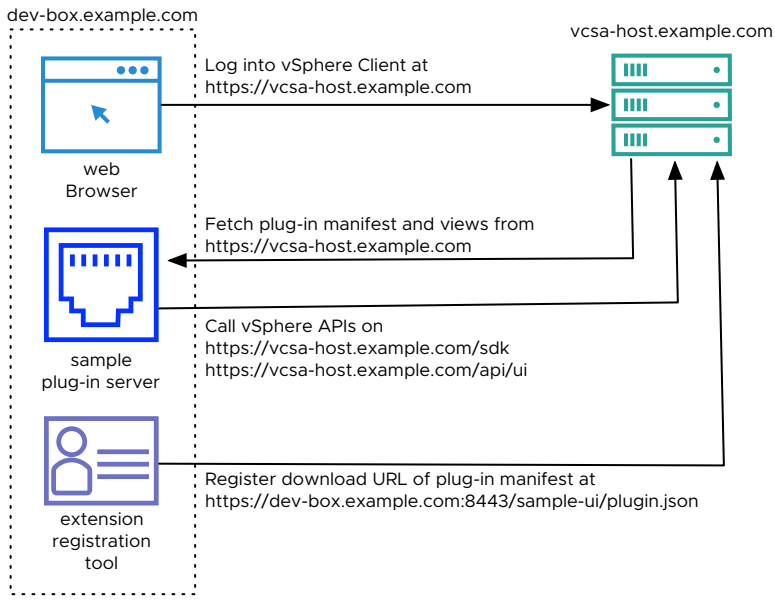
The sample remote plug-in package contains several components:

- The user interface is written in JavaScript and Angular, using Clarity design components to maintain compatibility with the vSphere Client user interface.
- The sample plug-in server is written in Java, but Java is not a requirement. The server includes the following:
  - In-memory data storage for fictitious Chassis objects.
  - Controller logic to handle user interface requests for Chassis objects and vSphere HostSystem objects.
  - Service interfaces for operations on both kinds of objects.

  - A library layer to interface to vCenter Server, including logic to handle delegated authentication.
- The `plugin.json` file specifies the vSphere Client extension points that the plug-in extends.
- The `spring-context.xml` file contains the Spring bean definitions for the plug-in server.
- The `pom.xml` file specifies how Maven will install dependencies and build the plug-in deliverable.
- The `application.properties` file specifies properties that the Spring application server uses to deploy the plug-in server.

The following diagram illustrates the basic architecture of the remote plug-in sample, when installed in a simple vSphere environment.

Figure 1-1. Remote Plug-in Sample Environment



Build the vSphere Client Remote Plug-in Sample

You build the remote plug-in sample with a Maven build directed by the sample's `pom.xml` file.

After you download the vSphere Client SDK, you must build the remote plug-in sample before you can run it.

1 Generate an SSL certificate and a JKS key store for the remote plug-in server using Java's `keytool` certificate management utility:

PowerShell example:

```
PS C:\> keytool -genkeypair -alias plugin-server -keyalg RSA -keysize 4096 `
-storetype JKS -keystore keystore.jks -storepass "remote.plugin" -keypass "remote.plugin" `
-dname "CN=mydevbox"
```

MacOS example:

```
Downloads % keytool -genkeypair -alias plugin-server -keyalg RSA -keysize 4096 \
-storetype JKS -keystore keystore.jks -storepass "remote.plugin" -keypass "remote.plugin" \
-dname "CN=mydevbox"
```

Note Do not change the values of the arguments of `keytool` utility except the value *mydevbox* in the `-dname` "CN=mydevbox" argument. The value *mydevbox* must be replaced with the IP address or the DNS name of the machine on which the remote plug-in server will run. The value is significant because vCenter Server will check whether the Common Name (CN) field value of the remote plug-in server certificate matches the host through which remote plug-in resources are served.

2 Replace the default key store in the Remote Plug-in Sample code with the newly generated one.

PowerShell example:

```
PS C:\> Move-Item -Path "keystore.jks" -Destination \
"html-client-sdk\samples\remote-plugin-sample\src\main\resources\keystore.jks" -Force
```

MacOs example:

```
Downloads % mv -f keystore.jks \
 ~/html-client-sdk/samples/remote-plugin-sample/src/main/resources/keystore.jks
```

3 Navigate to the sample directory:

PowerShell example:

```
PS C:\> cd html-client-sdk/samples/remote-plugin-sample
```

MacOS example:

```
Downloads % cd ~/html-client-sdk/samples/remote-plugin-sample
```

4 Install the Web Services API library into the local Maven repository:

```
mvn validate
```

5 Build the sample:

```
mvn clean install
```

For production builds, substitute `mvn clean install -Dproduction.build=true` in step 5.

Prerequisites

Before building the remote plug-in sample, you must have the following:

- You need Java 8 to compile the code for this sample.
- You need Maven 3 to build the plug-in package.
- Download the SDK and unzip it. See Remote Plug-in Sample Directory Structure for information about the structure of the SDK archive.

Results

These steps install the Web Services API library into the local Maven repository, download and build the sample executable, download the Clarity design system, and build the JAR file that contains the sample components. The sample is a Spring Boot Application that will start an embedded Tomcat server when you run the sample in a command shell.

Choose a production build for the client code to run more efficiently. Choose a development build for a better debugging experience.

What to do next

After you build the remote plug-in sample, you need to locate authentication information, start the plug-in server, and register the plug-in with vCenter Server.

Find the SSL Thumbprint and GUID of vCenter Server

Before you start the remote plug-in server, you need to find the certificate thumbprint (fingerprint) and GUID of the vCenter Server where you want to register the plug-in.

Prerequisites

vCenter Server must be running while you do this procedure.

Procedure

1. Connect a browser to the vCenter Server.

   The URL for vCenter Server looks similar to this: `https://vcenter-1.example.com`

   The browser displays a launch screen, with a small padlock icon in the address field.
2. Click **LAUNCH VSPHERE CLIENT (HTML5)** and log in to the vSphere Client.

   The browser displays the default Hosts and Clusters view.
3. If you connected to a vCenter Server instance in an extended linked mode environment, you must select the chosen vCenter Server instance in the navigation pane on the left of the vSphere Client screen.

   The URL in the browser address box contains an embedded managed object reference, similar to the following:

`Folder:group-d1:56d373bd-4163-44f9-a872-9adabb008ca9`. This is an extended managed object reference that ends with the GUID of the vCenter Server instance. The GUID is a string of 32 hexadecimal digits, organized in groups of 4, 8, or 12 digits, separated by hyphens.

4 Copy the 32 hexadecimal digits of the GUID, along with the inset hyphens, and save this into a shell variable or a text file.

   You will use the GUID when you start the plug-in server.

5 Click the padlock icon in the browser address field to access a certificate information window.

   The browser displays a brief summary of browser properties.

6 Click **Details** to display more certificate information.

   The browser displays full details of the vCenter Server certificate.

7 Scroll through the certificate details to find the SHA-256 fingerprint.

   The SHA-256 fingerprint is a string of 64 hexadecimal digits, usually in pairs separated by spaces or other non-alphanumeric delimiters.

8 Select the fingerprint string and copy it to a text file.

9 Edit the text file to remove all spaces or other delimiters from the fingerprint string.

   You now have a string of 64 contiguous hexadecimal digits. This is the *thumbprint* of the vCenter Server instance.

What to do next

The thumbprint and GUID of the vCenter Server instance are needed to start the remote plug-in server. After starting you will determine the plug-in server's certificate thumbprint.

Start the Remote Plug-in Server

The remote plug-in sample has an embedded Spring Boot application server. You can start the server by using a Java command with arguments that configure the server to communicate with a vCenter Server instance. This is the vCenter Server instance with which you will register the plug-in.

Prerequisites

- Before you can run the remote plug-in sample, you must have built the SDK sample code.
- Before you run the remote plug-in sample, choose the vCenter Server where you will register the plug-in.
- Collect the following information about the vCenter Server instance:
  - Fully qualified domain name
  - Port number for Web Services API access
  - Certificate thumbprint

Procedure

1 In a shell window, change to the root directory of the remote plug-in sample.

```
cd sdk/samples/remote-plugin-sample
```

2 Run the JAR file in the `target` directory.

The command to run the plug-in JAR file requires several arguments, including the thumbprint, GUID, DNS name, and HTTPS port number of the vCenter Server instance. You can also specify `--logging.path`, which creates a subdirectory (if it does not already exist) and stores server log files in the subdirectory. Use a command similar to the following example, but substitute the details that pertain to your vCenter Server:

```
java -jar target/remote-plugin-sample-7.0.1.00000.jar \
  --logging.path=logdir \
  --vcenter.guid=223b94f2-af15-4613-5d1a-a278b19abc09 \
  --vcenter.thumbprint=274172e07a754b9811a4fb5fc45384a79a5c258d13fa1667185016f28685fc54 \
  --vcenter.fqdn=vcenter-1.example.com --vcenter.port=443
```

Results

The plug-in application server runs. It might take a few minutes to initialize, and the console displays a number of output lines. When the server is ready, the console displays two lines saying `Tomcat started` and `Started SpringbootApplication`.

Example:

```
2021-02-25 04:36:49.442  INFO 76 --- [           main]
 o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8443
 (https) with context path '/sample-ui'
2021-02-25 04:36:49.442  INFO 76 --- [           main]
 c.v.sample.remote.SpringBootApplication : Started SpringBootApplication in
 34.752 seconds (JVM running for 36.127)
```

What to do next

If you have not yet registered the plug-in with a vCenter Server, find the plug-in server thumbprint and use it to register the plug-in. Registration enables the plug-in to render global views and object-specific views for objects managed by the vCenter Server instance.

Get Thumbprint or Certificate of Remote Plug-in Server

The remote plug-in sample has an embedded application server with a self-signed certificate that is used for encrypted communications. The certificate and its thumbprint are stored in a Java keystore file.

To register a remote plug-in with vCenter Server, you need to determine the thumbprint of the plug-in application server, or save its security certificate, which you then use in arguments to the registration command.

Prerequisites

- Start the server.
- Find the server's port number. The default port number is 8443. You can configure a different port number in the `application.properties` file.

Procedure

1. Connect a browser to the application server, for example using the URL of the plug-in manifest.

   The default URL for the manifest file is `https://localhost:8443/sample-ui/plugin.json`.

2. Examine the certificate presented by the application server.

   This step depends on the browser. In Firefox for example, you click the padlock icon next to the URL, then select **Connection secure** > **More information** > **View Certificate**. The thumbprint is the field labelled **SHA-256** and the certificate can be downloaded by the **PEM (cert)** link.

   Note A SHA-1 fingerprint is also supported, but SHA-1 is deprecated in favor of SHA-256.

3. Save the thumbprint in a text file, or save the PEM certificate in a file with `.cer` suffix.

   vCenter Server expects the thumbprint (fingerprint) to contain colon-separated character pairs. As of vSphere 8.0 U3, the registration script changes unseparated or space-separated character pairs into the expected format.

What to do next

Use the application server thumbprint or PEM certificate when you register the plug-in with vCenter Server.

Register the vSphere Client Remote Plug-in Sample

Before you can view the remote plug-in sample in the vSphere Client, you must register it with a vCenter Server instance to which you want to connect. The vSphere Client SDK contains a vCenter Server plug-in registration tool that registers a plug-in with a vCenter Server ExtensionManager.

You have installed the SDK, built the sample, and are ready to run the remote plug-in.

Prerequisites

- Start the server.
- If needed, start vCenter Server.
- If needed, change permissions on the plug-in registration tool to allow execute access.

- Collect the following information to prepare the registration command:
  - The host name or IP address of the machine where you are running the sample plug-in server. This address must be accessible to the vCenter Server instance so that it can download the plug-in manifest file.
  - The port number where the plug-in server receives HTTPS requests. The sample serves port 8443 by default. The port must be open on the firewall of your development machine and accessible to the vCenter Server.
  - The host name or IP address of the vCenter Server where you want to register the remote plug-in sample.
  - The username and password of a vSphere user who has permission to access the ExtensionManager on the vCenter Server where you want to register the remote plug-in sample. For example, `administrator@vsphere.local` normally has the necessary `Extension.Register` permission.
  - The SHA-256 thumbprint or SSL certificate of the plug-in server, so that vCenter Server can retrieve the plug-in manifest file. With a certificate, check that the host name in the plug-in server certificate is specified according to subjectAltName extension (RFC 2818, Section 3.1 Server Identity).

    Note vCenter Server expects the thumbprint (fingerprint) to contain colon-separated character pairs. As of vSphere 8.0 U3, the registration script changes unseparated or space-separated character pairs into the expected format.
  - The version number of the remote plug-in sample.
  - The key of the remote plug-in sample, defined in the plug-in manifest, `plugin.name` by default.
  - Path from plug-in web server root to the plug-in manifest file, `/sample-ui/plugin.json` by default.
  - Location of JSON files for `taskList`, `faultList`, `privilegeList`, and `resourceList`.

For more information about the registration tool, see vSphere Client Plug-in Registration Tool.

Procedure

1 In a command terminal or shell, change directory to the `prebuilt` folder, which contains an `extension-registration.bat` script for Windows, and an `extension-registration.sh` script for Unix or Linux.

```
cd html-client-sdk/tools/*plugin*/prebuilt
```

2 Run the extension-registration script appropriate for your operating system, specifying the prerequisite parameters.

For a Unix or Linux shell, use this syntax:

```
path2json=../../../samples/remote-plugin-sample/registration/resources
./extension-registration.sh  -action registerPlugin  -remote \
-url https://myvcenter/sdk \
-username administrator@vsphere.local  -password 'mysecret' \
-key com.vmware.sample.remote  -version 1.0.0 \
-pluginUrl https://mydevbox:8443/sample-ui/plugin.json \
-serverThumbprint 19:FD:2B:0E:62:5E:0E:10:FF:24:34:7A:81:F1:D5:33:\
19:A7:22:A0:DA:33:27:07:90:0F:8E:8D:72:F1:BD:F1 \
-serverCertificateFile /tmp/plugin-server.cer \
-c 'Example, Inc.'  -n 'Remote Plug-in'  -s 'This is a sample plug-in' \
--faultList $path2json/faultList.json --privilegeList $path2json/privilegeList.json \
--resourceList $path2json/resourceList.json --taskList $path2json/taskList.json
```

For a DOS command shell, use this syntax:

```
set path2json=..\..\..\samples\remote-plugin-sample\registration\resources
.\extension-registration.bat  -action registerPlugin -remote ^
-url https://myvcenter/sdk ^
-username administrator@vsphere.local  -password "mysecret" ^
-key com.vmware.sample.remote  -version 1.0.0 ^
-pluginUrl https://mydevbox:8443/sample-ui/plugin.json ^
-serverThumbprint 19:FD:2B:0E:62:5E:0E:10:FF:24:34:7A:81:F1:D5:33:^
19:A7:22:A0:DA:33:27:07:90:0F:8E:8D:72:F1:BD:F1 ^
-serverCertificateFile C:\plugin-server.cer ^
-c "Example, Inc."  -n "Remote Plug-in"  -s "This is a sample plug-in" ^
--faultList %path2json%\faultList.json --privilegeList %path2json%\privilegeList.json ^
--resourceList %path2json%\resourceList.json --taskList %path2json%\taskList.json
```

For a PowerShell prompt, use this syntax:

```
Set-Variable path2json ..\..\..\samples\remote-plugin-sample\registration\resources
.\extension-registration.bat  -action registerPlugin  -remote `
-url https://myvcenter/sdk `
-username administrator@vsphere.local  -password 'mysecret' `
-key com.vmware.sample.remote  -version 1.0.0 `
-pluginUrl https://mydevbox:8443/sample-ui/plugin.json `
-serverThumbprint 19:FD:2B:0E:62:5E:0E:10:FF:24:34:7A:81:F1:D5:33:`
19:A7:22:A0:DA:33:27:07:90:0F:8E:8D:72:F1:BD:F1 `
-serverCertificateFile C:\plugin-server.cer `
-c 'Example, Inc.'  -n 'Remote Plug-in'  -s 'This is a sample plug-in' `
--faultList $path2json\faultList.json --privilegeList $path2json\privilegeList.json `
--resourceList $path2json\resourceList.json --taskList $path2json\taskList.json
```

If a password contains special characters, use appropriate escape sequences for your shell.

Note When registering the plug-in, it's highly recommended to provide the SSL certificate of the plug-in server in addition to the thumbprint of the SSL certificate of the plug-in server. Performing a full SSL certificate check during SSL handshake is more secure than performing an SSL certificate thumbprint check. Furthermore, in future vSphere releases, support for SSL certificate thumbprints will be dropped and full SSL certificates will be required.

Results

The registration script displays a message that the plug-in is "successfully registered in vCenter."

An `Extension` record is added in the ExtensionManager of the vCenter Server instance.

What to do next

In a web browser, connect to the vCenter Server URL and verify that the remote plug-in displays a Global View.

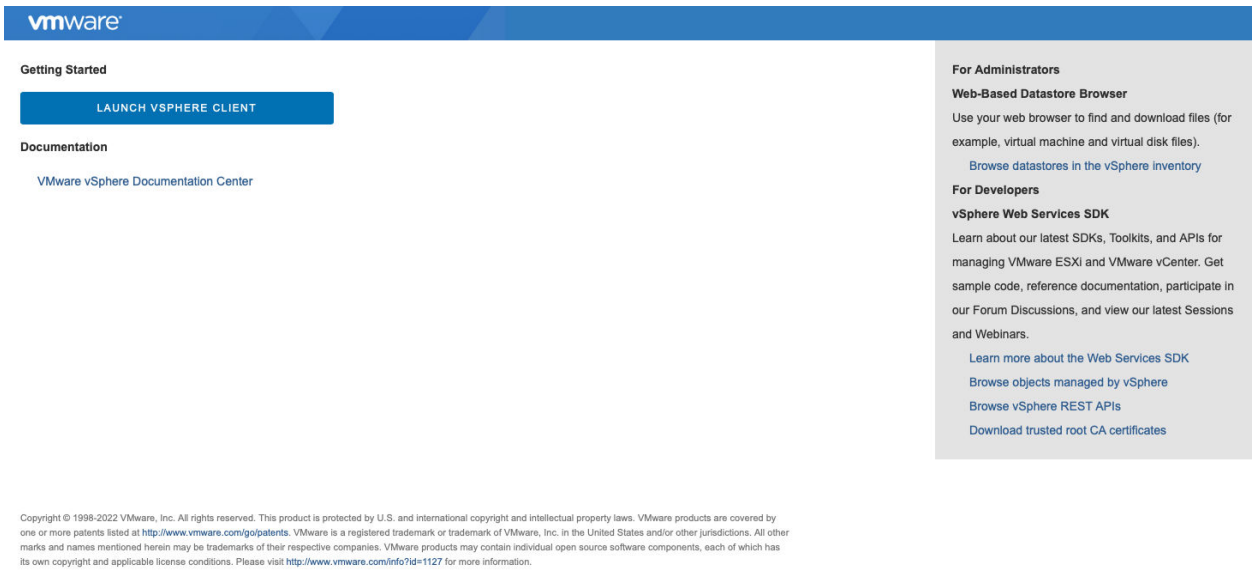
Viewing the vSphere Client Remote Plug-in Sample

After you build and register the remote plug-in sample, you can view it in the vSphere Client.

To view the remote plug-in sample, you must run the plug-in server and register it with the vCenter Server instance to which you connect your browser and run the vSphere Client.

To view the plug-in user interface, open a browser window and connect to the vCenter Server instance where you registered the plug-in. Use the fully qualified domain name of the vCenter Server to match the server certificate. From the welcome screen you can launch the vSphere Client.

Figure 1-2. vCenter Server Welcome Screen



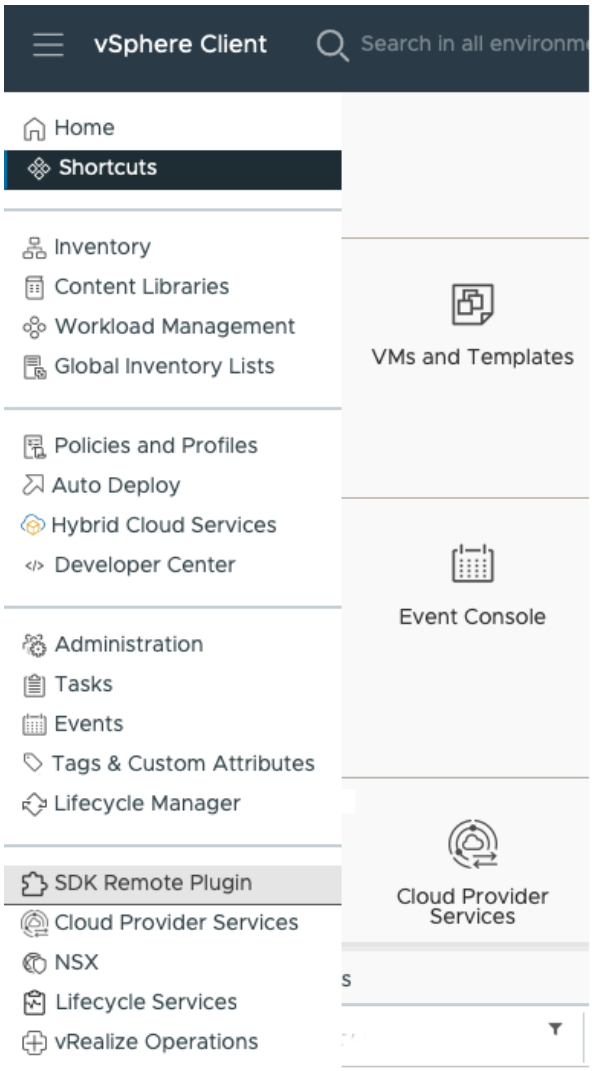
After you select **LAUNCH VSPHERE CLIENT** and authenticate with vCenter Server, the vSphere Client displays the home page with an **SDK Remote Plugin** link in the object navigator pane on the left. The link leads to the remote plug-in global view.

The plug-in global view includes the following subviews:

- A welcome page is the default page for the global view.
- A settings page allows you to change the number of items displayed in the Chassis List page.
- A Chassis List page displays summary information about the Chassis objects currently in the store. The Chassis store is initialized with several random Chassis objects for display. On the Chassis List page you can do the following actions:
  - Edit the Chassis object properties in a modal dialog.
  - Display related Host objects.
  - Display the Monitor subview or the Configure subview.

The code that supports these actions demonstrates how to use a modal dialog, how to create, delete, and update Chassis objects, and how to make calls to the plug-in server and the vCenter Server. The following illustrations show some of the features of the global view.

Figure 1-3. Selecting the Sample Plug-in Global View in the Navigation Pane



The Global View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"global": {
   "view": {
      "navigationId": "entrypoint",
      "uri": "index.html#/entry-point",
      "navigationVisible": false
   }
}
```

The Global View content is specified in the file `entry.point.component.html`.

Figure 1-4. Sample Plug-in Global View Welcome Page

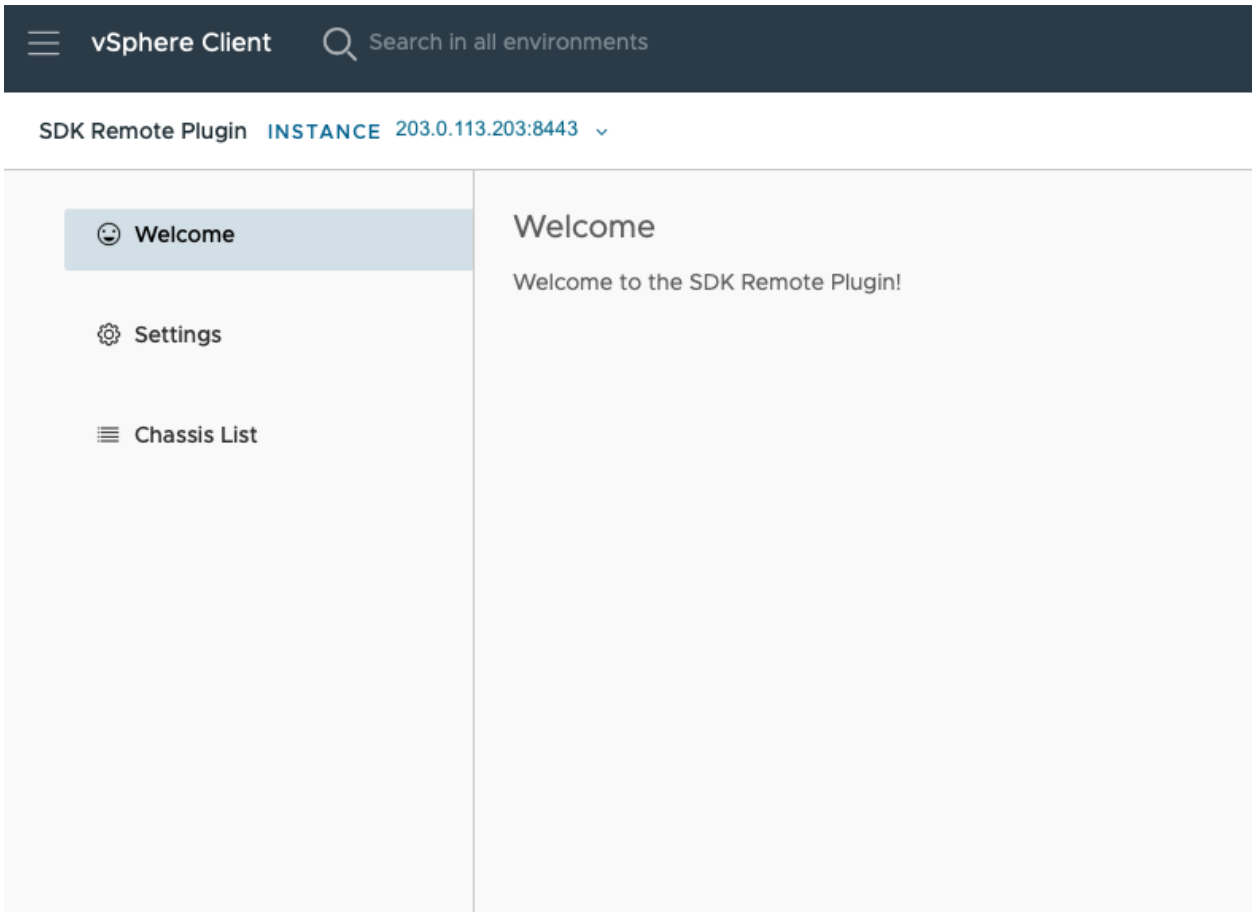


Figure 1-5. Sample Plug-in Global View Chassis List Page

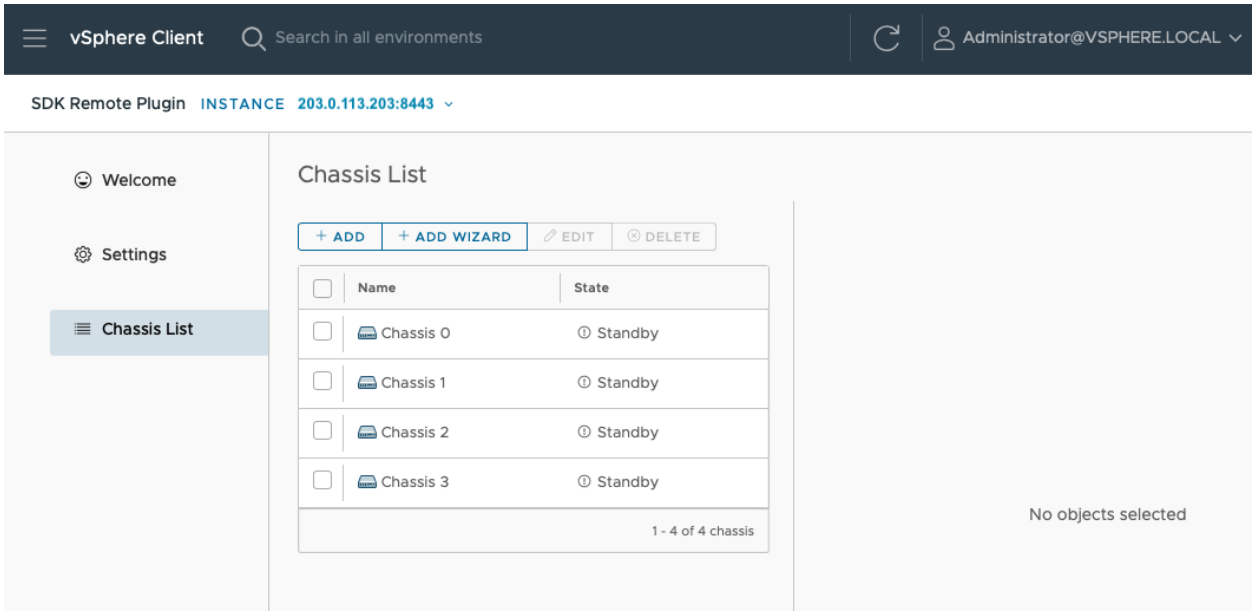


Figure 1-6. Sample Plug-in Edit Chassis Dialog

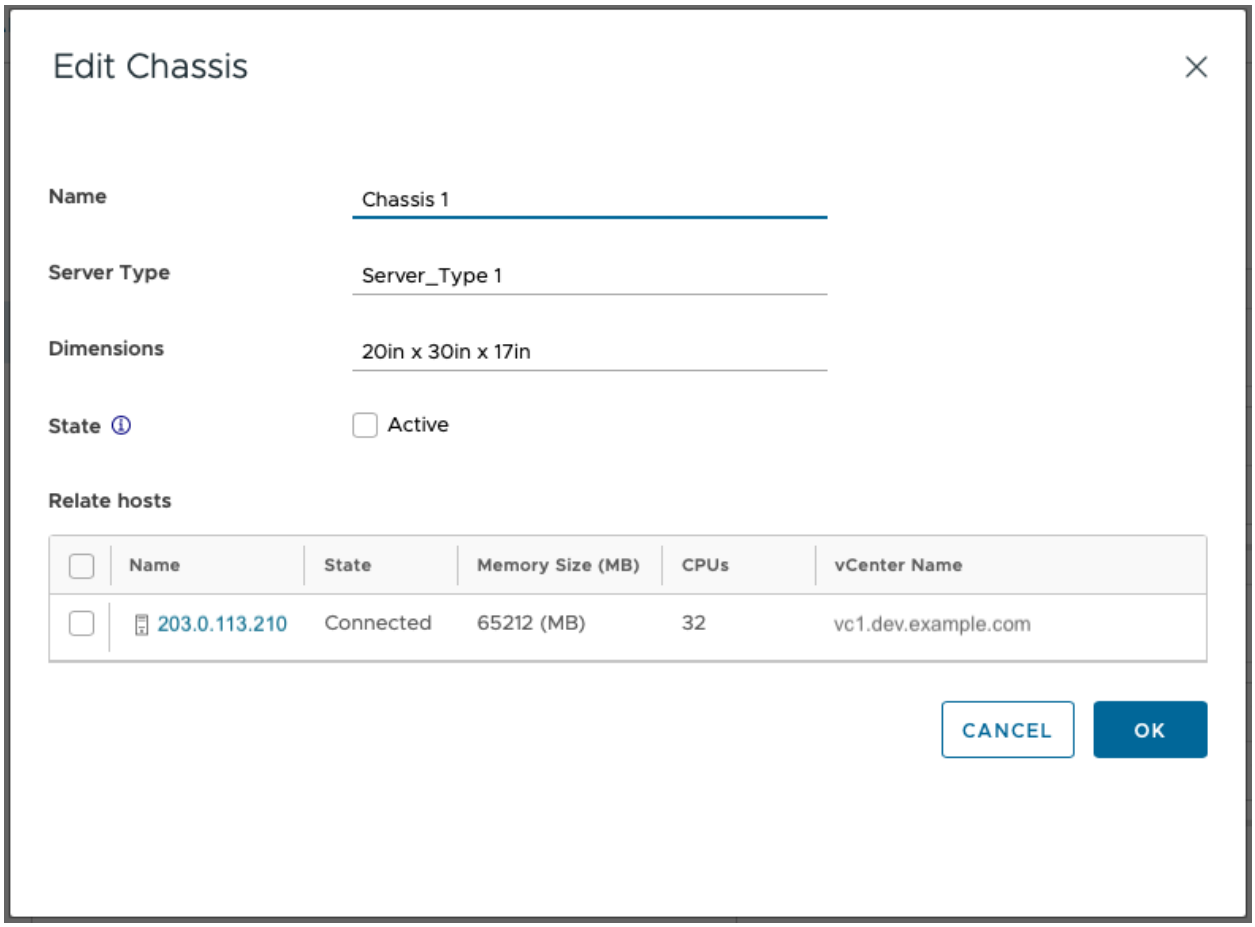
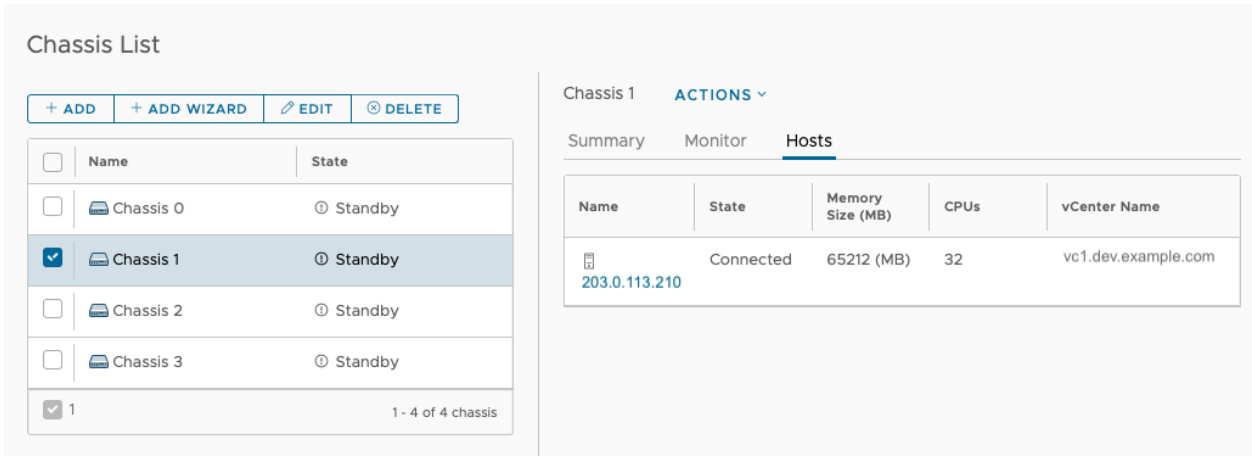
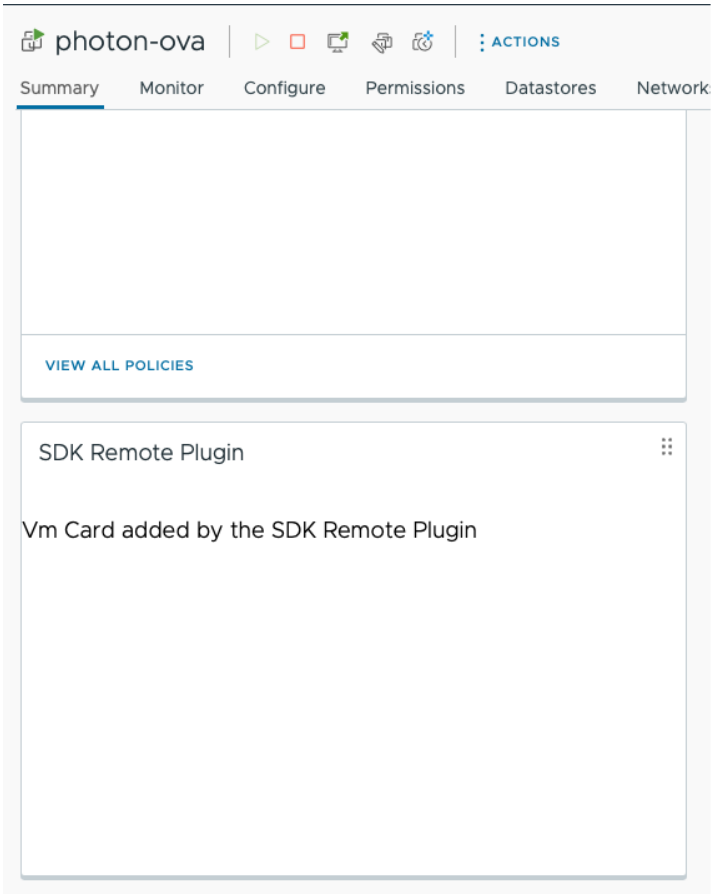


Figure 1-7. Sample Plug-in Chassis with Related Host



In addition to the global view and its subviews, the remote plug-in sample provides views that show how to extend the vSphere VirtualMachine object. It also shows the use of a card within the Summary view.

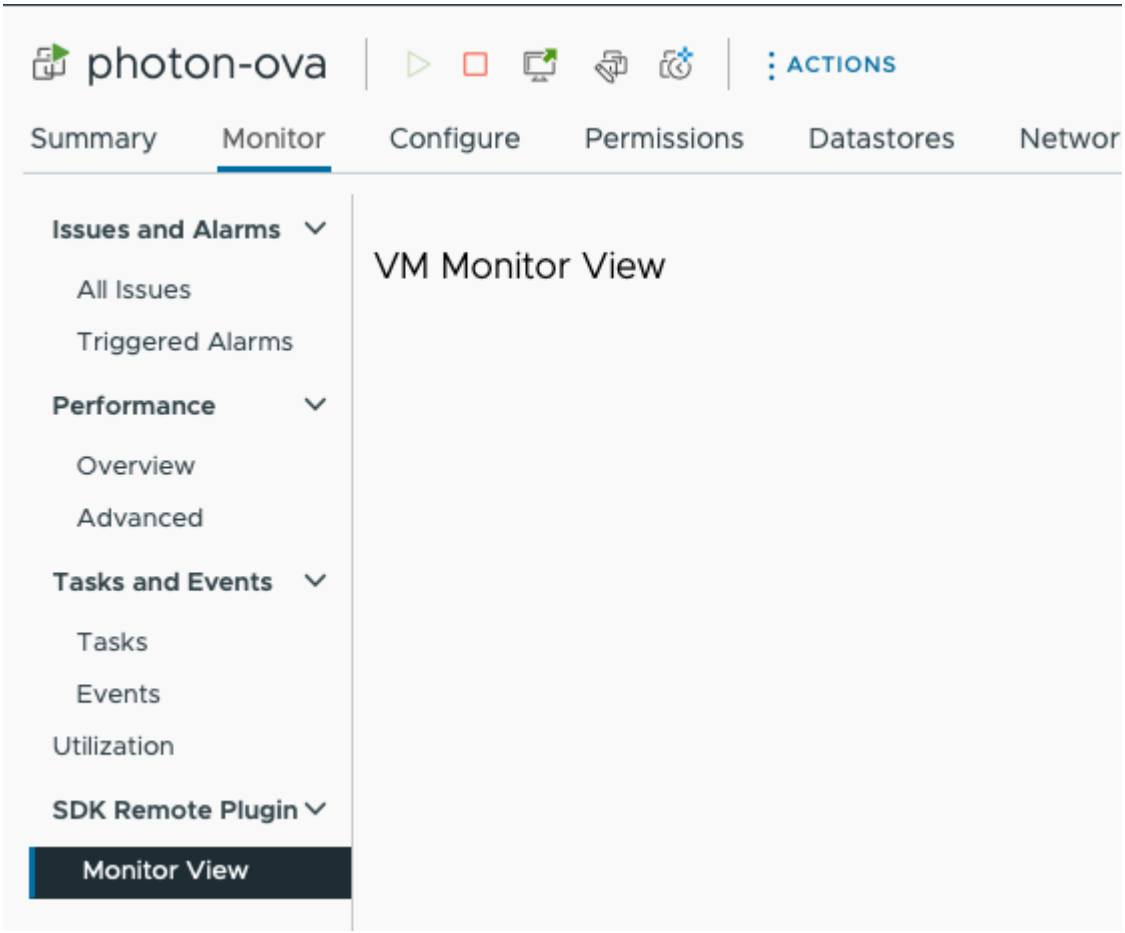
Figure 1-8. Card Extending the VirtualMachine Summary View



The VirtualMachine Summary View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
   "VirtualMachine": {
      "summary": {
         "view": {
            "uri": "index.html#/vm-card",
         }
      },
      ...
   }
}
```

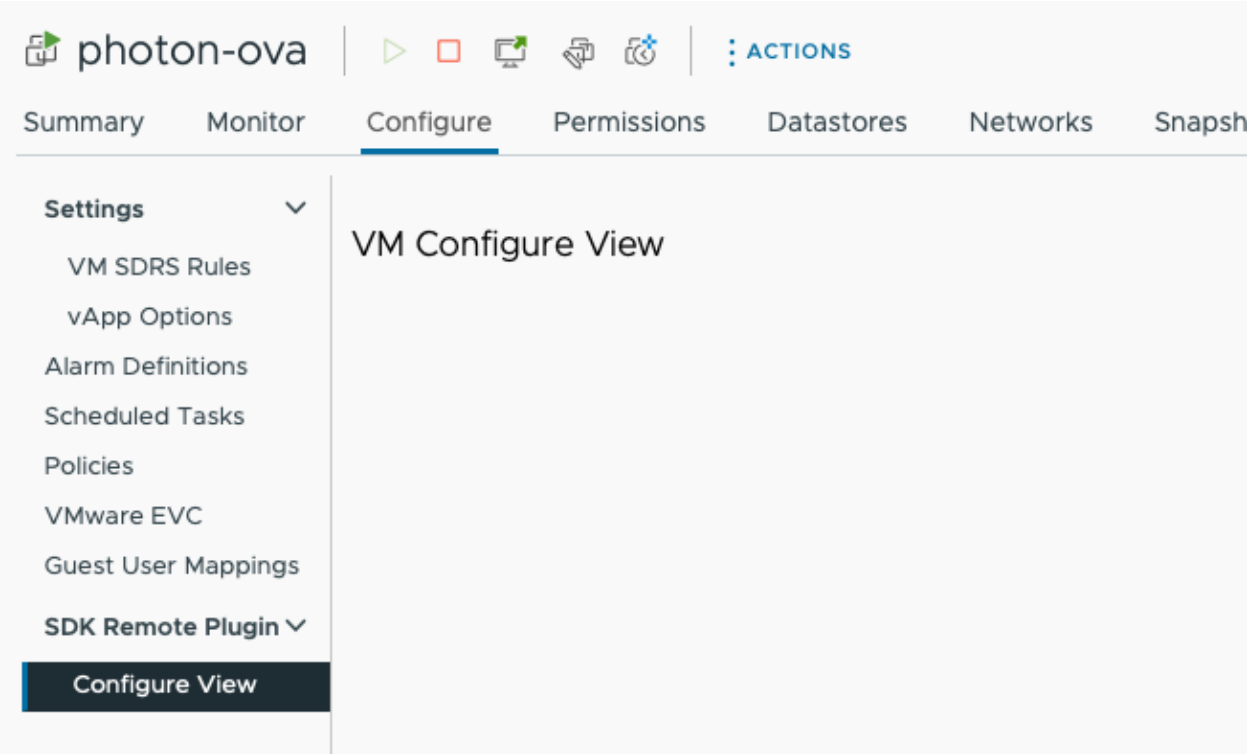
Figure 1-9. Extending the VirtualMachine Monitor View



The VirtualMachine Monitor View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
   "VirtualMachine": {
      ...
      "monitor": {
         "views": {
            "navigationId": "vmMonitor",
            "labelKey": "vm.monitor.view.title",
            "uri": "index.html#/vm-monitor",
         }
      }
   }
}
```

Figure 1-10. Extending the VirtualMachine Configure View



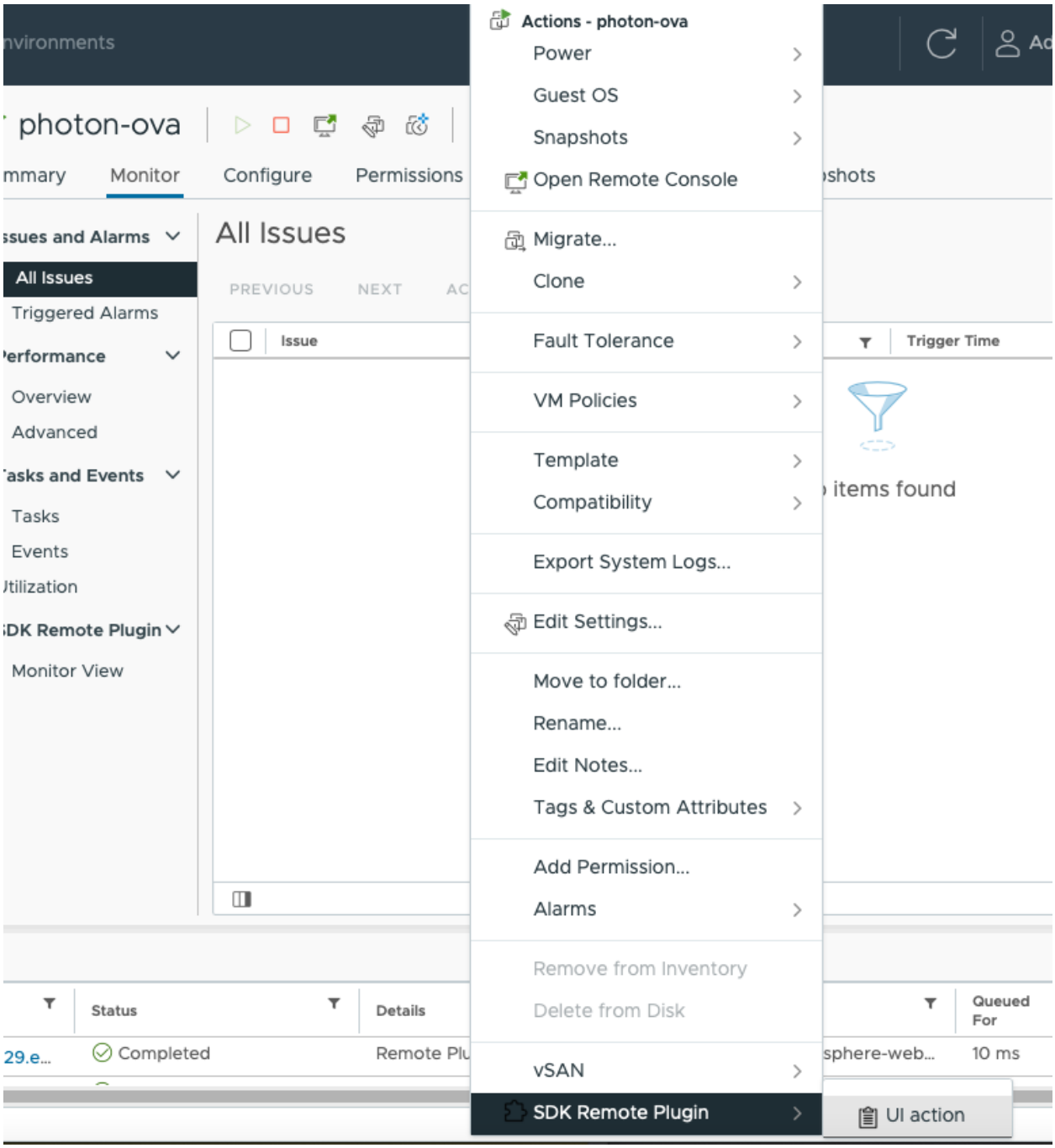
The VirtualMachine Configure View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
   "VirtualMachine": {
      "configure": {
         "dynamicUri": "dynamicItems/vm/configure",
         "views": {
            "navigationId": "vmConfigureView",
            "labelKey": "vm.configure.view.title",
            "uri": "index.html#vm-congfigure",
            "dynamic": true
         }
      }
   }
}
```

Note The Configure View is dynamic, which means that its appearance is conditional. For more information about dynamic views, see Chapter 6 Dynamic Extensions for Remote Plug-ins.

The remote plug-in sample shows how to add an action item to a context menu for a VirtualMachine object. You can see how this appears in the following illustration.

Figure 1-11. Action Added to VirtualMachine Context Menu



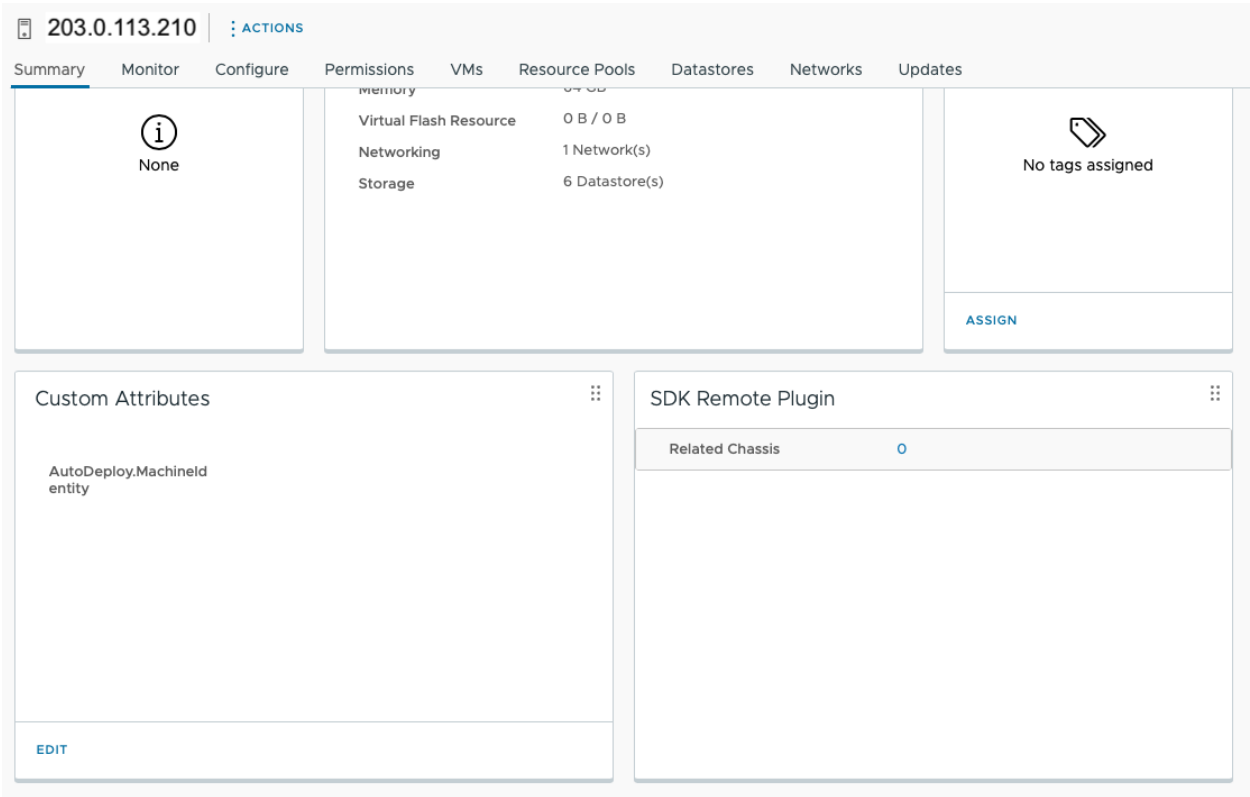
The VirtualMachine context menu extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
   "VirtualMachine": {
      "menu": {
         "dynamicUri": "dynamicItems/vm/actions",
         "actions": {
            "labelKey": "RemoteSample:vm.action.label",
```

```
            "icon": {
               "name": "action-vm"
            },
            "trigger": {
               "type"; "modal",
               "uri": "index.html#/vm-action-modal",
               "titleKey": "vm.action.model.title",
               "size": {
                  "width": 600,
                  "height": 205
               }
            },
            "acceptMultipleTargets": true,
            "dynamic": true
         }
      }
   }
}
```

The remote plug-in sample displays a HostSystem Summary view and a HostSystem Monitor view also. The Summary view shows the number of related Chassis objects. You click the number to reach the Monitor view. The Monitor view displays a datagrid listing all available Chassis objects. To create a relation between a HostSystem and a Chassis object, select the check box beside the Chassis object and click **Update**.

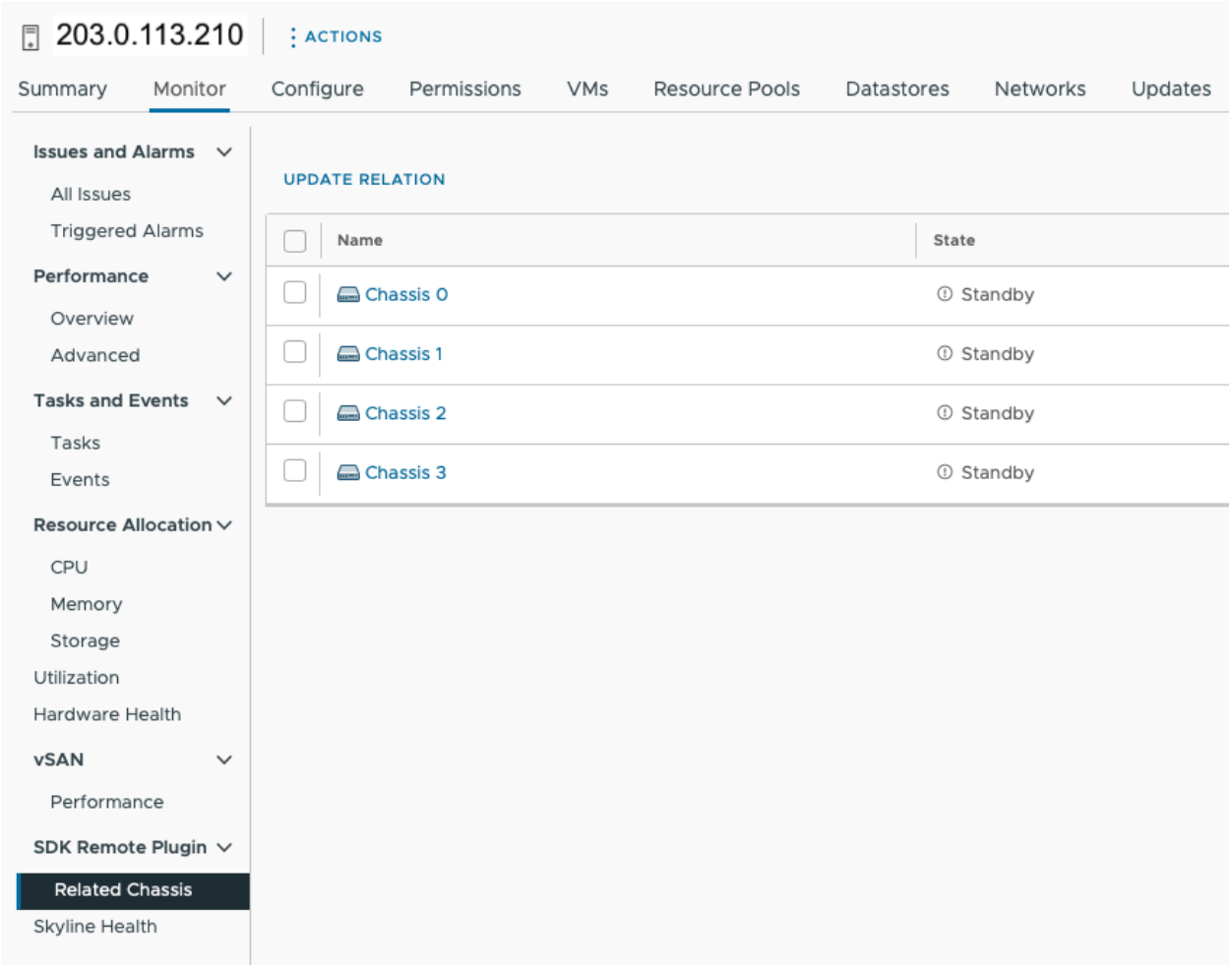
Figure 1-12. Card Extending the HostSystem Summary View



The HostSystem Summary View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
   "HostSystem": {
      "summary": {
         "view": {
            "uri": "index.html#/host-card"
         }
      }
   }
}
```

Figure 1-13. Extending the HostSystem Monitor View



The HostSystem Monitor View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
   "HostSystem": {
      "monitor": {
         "views": [
```

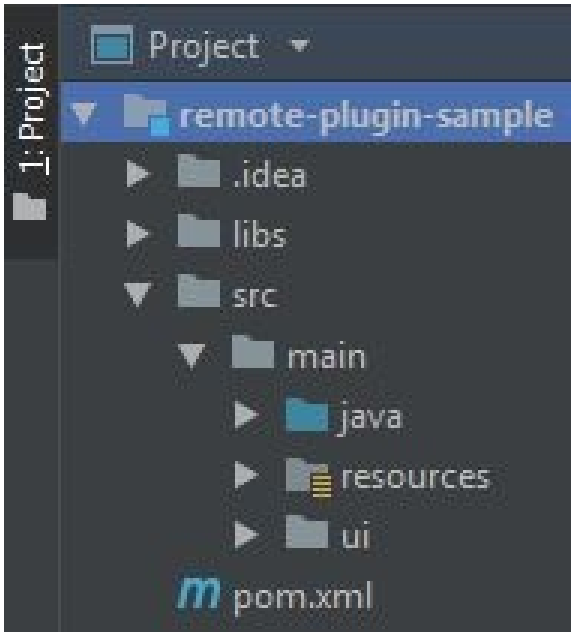
```
            {
               "navigationId": "hostMonitor",
               "labelKey": "host.monitor.view.title",
               "uri": "index.html#/host-monitor"
            }
         ]
      }
   }
}
```

Remote Plug-in Sample Directory Structure

The remote plug-in sample code is organized as a Maven project. You can locate certain important files in the project as follows.

Top-Level Directory

Figure 1-14. Remote Plug-in Sample Top-Level Directory



The top-level directory, `remote-plugin-sample`, is located within the `html-client-sdk/samples` directory of the SDK zip file.

- The `libs` subdirectory contains the external library `vim25.jar`, which must be installed as a local dependency during the project build and deploy process. The library allows access to vSphere managed objects. In vSphere 8.0 U2 the `vim25.jar` library was deleted from the Client SDK, so developers who don't have it must find it in the vSphere Management SDK.
- Nested inside the `src` directory you can find a `java` directory, which contains Java source files for the sample plug-in server, at `java/com/vmware/sample/remote/*`.

- Also nested inside the `src` directory you can find a `ui` directory, which contains source code for the sample plug-in UI, at `ui/src/app/{model, service, views}/*`.
- To build the project, use the `pom.xml` file in the top-level directory of the remote plug-in sample, `html-client-sdk/samples/remote-plugin-sample/pom.xml`.

Remote Plug-in Architecture in the vSphere Client

2

The vSphere Client remote plug-in architecture is designed to integrate plug-in functionality into the vSphere Client without the need to run inside vCenter Server. This provides plug-in isolation and enables scale-out of plug-ins that operate in large vSphere environments. The remote plug-in architecture provides the following benefits:

- Your plug-in is protected from interference by unstable or compromised plug-ins loaded in the same vSphere Client.
- An incompatible plug-in does not interfere with vCenter Server operation.
- Plug-in compatibility is robust across vCenter Server upgrades.
- You can deploy a number of plug-in versions within the same vSphere environment. Your remote plug-in user interface needs to communicate with only a single back-end server.
- The topology of deployed plug-ins is well defined and easy to troubleshoot.

Read the following topics next:

Components of the vSphere Client Architecture

The vSphere Client architecture enables administrators to manage vSphere environments of varying scale and complexity with a single user interface. It supports environments ranging from a single vCenter Server to a number of vCenter Server instances in Enhanced Linked Mode or Hybrid Linked Mode.

The user interface component runs in a browser, managing and displaying HTML5 views. It communicates with the vsphere-ui service, requesting HTML and Javascript files with vSphere inventory data, authenticating as needed. The user interface also manages a sandbox for each active plug-in and provides client library services to the plug-in user interface.

The vsphere-ui service is an OSGi Java application server that runs on every vCenter Server node. The vsphere-ui service communicates with all of the services provided by vCenter Server by using a variety of API styles and protocols. The vsphere-ui service maintains an authenticated session connection as a client of each of the services.

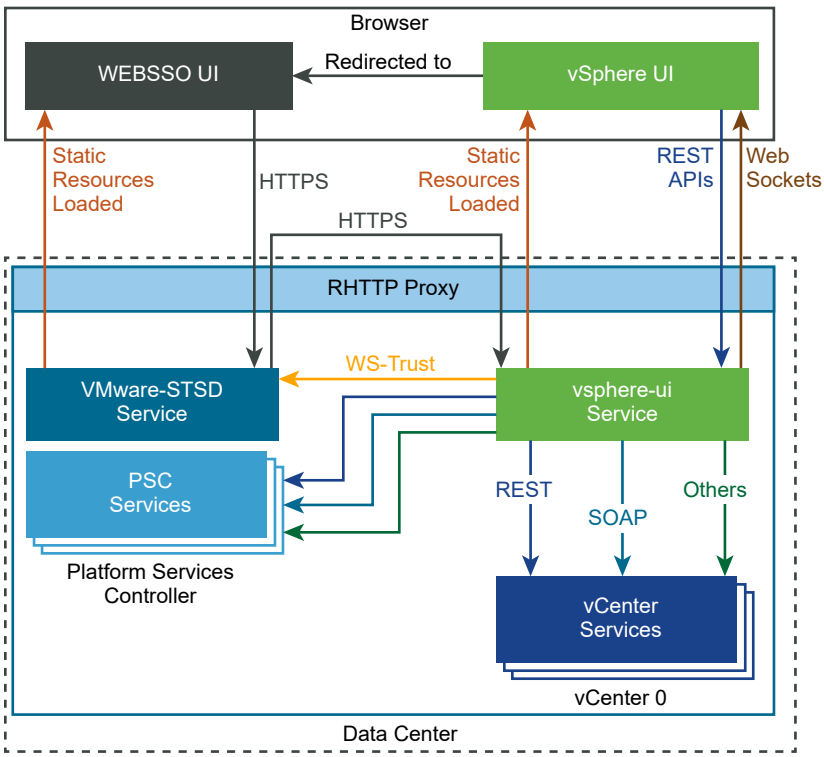
The vsphere-ui service also provides REST and Web Sockets APIs to the vSphere Client user interface running in the browser. The service supports authentication for users of the vSphere Client by redirecting the browser to a login user interface provided by the VMware vCenter Single Sign-on service.

vCenter Server Configurations

Remote plug-ins can operate in linked mode configurations of vCenter Server as well as in single vCenter Server instances. Illustrations in this chapter show the components and communication paths for both kinds of configuration.

The following illustration shows a single vCenter Server communicating with a web browser.

Figure 2-1. vCenter Server and Web Browser Communicatiions



In a linked mode configuration, the vsphere-ui service handles delegation of requests to the linked vCenter Server instances, using the same protocols that it uses when communicating with its own services..

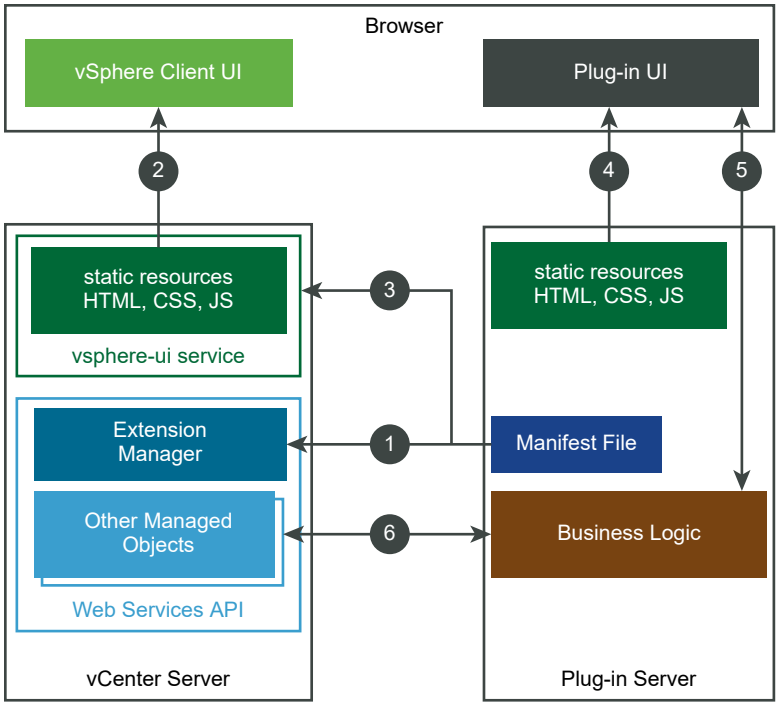
Communication Paths in Remote Plug-in Architecture

This diagram shows some of the communication paths between a plug-in, its user interface, and the vCenter Server to which the user interface is connected.

On the front end, the vsphere-ui service downloads and parses the plug-in manifest and serves UI components to the browser, including references to plug-in components, which are served by the plug-in server. These paths use simple HTTPS communications.

The back-end communication paths between the plug-in server and the vCenter Server Web Services API use SOAP messages over HTTPS. These communications are described in more detail in Communication Paths for Authentication in the Remote Plug-in Server.

Figure 2-2. A Simplified View of Remote Plug-in Architecture



The circled numbers identify the following data paths:

1. The remote plug-in installer registers the plug-in manifest file with the vCenter Server Extension Manager, by using the Web Services API.

2. A web browser downloads user interface elements of the vSphere Client from the vsphere-ui service in vCenter Server.

3. The vsphere-ui service downloads and parses the plug-in manifest file to determine where the plug-in extends the user interface.

4. The browser downloads user interface elements of the plug-in from the plug-in server.

5. The plug-in user interface requests data from the plug-in server.

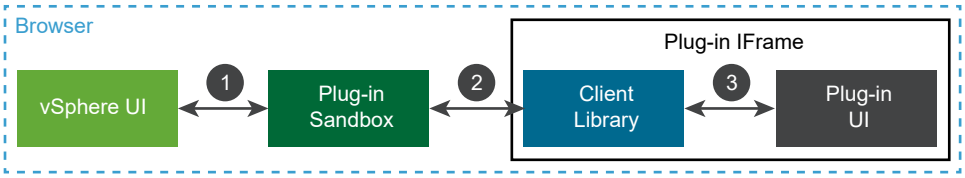
6. The plug-in server uses the Web Services API to interact with vCenter Server.

Communications Among UI Components in the vSphere Client

The vSphere Client user interface loads both its own components and the components belonging to the plug-in user interface. The vsphere-ui service reads the plug-in manifest file to determine where it should insert plug-in components in the user interface.

The user interface components loaded in the browser are organized as shown in the following diagram.

Figure 2-3. User Interface with a Sandboxed Plug-in



Paths:

1 Internal JavaScript methods

2 `window.postMessage()` method in browser

3 Public JavaScript methods

The plug-in user interface operates within its own iFrame, isolated from other plug-ins. The plug-in loads a copy of the vSphere Client JavaScript API client library, which is its sole connection to other client code. The plug-in code communicates with the client library code using JavaScript method calls.

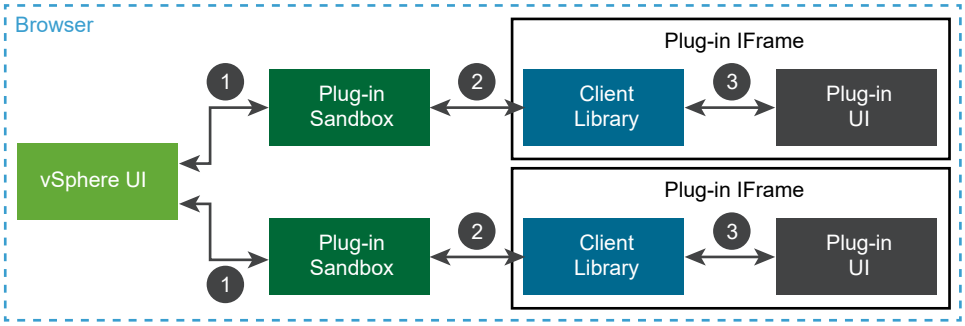
The client library communicates with the sandbox component that the vSphere Client provides to interface with the plug-in UI components. The communication with the sandbox uses the browser's `window.postMessage()` API. This makes it possible for the iFrame and its parent window to be loaded from different origins.

Note In the vSphere 6.7 Update 1 release of the vSphere Client, the iFrame and its parent window share the same origin. Do not depend on this to remain the same in future releases. In cloud environments, the iFrame is loaded from a secondary host.

The plug-in sandbox communicates with other components of the vSphere Client user interface by using internal Javascript APIs.

If the vSphere Client has more than one plug-in active, each plug-in is allocated its own sandbox, and operates within its own iFrame, as shown in the following illustration.

Figure 2-4. User Interface with Two Plug-ins in Separate Sandboxes



Paths:

1 Internal JavaScript methods

2 `window.postMessage()` method in browser

3 Public JavaScript methods

In this case, each plug-in UI communicates with its own back-end server.

Client-Server Communications with Remote Plug-ins

Client requests to back-end services are handled in a similar way for plug-in user interface components and vSphere Client user interface components.

A plug-in sandbox runs outside the plug-in iFrame. The sandbox component, along with the rest of the vSphere Client user interface components, sends requests to the vsphere-ui service of the vCenter Server instance that the browser connected to. Service requests use REST and Web sockets over HTTPS. All requests pass through the vCenter Server reverse proxy, which routes them to the correct server components.

A plug-in user interface sends service requests to the plug-in back end, using any RPC style on top of the HTTPS transport protocol. All requests pass through the vCenter Server reverse proxy, which routes them to the plug-in server. The proxy routing is configured at the time the plug-in is installed; it simplifies dealing with self-signed certificates, and avoids problems with cross-origin resource access.

Security Concepts for Remote Plug-ins

Remote plug-ins typically use the HTTP protocol as a transport for requests, whether using REST or SOAP requests. Authentication methods vary, depending on the target endpoint.

Client-side sessions with REST endpoints are tracked with a session token passed in a custom HTTP header named `webClientSessionId`. Server-side sessions with SOAP endpoints are tracked with a cookie-based session ID.

A plug-in developer can choose what form of authentication suits the plug-in server component. A best practice is to authenticate by using the session token that the plug-in user interface can get from the client library.

To use the client-side session token, the plug-in server sends the token to a specific REST endpoint of the vsphere-ui service. The vsphere-ui service verifies the authenticating token, and then returns a session clone ticket. The plug-in server uses the clone ticket with the vSphere Web Services API to obtain a SOAP session. The authentication process is described in more detail in Chapter 10 Advanced Considerations for Remote Plug-in Servers.

Creating a Remote Plug-in for the vSphere Client

3

This chapter outlines how to create your own remote plug-in for the vSphere Client. The goal is to illustrate the entire process using simple components. Other chapters deal with some of the steps in more detail.

To create a remote plug-in, you need both front-end components and back-end components. The front-end components constitute a user interface based on HTML 5. The back-end components include a web server and business logic in support of the front end.

For the front end of your plug-in, you can choose any JavaScript-compatible language. The Remote Plug-in Sample in the vSphere Client SDK uses Angular and Clarity. A best practice is to use Clarity elements to harmonize with the look and feel of the vSphere Client.

For the back end, you can choose any language and any web server. The SDK samples are written in Java and built with Maven, but there is no restriction on the language or tools that you use for back-end development. The SDK samples use Tomcat as a web server because it is bundled with Spring.

Read the following topics next:

- Code Components for a vSphere Client Remote Plug-in
- Deployment Requirements for a vSphere Client Remote Plug-in
- Using Auxiliary Plug-in Servers
- vSphere Client Plug-in Registration Tool
- Sample Manifest File for a Remote Plug-in

Code Components for a vSphere Client Remote Plug-in

A remote plug-in for the vSphere Client includes several code components that you create.

To create code for a complete plug-in, you must do the following:

- Choose the extension points where your user interface views will extend the vSphere Client.
- Create front-end views that provide the user interfaces for data access.
- Create back-end controllers that interface between services and user interface views.
- Create models for your logical data objects.

- Create back-end services that translate between data models and the objects in storage that back the models.
- Import `vim25.jar`, which as of vSphere 8.0 U2 is available only in the vSphere Management SDK, to access the Web Services SDK. The `vim25` library provides RPC access to the PropertyCollector, used to retrieve properties of vCenter Server managed objects.

Deployment Requirements for a vSphere Client Remote Plug-in

To prepare a remote plug-in for deployment, you must prepare at least one file and launch one or more processes. The simplest plug-in can be implemented with a single back-end server and a plug-in manifest file that describes the extension views and other details about the plug-in.

Deployment of a remote plug-in takes place at run time, but you must make preparations in advance. You need to do the following to prepare for plug-in deployment:

- Run a web server that provides plug-in components on demand.
- Run your plug-in server binary. The plug-in server runs on a virtual or physical machine of your choice, but it must be the same machine as the web server.
- Prepare a plug-in manifest file, `plugin.json`, that specifies the plug-in components. See also Sample Manifest File for a Remote Plug-in The manifest file must be accessible by HTTPS, on the same machine as the web server.
- Register your plug-in with a vCenter Server instance. You can register either by using the registration script in the SDK or by writing your own registration tool. In either case, you need the URL and credentials to access the vCenter Server instance. You also need the security certificate or thumbprint of your plug-in manifest server, and the URL of your plug-in manifest file. The connection should be secure HTTP (HTTPS).

Note For production installations, a best practice is to create a registration tool that does not expose vCenter Server credentials on the command line.

Note When registering the plug-in, it's highly recommended to provide the SSL certificate of the plug-in server in addition to the thumbprint of the SSL certificate of the plug-in server. Performing a full SSL certificate check during SSL handshake is more secure than performing an SSL certificate thumbprint check. Furthermore, in future vSphere releases, support for SSL certificate thumbprints will be dropped and full SSL certificates will be required.

To register a plug-in by using the script in the SDK, you use the `registerPlugin` function of the `extension-registration` script. The script arguments are demonstrated in the following example.

```
./extension-registration.sh  -action registerPlugin  -remote \
-url https://myvcenter/sdk \
-username administrator@vsphere.local  -password mysecret \
-key com.mycompany.myplugin  -version 1.0.0 \
-pluginUrl https://mydevbox:8443/myplugin/plugin.json \
```

```
-serverThumbprint 19:FD:2B:0E:62:5E:0E:10:FF:24:34:7A:81:F1:D5:33:\
19:A7:22:A0:DA:33:27:07:90:0F:8E:8D:72:F1:BD:F1 \
-serverCertificateFile /tmp/plugin-server.cer \
-vCenterServerThumbprint 2A:0E:3C:1F:73:6F:1F:21:00:35:45:8B:92:02:E6:44:\
2A:B8:33:B1:EB:44:38:18:A1:10:9F:9E:83:02:CE:02 \
-c 'Example, Inc.'  -n 'Remote Plug-in'  -s 'This is a remote plug-in'
```

To register a plug-in by means of your own tool, you must create an `Extension` type record in the `ExtensionManager` managed object. The `Extension` record must describe the plug-in manifest server in two places: `client[0]` and `server[0]`. These two array entries are similar in several properties, but have important differences:

- The `client` array, which has type `ExtensionClientInfo[]`, contains the plug-in version in the first element. This is required so that the vSphere Client can manage plug-ins correctly.
- The `client` array must have a `type` property with a value of `vsphere-client-remote` in the first element.
- The `server` array, which has type `ExtensionServerInfo[]`, must have a `type` property with any value you choose in the first element. A best practice is to assign the value `MANIFEST_SERVER` to the first element of the array, to identify the manifest server for the plug-in.
- To support HTTPS connections, the `server` array must have either `serverThumbprint` or `serverCertificateFile` or both in the first element. Property `serverThumbprint` takes the value of the SHA-256 hash of the server certificate, while `serverCertificateFile` takes the value of the PEM-encoded server certificate (string).

  Note HTTP connections are not recommended for production use with the vSphere Client.

Both the `client` array and the `server` array must have the same value in the `url` property of the first array element. The value must be the URL of the plug-in manifest server.

For information about the plug-in manifest, see Sample Manifest File for a Remote Plug-in. For information about auxiliary plug-in processes, see Using Auxiliary Plug-in Servers. For information about the plug-in registration script, see vSphere Client Plug-in Registration Tool.

Using Auxiliary Plug-in Servers

A remote plug-in package must include a primary server that serves the plug-in manifest file. A plug-in can also include auxiliary servers that serve other plug-in resources.

Plug-in packages must be downloaded with secure HTTP (HTTPS) protocol.

You might want to design a plug-in with auxiliary servers for several reasons:

- To provide load balancing
- To separate performance tiers
- To manage security risks

- To localize database access
- To take advantage of different coding languages or frameworks
- To facilitate re-implementation

For more information about auxiliary servers, see Chapter 10 Advanced Considerations for Remote Plug-in Servers.

vSphere Client Plug-in Registration Tool

The vSphere Client SDK contains a plug-in registration tool that manages plug-in extension registration records in the vCenter Server ExtensionManager. The tool registers, unregisters, and updates the registration record of a plug-in.

Plug-in Registration Script

To register a plug-in, use this registration script in the SDK. It's a wrapper for a Java tool, which opens a session with a vCenter Server instance using the VMware Web Services API.

```
tools/vCenter plugin registration/prebuilt/extension-registration.sh
```

Plug-in Registration Script Syntax

The arguments of the registration script function as follows:

- **`-action`** (required) can be one of:
  - **`registerPlugin`**
  - **`updatePlugin`**
  - **`unregisterPlugin`**
  - **`isPluginRegistered`**
- **`-c`** or **`-company`** is the name of the plug-in vendor.
- **`-insecure`** bypasses security checks on the vCenter Server certificate. This is not recommended for a production environment.
- **`-k`** or **`-key`** (required) is an identification string for the plug-in. The plug-in registration record in the vCenter Server ExtensionManager contains this identification.
- **`-local`** (default) is used to register or update a local plug-in. See also **`-remote`**.
- **`-n`** or **`-name`** is a user-friendly identification string for the plug-in.

- **-p** or **-password** (required) authenticates the vCenter Server user account. See also **-username**.
- **-ps** or **-pluginServers** is a string that contains a JSON array of objects that specify endpoints for auxiliary services belonging to the plug-in. Each endpoint object must contain a `url` property to enable connections to the endpoint. The scheme must specify HTTP or HTTPS as the protocol. To support HTTPS connections to an endpoint, at least one of the fields `serverThumbprint` or `serverCertificateFile` must be set in the endpoint object. HTTP connections are not recommended for production use with the vSphere Client. The `type` property, required for auxiliary servers, is a service registration identifier string by which the plug-in front end can discover a needed endpoint.

  Choice and optional properties that can also be present in the object are:

  - `serverThumbprint` is the thumbprint (fingerprint) of the auxiliary server's SSL certificate. See constraints for the `-st` argument below.
  - `serverCertificateFile` is the path of the auxiliary server's SSL certificate on the local filesystem. See constraints for the `-scf` argument below. The certificate of the auxiliary server must contain the host name specified according to subjectAltName extension (RFC 2818, Section 3.1 Server Identity).
  - `label` (not currently used)
  - `summary` is a brief description of the auxiliary server.
  - `company` is the name of the plug-in server vendor.
  - `adminEmail`
- **-pu** or **-pluginUrl** (required to register a plug-in) is the URL of the plug-in manifest served by the plug-in back end. The scheme must specify HTTP or HTTPS as the protocol. The path segment of the `-pluginUrl` must be specified relative to the directory where you run the plug-in server. **
- **-remote** (required for a remote plug-in) is used to register or update a remote plug-in. See also **-local**.
- **-s** or **-summary** is a brief description of the plug-in.
- **-show** or **-showInSolutionManager** specifies that the plug-in will appear in the Solutions list of the Administration panel.
- **-st** or **-serverThumbprint** is the SHA-256 signature of the plug-in back-end server certificate. Character pairs should be separated by colon separators.
- **-scf** or **-serverCertificateFile** is the path on the local filesystem of the plug-in manifest server's SSL certificate file, in PEM or DER format. The certificate of a plug-in server record must contain the host name of the plug-in server in its `subjectAltName` extension as per RFC 2818, Section 3.1 Server Identity.

- **-u** or **-username** (required) identifies a vCenter Server user account that has permission to write to the vCenter Server ExtensionManager. See also **-password**.
- **-url** (required) is the URL of the **/sdk** resource of the vCenter Server. Use the fully qualified domain name of the vCenter Server instance. For example: **https://my-vcsa.example.com/sdk**
- **-v** or **-version** (required) identifies the plug-in version.
- **-vct** or **--vcenterServerThumbprint** supplies the certificate thumbprint for the vCenter Server instance. You can use this in development environments when the certificate is self-signed or otherwise not recognized by the browser.
- **-eventList** - path to the event list JSON file, relative to the file system root of the manifest server. The file should contain a JSON formatted array of event infos. Each object in the array must specify the *eventId* of the event. In addition, an optional XML descriptor for the `EventType` can be specified.
- **-faultList** - path to the fault list JSON file, relative to the file system root of the manifest server. The file should contain a JSON formatted array of fault infos. Each object in the array must specify the *faultId* of the fault.
- **-privilegeList** - path to the privilege list JSON file, relative to the file system root of the manifest server. The file should contain a JSON formatted array of privilege group objects. Each object in the array must specify the *groupId* of the privilege group and the privileges in that group. Each object in the `privileges` array must specify an *privilegeId* of the privilege.
- **-resourceList** - path to the resource list JSON file, relative to the file system root of the manifest server. The file should contain a JSON formatted object, where the keys are locales (for example 'en', 'fr', 'de' ..) and the values are objects with a key to be localized and the value is the localizable message.
- **-taskList** - path to the tasks list JSON file, relative to the file system root of the manifest server. The file should contain a JSON formatted array of task infos. Each object in the array must specify the *taskId* of the task.

** To support HTTPS connections to the plug-in manifest server, at least one of the fields `serverThumbprint` or `serverCertificateFile` must be set in the endpoint object. Insecure HTTP connections are not recommended for production use with the vSphere Client.

Note When registering a plug-in, it's highly recommended to provide SSL certificates of the plug-in servers in addition to the thumbprints of the SSL certificates of the plug-in servers. Performing a full SSL certificate check during SSL handshake is more secure than performing an SSL certificate thumbprint check. Furthermore, in future vSphere releases, support for SSL certificate thumbprints will be dropped and full SSL certificates will be required.

Sample Manifest File for a Remote Plug-in

The following sample manifest file contains the minimum elements required for a simple plug-in.

```
{
   "manifestVersion" : "1.0.0",
   "requirements": {
      "plugin.api.version": "1.0.0"
   },
   "configuration": {
      "nameKey": "plugin.name",
      "icon": {
         "name": "main"
      }
   },
   "definitions": {
      "i18n": {
         "locales": ["en-US"],
         "definitions": {
            "plugin.name": {
               "en-US": "Hello World"
            }
         }
      }
   },
   "global": {
      "view": {
         "uri": "index.html"
      }
   }
}
```

Note The URIs specified in the manifest file are relative to the location of the manifest file itself. That is, the directory containing `plugin.json` should be considered the server root directory for plug-in resources.

Deploying Remote Plug-ins for the vSphere Client

4

A remote plug-in must be deployed by the vSphere Client before it can be used. Deployment is the process of preparing components of vCenter Server to accept communications from the plug-in and to display plug-in views in the browser. Developers have a choice of deployment topologies, as explained later in this chapter.

Read the following topics next:

Remote Plug-in Life Cycle

Remote plug-in content becomes visible in the vSphere Client after you prepare and register the plug-in with a vCenter Server instance.

The plug-in displays when you connect a browser to the vSphere Client URL of a vCenter Server instance, or of a vCenter Server linked to the instance where the plug-in is registered, and log in.

A remote plug-in has the following life cycle stages:

Build

The plug-in developer assembles necessary resources and builds the package(s) to implement the remote plug-in.

Run

The plug-in developer starts the plug-in manifest server.

Registration

The plug-in developer registers the plug-in as a vCenter Server extension, using the ExtensionManager managed object of the vCenter Server instance.

Discovery

A vCenter Server instance discovers a new plug-in registration, or a new version registration.

Deployment

A vCenter Server instance downloads the plug-in manifest file, verifies certification and compatibility, and configures its reverse proxy to route server requests.

Use

The browser downloads plug-in resources from a back-end server into an iFrame of the vSphere Client user interface, and the plug-in user interface operates in conjunction with the back-end server or servers.

Uninstallation

The developer unregisters the plug-in, and all linked vCenter Server instances delete the routing configuration.

Remote Plug-in Deployment

When the vSphere Client discovers a remote plug-in, it schedules the plugin for deployment. vCenter Server takes the following steps to deploy a remote plug-in in the vSphere Client:

- Download the remote plug-in manifest, `plugin.json`, from the location specified in the `ClientInfo` property registered in the ExtensionManager. When you use the extension-registration script provided in the SDK this is the value of the `-pu` or `-pluginUrl` argument. vCenter Server downloads your plug-in manifest into `/etc/vmware/vsphere-ui/vc-packages/vsphere-client-serenity/`*`your_plugin_name-your_plugin_version`*.
- Parse the plug-in manifest to determine whether its specified version and environment are compatible with deployment on this vCenter Server instance.
- Parse the plug-in manifest to determine what views will be shown in the user interface, and add them to an internal extension point table.
- Configure the vCenter Server reverse HTTP proxy to route plug-in UI traffic to the remote plug-in server or servers.

After these steps complete successfully, the vSphere Client user interface displays a notification that the remote plug-in is installed.

Plug-In Caching

When the vSphere Client installs a plug-in, it downloads the plug-in manifest file and caches it. The cached copy is re-used whenever the vsphere-ui process restarts.

After a plug-in has been unregistered, the vsphere-ui service detects the change immediately. The cached copy of the plug-in manifest is deleted from the cache.

When a plug-in is upgraded, the vsphere-ui service detects the change as soon as the plug-in registration entry is updated with a new version number. At that time, the old plug-in version is undeployed, the cached copy of the manifest is removed, and the new plug-in version is deployed.

If plug-in resources are upgraded without changing the version number, such as when developing a plug-in, the vsphere-ui service does not detect the change. To deploy plug-in changes during development, use `pluginDeveloperMode` to enable the **`Redeploy`** button in the user interface. For more information, see Redeploying Plug-ins During Development.

Remote Plugin Uninstallation

Unregistering a plug-in package from vCenter Server causes the vSphere Client service to delete the plug-in from the environment. A vCenter Server instance notifies all linked vCenter Server instances as soon as the plug-in is unregistered. The linked vCenter Server instances undeploy the plug-in, by using the following steps:

- Delete the plug-in from internal extension point tables.
- Delete the plug-in's reverse proxy routing rules.
- Delete the plug-in's manifest from cache.

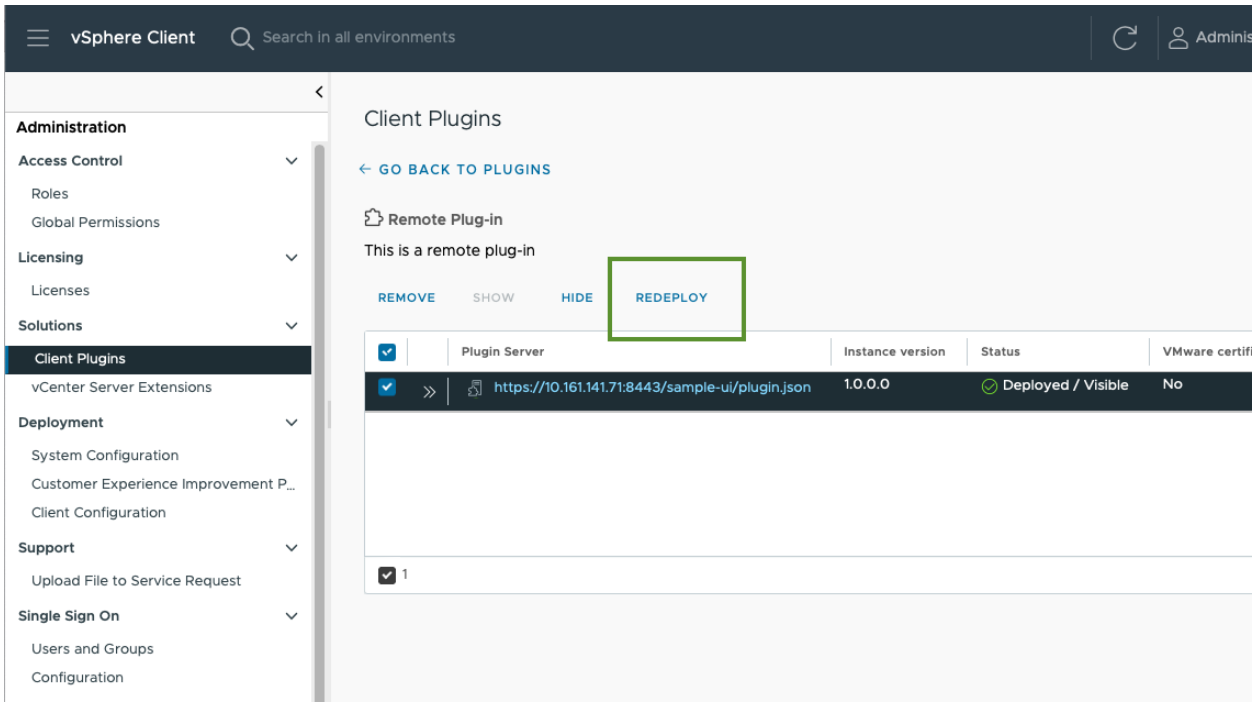
While the process of uninstallation takes place, the vSphere Client UI does not wait for the process to complete. You might need to refresh the browser window after the vSphere Client displays a notification that the uninstallation is complete.

Redeploying Plug-ins During Development

The vSphere Client service discovers new remote plug-ins as soon as they are registered with any linked vCenter Server instance. Some later changes to the registration record can cause redeployment of the plug-in. To bypass the need to change the registration record during plug-in development, you can trigger redeployment of plug-in changes by using the **`Redeploy`** feature.

When you are developing and testing a plug-in, you can trigger redeployment frequently by using the **`Redeploy`** button, without the need to make changes to the extension registration record. After the **`Redeploy`** button is enabled, it is visible in the **Client Plug-Ins** window, where it applies to any selected plug-in. The button causes vCenter Server to undeploy the selected plug-in, and then deploy it again based on the same registration record.

Figure 4-1. Redeploy feature in Plugin Developer Mode



To enable the **Redeploy** button, set `pluginDeveloperMode` by appending the query parameter `pluginDeveloperMode=true` to any URL loaded into the browser window. For example, the URL could be the following:

`https://vcenter-server-fqdn/ui/?pluginDeveloperMode=true`

When you set `pluginDeveloperMode`, it remains in effect until you browse to a URL that contains the query parameter `pluginDeveloperMode=false`, or until you refresh the browser window while the browser address field does not include `pluginDeveloperMode=true`.

Note The **Redeploy** button requires the `Plugin.Management` privilege, which is available by default to all users who have the Administrator role.

Specifying Remote Plug-in Compatibility

The vSphere Client provides a way to specify compatibility between remote plug-ins and vCenter Server versions, and also compatibility with public cloud and private cloud environments.

When a remote plug-in is registered with a vCenter Server instance, each linked instance is notified. Each notified instance responds by checking compatibility constraints specified in the plug-in manifest. The constraints determine whether the notified instance will involve the plug-in in future requests that access vSphere resources on the registration instance.

Each notified instance must satisfy client constraints, and the registration instance must satisfy server constraints, to enable plug-in traffic. If both sides meet the constraint conditions, the notified instance deploys the plug-in for future connections to the registration instance.

Deployment includes configuring extension point metadata to deploy plug-in views in the vSphere Client user interface, and setting proxy rules to forward view requests to the plug-in server. This applies to views that access vSphere resources that the vCenter Server manages.

In the plug-in manifest file you can specify two points of control for plug-in compatibility. First, you can control whether any vCenter Server instance is compatible with the plug-in's user interface views. You specify this with the `vsphere.client` property. Second, you can control whether the plug-in is compatible with a vCenter Server instance where it is registered. You specify this with the `vcenter.server` property.

For example, if your plug-in is compatible with only an on-premises vCenter Server, you should specify that limitation in the `vcenter.server` object in the manifest. If the plug-in is registered in a cloud environment by mistake, vsphere-ui will refuse to deploy it. Or if your plug-in is incompatible with vSphere Client versions before 8.0, you should specify a minimum version of 8.0 in the `vsphere.client` manifest object. If a user connects to an earlier version vCenter Server, the client refuses to display any view served by the plug-in.

To specify compatibility constraints, modify the `requirements` object in the remote plug-in manifest file. The following nested objects are available to specify compatibility:

- `vcenter.server`

  Use this object to specify environments and versions of vCenter Server on which the plug-in can be deployed. You can specify these properties:

  - `environments`

    A specifier array for vCenter Servers. If property `environments` is specified in object `vcenter.server`, the plug-in can only be deployed there. If this property is absent, the plug-in can be deployed from any vCenter Server it is registered with. Legal values:

    - `onprem`
    - `cloud`

  - `version`

    A string containing a version or range of versions for vCenter Server instances where the plug-in can be registered and deployed. Legal formats:

    - *`version`* = a single compatible version string
    - `[`*`version`*`, `*`version`*`]` = an inclusive range of compatible versions
    - `(`*`version`*`, `*`version`*`)` = an exclusive range of compatible versions
    - `[`*`version`*`, )` = a minimum version
    - `(, `*`version`*`)` = all prior versions

where *version* is an integer or a series of dot-separated integers: *`n`*`[.`*`n`*`[.`*`n`*`]]`

Although bracket and parenthesis look unmatched, "[" is inclusive and "(" is exclusive. For example [8.0, 9.0) means everything from 8.0 up to but not including 9.0.

A best practice is to specify the form `[version, version)` to indicate compatibility that begins at one major release and includes all minor releases up to, but not including, the next major release.

The `version` property in this context applies to a vCenter Server instance with which a remote plug-in is registered. If a version constraint is present in the `vcenter.server` object, `vsphere-ui` processes in all linked vCenter Server instances will verify that the registration instance satisfies the version constraint. If it does not, `vsphere-ui` will not deploy the plug-in for connections to the registration instance of vCenter Server. If the `version` property is absent, any `vsphere-ui` is free to deploy the plug-in if other compatibility constraints are satisfied.

- `vsphere.client`

  Use this object to specify environments and versions of vSphere Client that can display the views served by the plug-in. When the plug-in is registered on one vCenter Server instance, and a user connects to a second vCenter Server instance, the `vsphere-ui` of the second instance will display plug-in views for resources managed by the registration instance only if the second instance satisfies the `vsphere.client` compatibility requirements.

  - `environments`

    A specifier array for vCenter Servers. If property `environments` is specified in object `vcenter.server`, the plug-in can only be deployed there. If this property is absent, the plug-in can be deployed from any vCenter Server it is registered with. Legal values:

    - `onprem`
    - `gateway`
    - `cloud`

  - `version`

    A string containing a version or range of versions for vSphere environments. If the `version` property is present in the `vsphere.client` object, it limits the versions that can deploy the plug-in to access resources on a vCenter Server instance where the plug-in is registered. Legal formats are:

    - *`version`* = a single compatible version string
    - `[`*`version`*`, `*`version`*`]` = an inclusive range of compatible versions
    - `(`*`version`*`, `*`version`*`)` = an exclusive range of compatible versions
    - `[`*`version`*`, )` = a minimum version
    - `(, `*`version`*`)` = all prior versions

where *version* is an integer or a series of dot-separated integers: *`n`*`[.`*`n`*`[.`*`n`*`]]`

Remote Plug-in Topologies

Enhanced linked mode (ELM) enables you to manage all linked vCenter Server instances from a single vSphere Client connected to any one of the linked vCenter Server instances. The way that

the vSphere Client visualizes the vSphere resources on a vCenter Server instance depends on the set of remote plug-ins installed on that instance. The topology of remote plug-ins is the overall configuration of plug-ins that determines how the vSphere Client interacts with the resources on different vCenter Server instances.

Remote Plug-in Terminology

The following terms are useful to understand concepts in this chapter.

- Plug-in – A product that extends the vSphere user interface with additional functionality.
- Plug-in Version – The version of the plug-in specified at the time of registration.
- Plug-in Instance – A plug-in server registered with a vCenter Server instance. The plug-in instance is defined by the plug-in product, version, and plug-in manifest URL.
- Plug-in UI – The part of the plug-in loaded inside an iFrame within the vSphere Client UI in the browser.
- Plug-in Server – A back-end process to which the plug-in UI talks.
- Plug-in Manifest Server – The plug-in process that serves the plug-in manifest file.
- Plug-in Auxiliary Server – A plug-in process that serves some plug-in resources, but not the manifest file.

Visibility of Remote Plug-in Views

In an Enhanced Linked Mode (ELM) environment, the vSphere Client is capable of loading views from plug-in instances registered with any of the linked vCenter Server instances. For this example, suppose you are working with three linked vCenter Server instances: VC1, VC2, VC3. You register instances of plug-in A with VC1 and VC2, but not with VC3.

When you register plug-in A with VC1, and again with VC2, all three linked vCenter Server instances discover the registrations. Assume that plug-in A has no configured version or environment constraints, so deployment can proceed freely on all three vCenter Server instances. Plug-in views are visible in your vSphere Client only when appropriate, as defined by these rules:

- Global views are always present in the vSphere Client, regardless of which linked instance of vCenter Server you connect to. A plug-in instance selector enables you to choose between plug-in servers. For example, if you connect to VC3 you can choose between the global views of plug-in A instances registered with VC1 and VC2.
- The visibility of an object-specific plug-in view depends on which vCenter Server instance manages the object. When you select an object managed by a given vCenter Server instance, you have access to object-specific views served by the plug-ins registered with that vCenter Server instance. For example, if you connect to VC3 and select an object managed by VC1, the vSphere Client displays the object-specific view served by plug-in A. However, if you connect to VC1 and select an object managed by VC3, the vSphere Client does not display the object-specific view because VC3 has no instance of plug-in A registered.

- Plug-in visibility can also be limited by version differences between the plug-in and your vSphere Client. For example, suppose you connect to VC3 and select an object managed by VC2, but the instance of plug-in A registered with VC2 uses a new method not recognized by the older version of the JavaScript API served by VC3. Your version of the vSphere Client is unable to display the view served by plug-in A.

  Note To avoid run-time issues with incompatible software versions, configure compatibility constraints as described in Specifying Remote Plug-in Compatibility.

Remote Plug-in Multi-Instance Support

The remote plug-in architecture allows for multiple instances of the same remote plug-in to be deployed within an ELM environment. Instances of a remote plug-in provide distinct server processes that can be completely equivalent in function or can implement different versions of the plug-in functionality. The illustrations that follow show characteristics of some possible topologies.

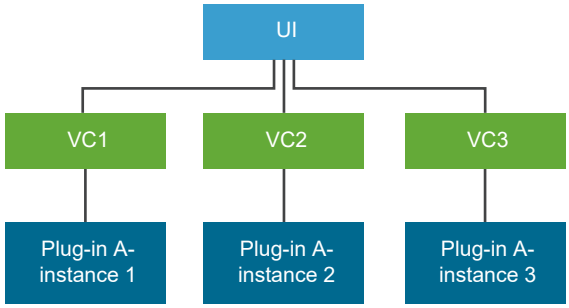
For example, consider an ELM environment with three vCenter Server instances. Suppose that the following extension registration commands are issued:

```
./extension-registration.sh -action registerPlugin -url https://vc1.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver1.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin1_server_thumbprint -remote
```

```
./extension-registration.sh -action registerPlugin -url https://vc2.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver2.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin2_server_thumbprint -remote
```

```
./extension-registration.sh -action registerPlugin -url https://vc3.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver3.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin3_server_thumbprint -remote
```

Figure 4-2. One Plug-in instance per vCenter Server

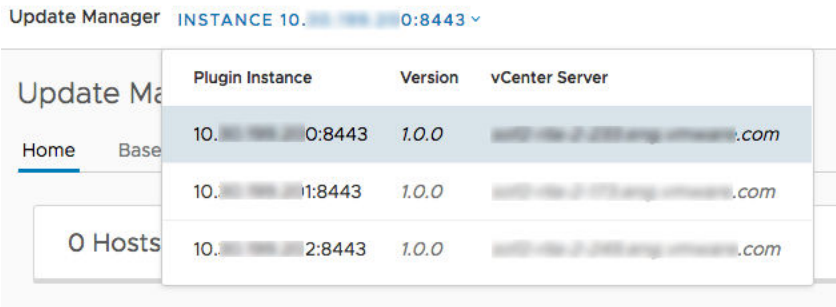


This topology has three instances of PluginA, such that the first plug-in server is registered with VC1, the second is registered with VC2, and the third server is registered with VC3. These plug-in servers are completely equivalent in function: the plug-in manifests that they host and the plug-in specific bits they run are identical. Because each server has a different URL for its manifest file, they are three separate instances of PluginA.

Object views will be loaded from the plug-in instance connected with the vCenter Server instance that the object belongs to. For example, if the plug-in declares a card on the VM summary page, then browsing VMs on VC2 will load the card from the second plug-in server, and browsing VMs on VC3 will load the card from the third plug-in server. Calls to the plug-in back end will be routed to the corresponding plug-in server instance.

Global views, however, will be aggregated in a single global view with an additional instance selector component that allows the user to choose between the global views of the different instances. A sample instance selector is shown below:

Figure 4-3. Selector for Single Plug-in Instance per vCenter Server

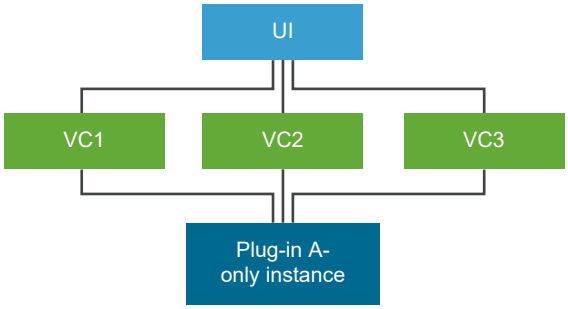


The instance selector shown above displays three back-end servers that represent the three instances of PluginA with version 1.0.0. The **Plugin Instance** column displays the IP address or fully qualified domain name of each plug-in instance, and the **vCenter Server** column displays the vCenter Server that each plug-in instance is connected to. Switching between the items in this drop-down will load the global view of the remote plug-in with version "1.0.0" from the specified plug-in instance.

An alternative topology, which could be considered for a data center with lower access volume, might configure a single shared plug-in instance like this:

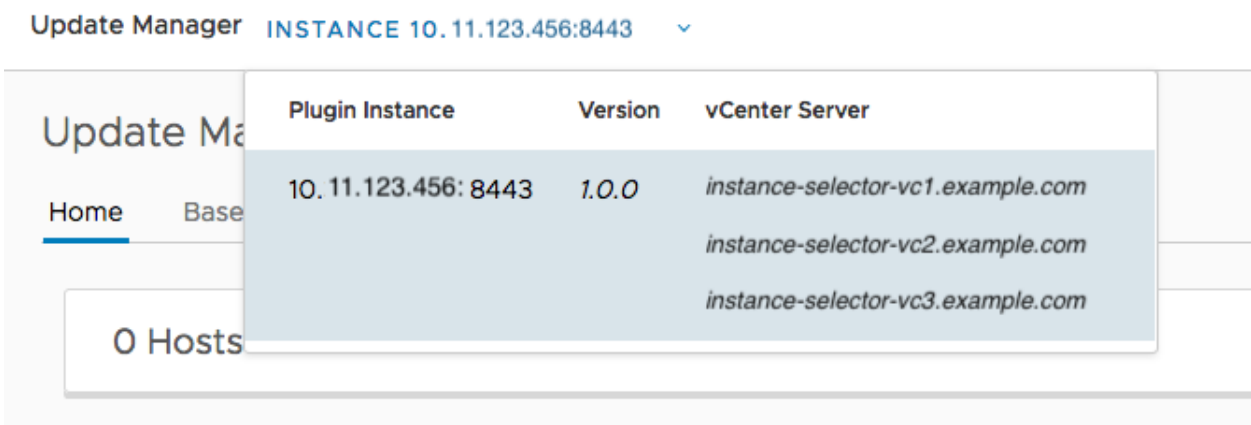
```
./extension-registration.sh -action registerPlugin -url https://vc1.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver1.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin1_server_thumbprint -remote

./extension-registration.sh -action registerPlugin -url https://vc2.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver1.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin1_server_thumbprint -remote

./extension-registration.sh -action registerPlugin -url https://vc3.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver1.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin1_server_thumbprint -remote
```

Figure 4-4. Singleton Plug-in Instance for Link Group



The plug-in is registered with all linked instances of vCenter Server. All registrations have the same plug-in ID, version, and manifest URL, so this is considered a single instance. All plug-in object specific views will be loaded from this one plug-in server. Because no other versions of the same remote plug-in are present in the environment the global view will contain a single item that represents the global view of the singleton instance, as shown in the following example selector:

Figure 4-5. Selector for Singleton Plug-in



With this topology, the plug-in can be upgraded to a newer version for an entire link group by replacing a single process. However, a singleton plug-in works best in a homogenous environment. In an environment that contain sdifferent versions of vCenter Server, the singleton plug-in might need to handle different API versions.

Similarly, if you upgrade an instance of vCenter Server in a singleton plug-in configuration, the plug-in could become incompatible or could fail entirely. For more flexible version support, consider using multiple-instance topologies instead.

Differentiating Plug-in Instances

When the vSphere Client checks the vCenter Server's Extension Manager for new remote plug-ins it also checks for new instances of a remote plug-in. If the vSphere Client finds two extension registrations in two different vCenter Server instances such that the extension ID and extension version are the same but the plug-in manifest URL (ExtensionClientInfo.url) is different, then these two extensions are considered to be different instances of the same remote plug-in. If the plug-in manifest URL is also the same for both registrations, both vCenter Server instances share the same plug-in instance.

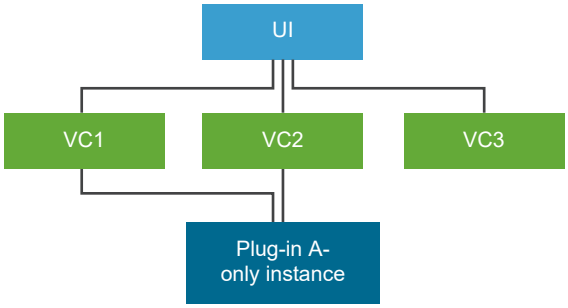
For example, consider an ELM environment with three vCenter Server instances. Suppose that the following extension registration commands are issued:

```
./extension-registration.sh -action registerPlugin -url https://vc1.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver1.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin1_server_thumbprint -remote
```

```
./extension-registration.sh -action registerPlugin -url https://vc2.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver1.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin1_server_thumbprint -remote
```

Both VC1 and VC2 have the same plug-in manifest, and thus the same plug-in server. This is considered to be a single plug-in instance, registered with two vCenter Server instances.

Figure 4-6. Partial Coverage by Singleton Plug-in



In this topology, browsers connected to VC1 or VC2 will see the same object-specific views. In a global view context, a browser connected to any of the linked vCenter Server instances displays a view selector that enables the user to choose a path to the plug-in server. Both VC1 and VC2 provide a proxy route to the same plug-in instance.

Now suppose that we register the following extension in VC3:

```
./extension-registration.sh -action registerPlugin -url https://vc3.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver2.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin2_server_thumbprint -remote
```

The extension registered in VC3 has the same ID and version as the one registered in VC1 and VC2 but has a different manifest URL. When the extension is registered, the vSphere Client will detect that this is a different instance of the remote plugin with ID `com.example.remoteplugin.test`, version `1.0.0`. The UI will show the following behavior:

- Object-specific views declared by plug-in instance 2 (registered in VC3) will be shown for the corresponding objects from VC3. However, the views declared by instance 1 (registered in VC1 and VC2) will be shown for objects from VC1 and VC2.
- For global views, a single entry point in the object navigator displays a plug-in instance view selector where the user will be able to switch between the global views supplied by either of the two plug-in instances.

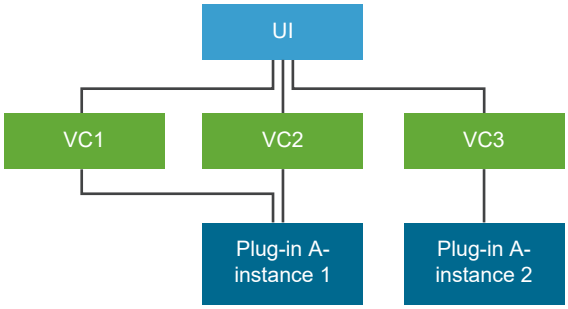
Different plug-in instances can also have different plug-in versions. For example, consider an ELM environment with three vCenter Server instances. Suppose that the following extension registration commands are issued:

```
./extension-registration.sh -action registerPlugin -url https://vc1.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver1.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin1_server_thumbprint -remote
```

```
./extension-registration.sh -action registerPlugin -url https://vc2.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver1.example.com/path/to/plugin.json" -v "1.0.0" -st
plugin1_server_thumbprint -remote

./extension-registration.sh -action registerPlugin -url https://vc3.example.com/sdk -u
"Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'Remote Plugin Test'
-s 'A test plugin demonstrating plugin instances' -k com.example.remoteplugin.test
-pu "https://pluginserver2.example.com/path/to/plugin.json" -v "2.0.0" -st
plugin2_server_thumbprint -remote
```

In this topology, VC1 and VC2 are running with a single instance of the plug-in, at version **1.0.0**, while VC3 is running with a second instance of the plug-in, at version **2.0.0**.

Figure 4-7. Remote Plug-in Version Differentiation



The newer version might have support for new features, such as changes in the API. This capability allows a plug-in to support custom features for some managed objects. It also can help to facilitate testing and rolling upgrades.

Custom Topologies

You can design your plug-in with any topology which may be appropriate for your needs. The following limitations apply:

- At most one remote plug-in instance with a given extension ID can be registered per vCenter Server
- Remote plug-in registrations with different extension IDs are considered different plug-ins.
- Remote plug-in registrations with the same extension ID but different manifest file locations are considered different instances of the same plug-in. In the case of multiple instances, any object-specific view is loaded by the plug-in server instance that is registered with the vCenter Server instance that manages the object. The view makes back-end calls to the server instance that loaded it.

- Remote plug-in registrations with the same extension ID and the same manifest file location are considered a single instance. That instance serves all the vCenter Server instances with which it is registered. Any object-specific view is loaded by the single plug-in server instance that is registered with all the vCenter Server instances.
- Remote plug-ins with the same extension ID and different manifest file locations can also have different versions. In that case, any object-specific view is loaded by the instance that is registered with the vCenter Server instance that manages the object. The details of an object-specific view might differ, depending on the version of the plug-in that is registered with the vCenter Server instance. The view makes back-end calls to the server instance that loaded it.
- If an object is managed by a vCenter Server instance that has no instance of the plug-in registered, then no object-specific view is displayed for that plug-in.
- In case of multiple instances, a single global view will be displayed for the remote plug-in, aggregating the global views of all plug-in instances, and a plug-in view selector allows for switching between the global view content from the different instances.

Deploying Auxiliary Plug-in Servers

When you divide plug-in features between several auxiliary servers, you register each server with a string identifier that indicates the service it provides. Any plug-in server can locate a service it needs by using the service identifier to look up the service endpoint URL in the extension registry record.

When vCenter Server deploys a remote plug-in, it caches the plug-in manifest and configures a routing rule for the reverse HTTP proxy, which directs UI traffic to the manifest server process. If the plug-in includes auxiliary servers, vCenter Server also creates rules to route UI traffic to the auxiliary servers.

VCenter Server maintains a mapping of service identifiers to proxy URLs. Plug-in code that runs in the browser can access that mapping by using the `getPluginBackendInfo()` method. The resulting data allows the JavaScript code to determine the proxy URLs for the services it needs. For more information about `getPluginBackendInfo()`, see vSphere Client JavaScript API: Application Interface.

For more information about registerirg auxiliary plug-in servers, see Registering Auxiliary Plug-in Servers.

Remote Plug-in Deployment Example with Simultaneous Users

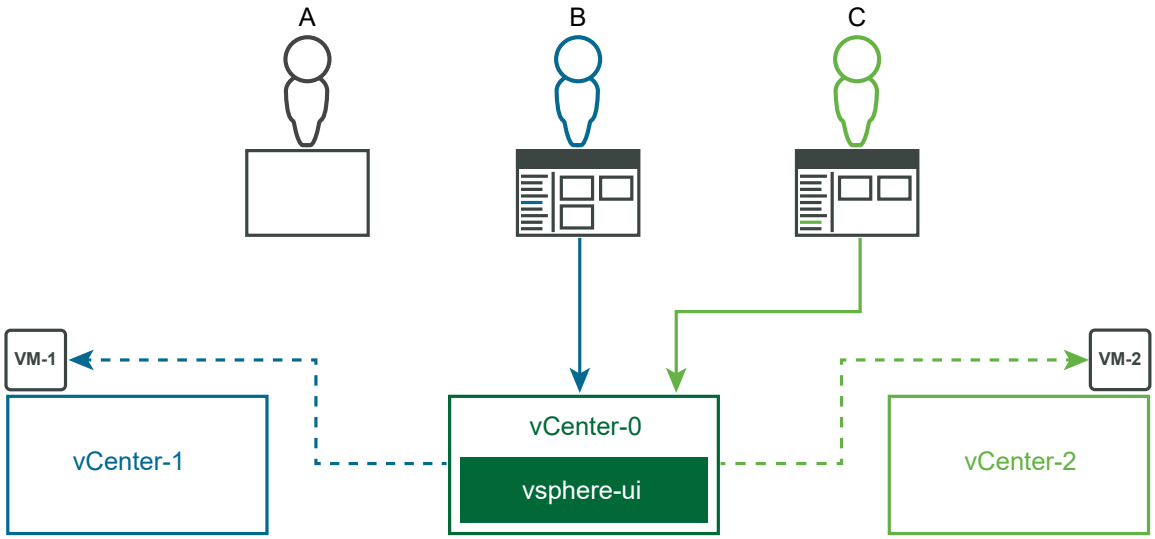
After being detected, a remote plug-in will be scheduled for deployment in the vSphere Client. The deployment of a remote plug-in, on a high level, consists of the following stages:

- vCenter Server downloads the remote plug-in manifest.
- vCenter Server parses the plug-in manifest.
- vCenter Server configures the VMware reverse HTTP proxy to route plug-in UI traffic.

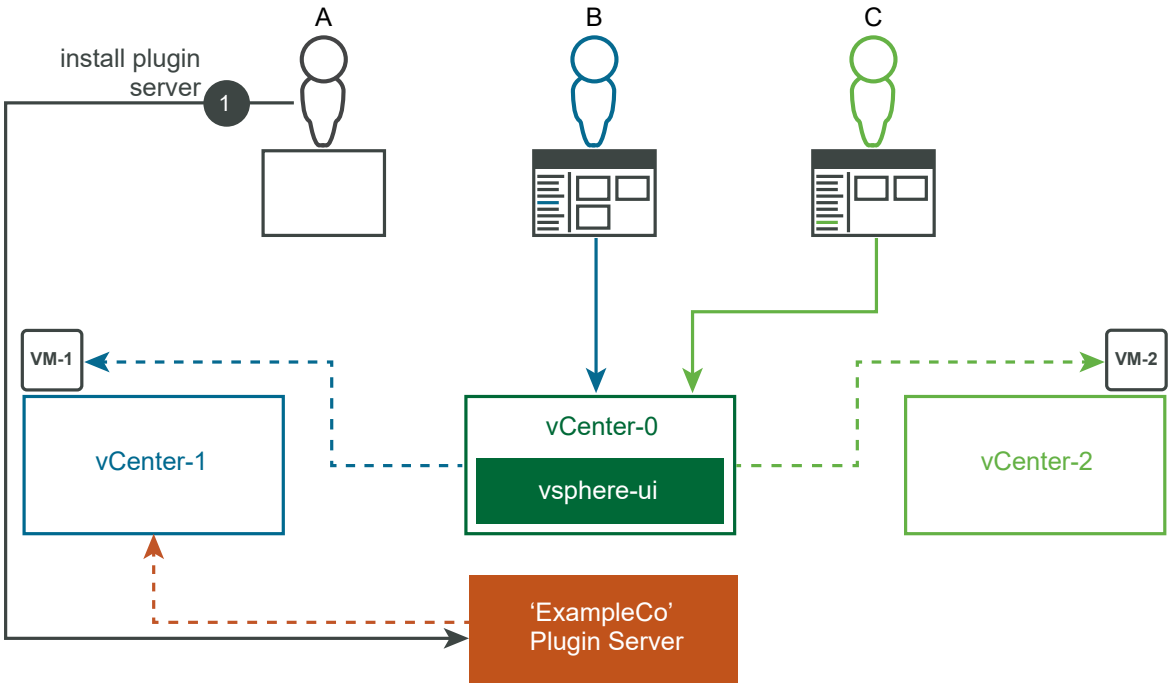
After these stages complete successfully, the vSphere Client UI displays a notification message that the remote plug-in is installed.

This example shows in more detail how the deployment process works, in a situation involving three users simultaneously accessing the data center. The initial state consists of the following:

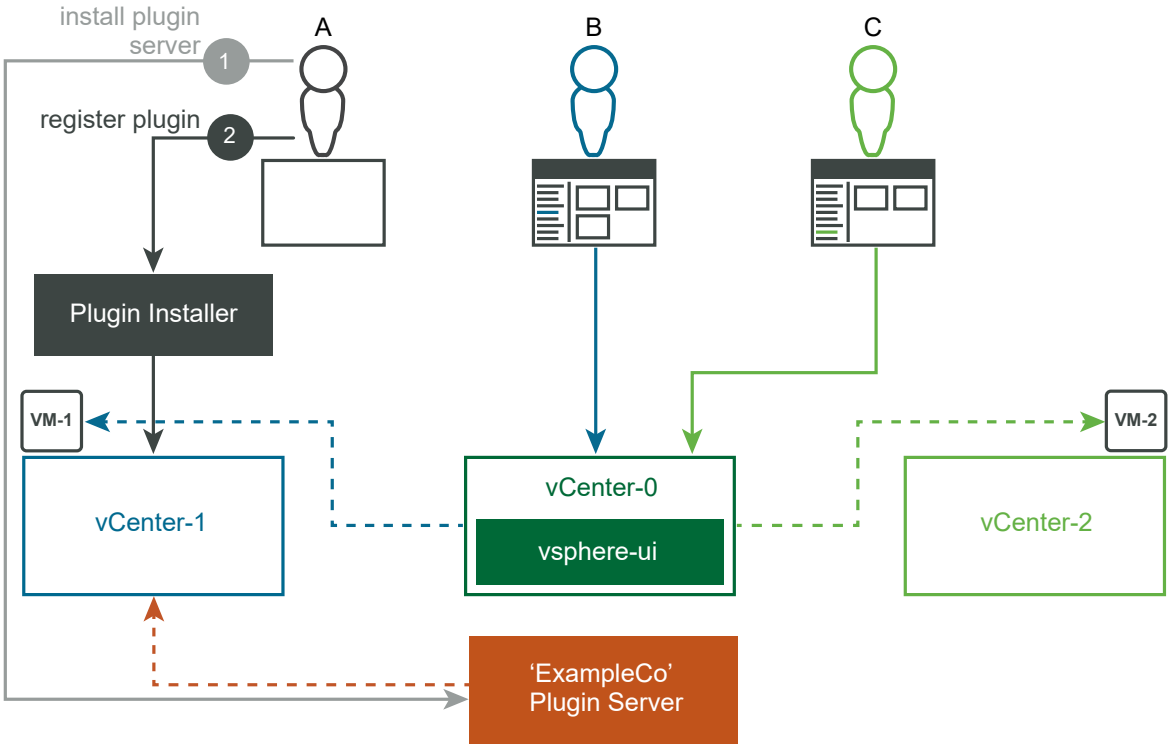
- Three vCenter Server instances in an ELM environment: vCENTER-0, vCENTER-1, and vCENTER-2.
- Three users are accessing the data center: Alpha, Blue, and Claire.
- Blue and Claire are already browsing the vSphere UI loaded from `vCENTER-0`
  - Blue is looking at the summary page of `VM-1` managed by `vCENTER-1`.
  - Claire is looking at the summary page of `VM-2` managed by `vCENTER-2`.
- Alpha is about to install a plug-in from Example Company:



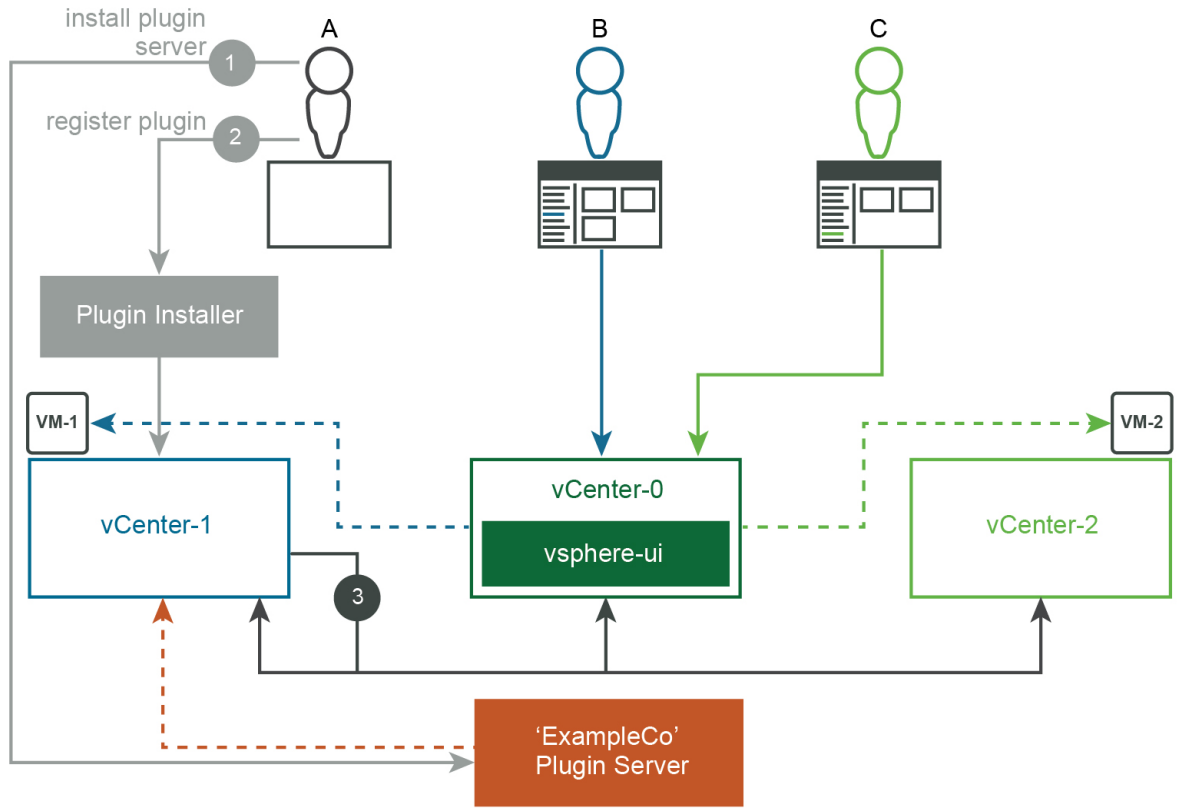
1 Alpha installs and configures the back-end server for the `ExampleCo` plug-in:



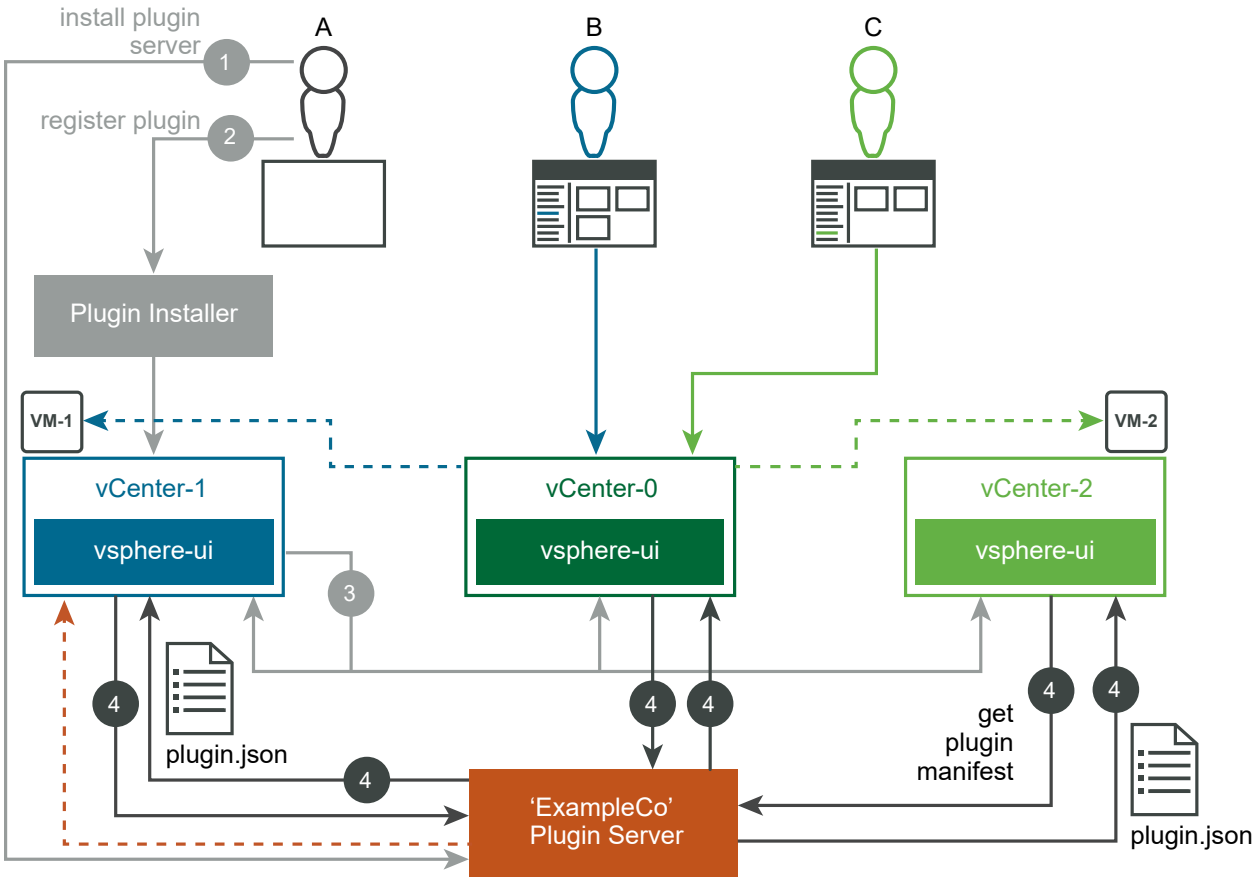
2 Alpha registers the `ExampleCo` plug-in with the `vCENTER-1` ExtensionManager by using Example Company's plug-in installer:



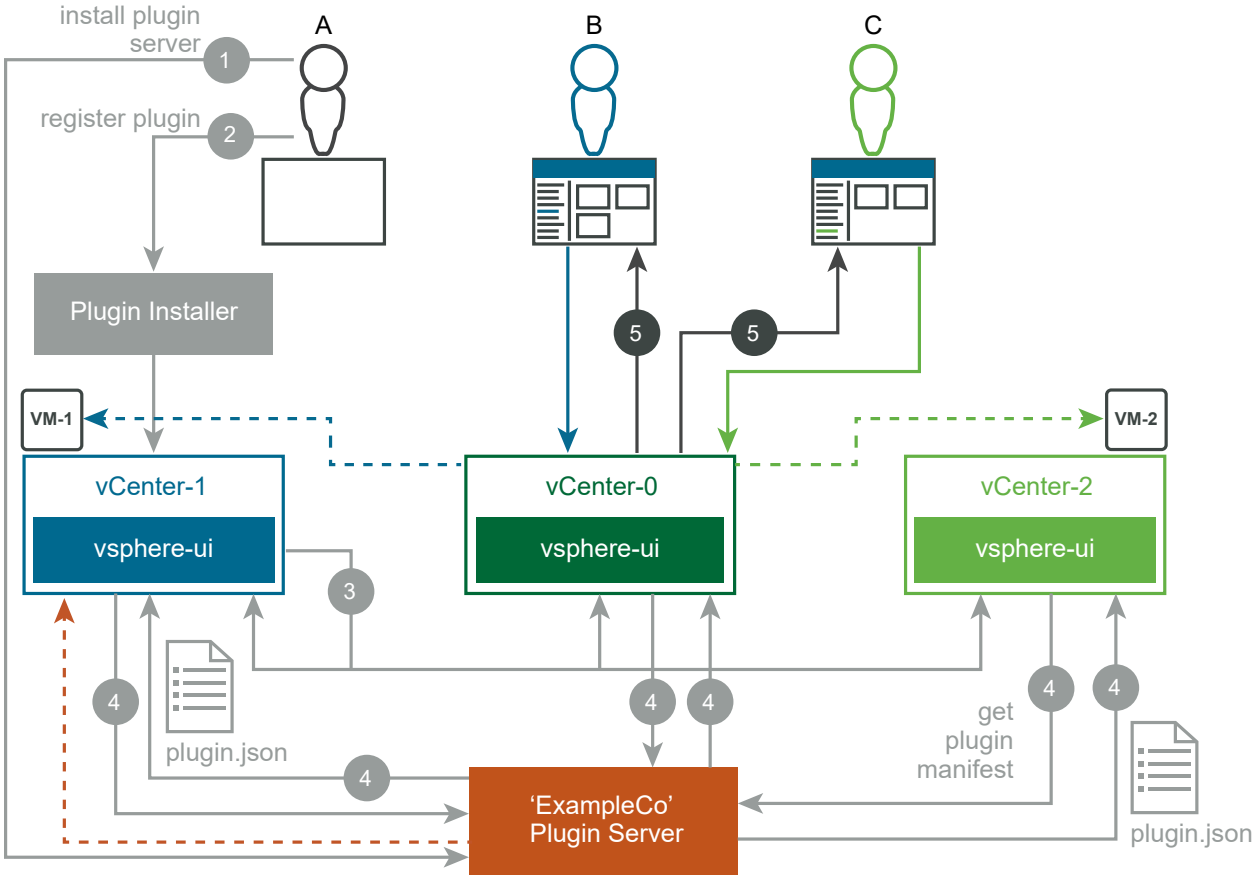
3 The plug-in registration triggers notifications to all linked vCenter Server instances:



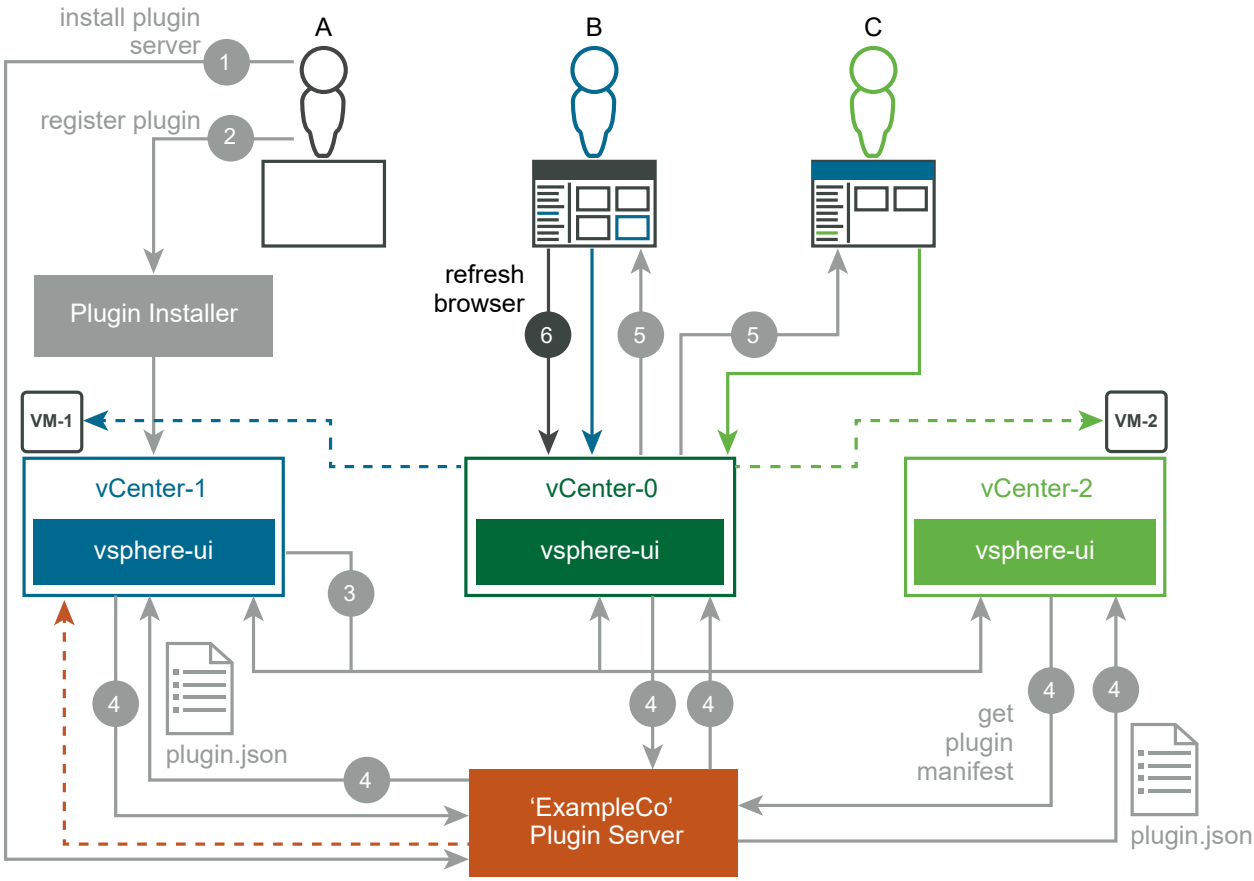
4 The `vsphere-ui` service in each vCenter Server instance downloads the plug-in manifest JSON of the `ExampleCo` plug-in from the plug-in manifest URL in the extension registration record:



5 The vsphere-ui service in `vCENTER-0` sends notifications to currently logged in vSphere Client users (Blue and Claire). Each user sees a blue notification banner at the top of the screen:

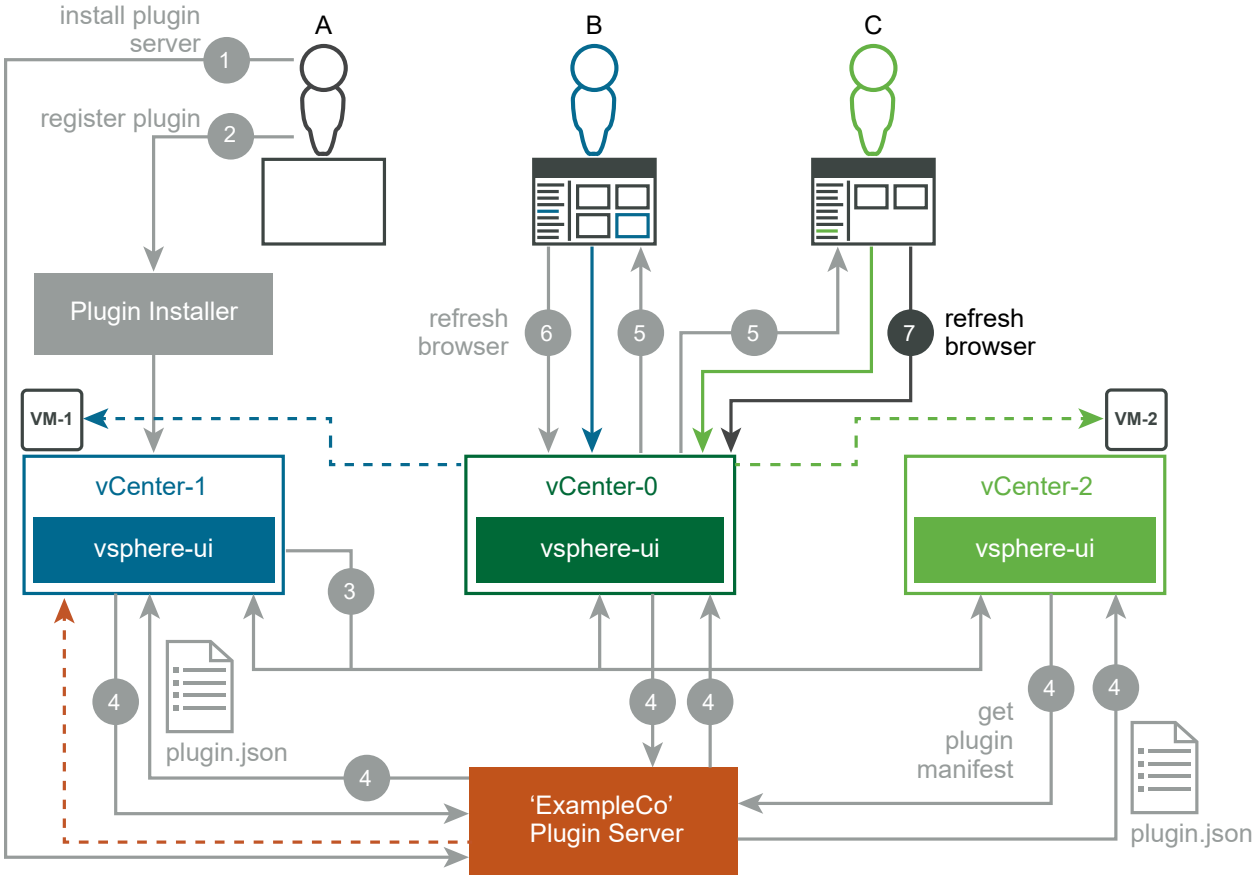


6 When Blue refreshes the vSphere Client UI in the browser, the `ExampleCo` plug-in is loaded for this user. The plug-in adds a card that extends the summary page of VMs. Because `VM-1` is managed by `vCENTER-1`, which has the `ExampleCo` plug-in registered, Blue sees the newly added card:

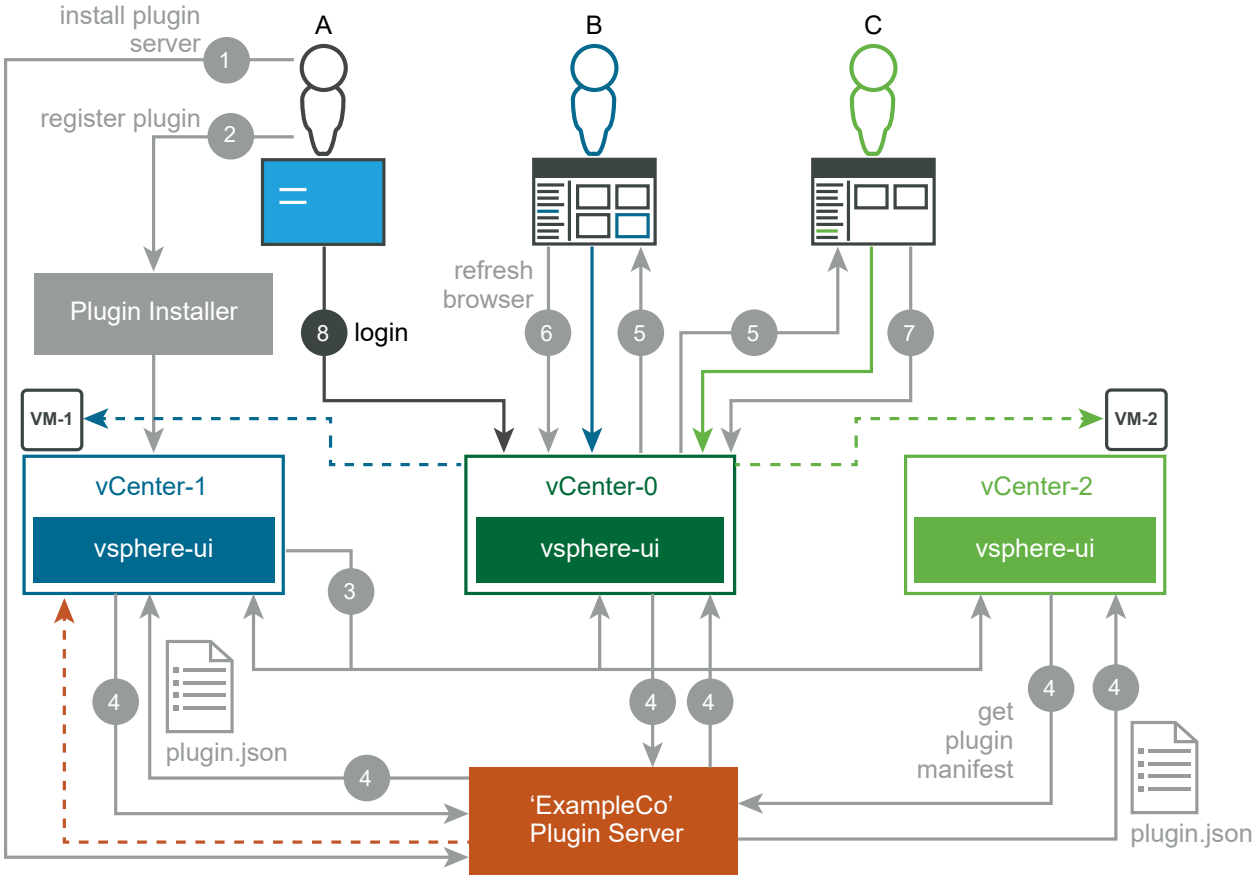


7 When Claire refreshes the vSphere Client UI in the browser, the `ExampleCo` plug-in can now be loaded for this user. However, Claire is looking at `VM-2`, which is managed by `vCENTER-2`. Because `vCENTER-2` does not have the `ExampleCo` plug-in registered, Claire does not see the newly added card.

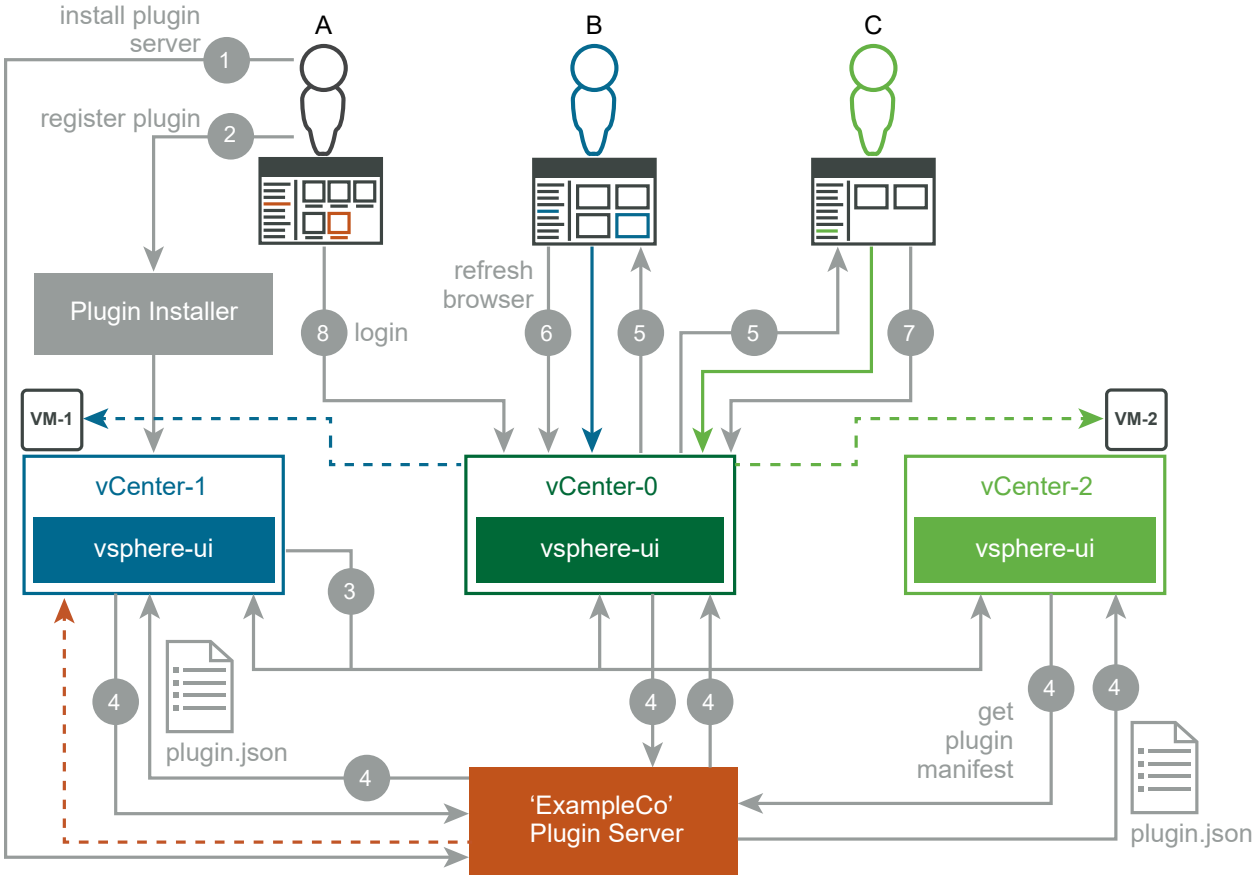
If Claire later navigates to a VM on `vCENTER-1`, the vSphere Client will display the card added by the `ExampleCo` plug-in.



8 Alpha logs in to the vSphere UI connected to `VCENTER-0`:



9 Alpha has completed the login and has loaded the vSphere Client UI, which displays the home screen of the vSphere Client. `VCENTER-0` deployed the `ExampleCo` plug-in prior to the login, so Alpha sees the new home page menu item and the shortcut link immediately:



Remote Plug-in Multi-Version Support

Unlike local plug-ins, the remote plug-in architecture allows for co-existence of remote plug-ins with the same ID but different versions.

Consider an ELM environment with three vCenter Server instances and the following plug-in registrations:

```
./extension-registration.sh -action registerPlugin -url https://vcenter-ip-or-fqdn-of-vc1/sdk -u "Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'ExampleCo' -s 'A test plugin demonstrating plugin instances' -k com.example.exampleco -pu "https://my-remote-plugin-server-version-1/path-to/plugin.json" -v "1.0.0" -st plugin_server_1_thumbprint -remote
```

```
./extension-registration.sh -action registerPlugin -url https://vcenter-ip-or-fqdn-of-vc2/sdk -u "Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'ExampleCo' -s 'A test plugin demonstrating plugin instances' -k com.example.exampleco -pu "https://my-remote-plugin-server-version-2/path-to/plugin.json" -v "2.0.1" -st plugin_server_2_thumbprint -remote
```

```
./extension-registration.sh -action registerPlugin -url https://vcenter-ip-or-fqdn-of-vc3/sdk -u "Administrator@vsphere.local" -p 'Admin!23' -c 'Example, Inc.' -n 'ExampleCo' -s 'A test plugin demonstrating plugin instances' -k com.example.exampleco -pu "https://my-remote-plugin-server-version-3/path-to/plugin.json" -v "3.2.0" -st plugin_server_3_thumbprint -remote
```

These commands register three extensions (one in each of the three vCenter Servers) with the same ID (`com.example.exampleco`) but different versions - version `1.0.0` on VC1, version `2.0.1` on VC2 and version `3.2.0` on VC3. These are three different versions of the remote plug-in with ID `com.example.exampleco`. When you log in to the UI you will see the following:

- Object views declared by plug-in `com.example.exampleco` version `1.0.0` will be shown on applicable objects from VC1. Calls to the plug-in back-end server will be routed to the plug-in server dedicated to version `1.0.0` of the plugin.
- Object views declared by plug-in `com.example.exampleco` version `2.0.1` will be shown on applicable objects from VC2. Calls to the plug-in back-end server will be routed to the plug-in server dedicated to version `2.0.1` of the plug-in.
- Object views declared by plug-in `com.example.exampleco` version `3.2.0` will be shown on applicable objects from VC3. Calls to the plug-in back-end server will be routed to the plug-in server dedicated to version `3.2.0` of the plug-in.
- There will be a single entry point in the object navigator that will take the user to a plug-in instance/version selector view where the user will be able to switch between the global views of the different versions and instances of the remote plug-in.

Remote Plug-in Multi-Manifest Support

You can create a remote plug-in capable of supporting different vSphere Client feature sets. To deploy this plug-in, you specify different manifest files to match the feature sets. The vSphere Client will choose a manifest compatible with the schema it supports.

You should use the multi-manifest feature when you want a single plug-in version to support different feature sets in the vSphere Client or different capabilities in the JavaScript API. The plug-in user interface code can check the existence of API methods or invoke `app.getClientInfo()` to determine which feature set the vSphere Client makes available.

Before 8.0 U3, it was possible for a multi manifest with a `vsphere.client` and/or `vcenter.server` requirements specification to be parsed and chosen for deployment, despite being incompatible with vSphere Client and/or vCenter Server requirements. This caused the plug-in to appear as Incompatible.

As of vSphere 8.0 U3, plug-in environment requirements are processed as part of plug-in manifest parsing and if they are not satisfied, parsing of the current file fails, causing the platform to try the subsequent plug-in manifest file in the queue.

Start by creating two or more manifest files, each conforming to the schema of a supported feature set. For instance, `plugin-80.json` excludes a feature deprecated in vSphere Client 7.0, while `plugin-70.json` includes that feature. After you test your manifest files separately, create a list of their file names as a JSON object named `manifests` and store it in a file named `plugin-multi-manifest.json`:

```
{
  "manifestVersion": "1.0.0",
  "manifests": [
    "plugin-80.json",
    "plugin-70.json",
    "plugin-67.json"
  ]
}
```

The `manifests` object conforms to a separate multi-manifest schema, such that the manifest parser treats it as an ordered list. The parser tries to validate each manifest file in turn for compatibility with `vsphere-ui`. The first file that conforms to the parser's standard manifest schema is the manifest that the vSphere Client uses for the plug-in.

Then create a zip file containing the `plugin-multi-manifest.json` file and the supported manifest files for different feature sets that the plug-in supports. All the files must be at the root of the zip file, rather than in subdirectories. The zip file can have any name you choose.

Finally, register the plug-in with a vCenter Server instance. Use the name of the zip file as the `--pluginUrl` argument to the registration script or as the `client.url` property in the vCenter Server extension record that your registration tool creates.

Note Multi-manifest support was added in vSphere 7.0 U1. To support versions before 7.0 U1 and a multi-manifest after 7.0 U1, you must add a manifest for all versions before 7.0 U1 in a `plugin.json` and place it inside of the Zip archive. This way, versions before 7.0 U1 use the `plugin.json` whereas versions after 7.0 U1 process the multi-manifest JSON. The first one that is parsed correctly will be used.

Choosing Extension Points for vSphere Client Plug-ins

5

The vSphere Client supports adding content at key extension points in the user interface. A plug-in developer can insert custom views that present objects and functions not provided by vSphere.

The available extension points closely follow the navigation experience in the vSphere Client, which facilitates a clear mapping of a plug-in to the views and workflows it owns. The extension points operate at a high level to allow the developer maximum creative space and flexibility.

Read the following topics next:

- Types of Extension Points in the vSphere Client
- Remote Plug-in Manifest Example

Types of Extension Points in the vSphere Client

The vSphere Client provides a number of integration points that plug-ins can extend. These integration points are also known as extension points, where the developer can define extensions that integrate into the vSphere Client user interface. The extension points are designed to allow maximum flexibility for a plug-in while minimizing the potential impact on other plug-ins running concurrently.

Action and View Extensions

The vSphere Client offers two broad classes of extensions: views and actions. Views give direct access to part of the user interface, where the plug-in can define custom displays and controls. Actions generally trigger messaging and business logic.

Views and actions are specified in the plug-in manifest file because they are anchored to controls in the vSphere Client user interface. Actions are realized in the user interface by means of modal dialogs that the user activates by choosing from the plug-in submenu of the vSphere Client global actions menu. Alternatively, headless actions, without a dialog, can navigate directly to other plug-in views. The user activates a view by navigating to a page or a tab that has a plug-in view extension defined for that context.

When a view extension activates, the plug-in receives an iFrame context within which it is free to operate, given certain restrictions such as not accessing the iFrame parent or other DOM elements outside the iFrame. The plug-in has access to a JavaScript API that it can use to interact with processes outside the iFrame in a managed way. These interactions include retrieval of the reverse proxy endpoints and authenticating the back-end server.

The JavaScript API also provides methods for the plug-in to offer to the user actions that are anchored to controls within the plug-in iFrame. These actions do not appear in the global actions menu, so they are not described in the plug-in manifest file. For more information, see vSphere Client JavaScript API: Modal Interface.

Extension Types

Extensions are separated into groups by the inventory context selection, and sub-groups by the purpose of the user interface element. The extension points are grouped as follows:

- **Global extension points**

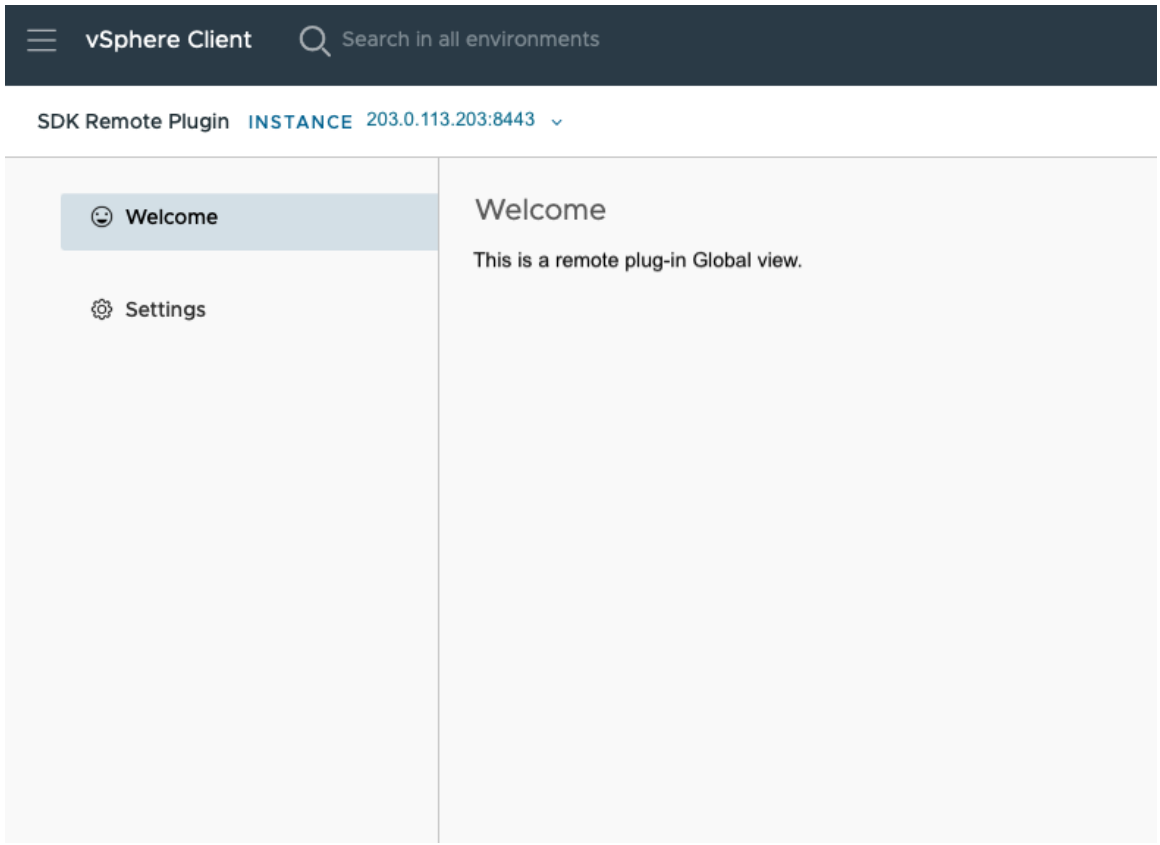
  These are plug-in UI elements that have global scope. Their context is the entire plug-in, rather than a particular inventory object.

  - **View**

    This is a single global UI element that can consume a large section of the vSphere Client real estate. When multiple global pages are required, they should be implemented within this single global view. Navigation between the nested pages must be handled by the plug-in front end.

    For example, this includes configuration pages or status dashboards. The following illustration shows the screen context for a global view.

Figure 5-1. Global View Extension



- **Object extension points**

  These are plug-in UI elements within the scope of the currently selected context object in the inventory. They are defined per object type and displayed only for the selected object. Their views can include other inventory objects or external objects, as long as there is a logical relevance to the current object.

  - **Summary Section View**

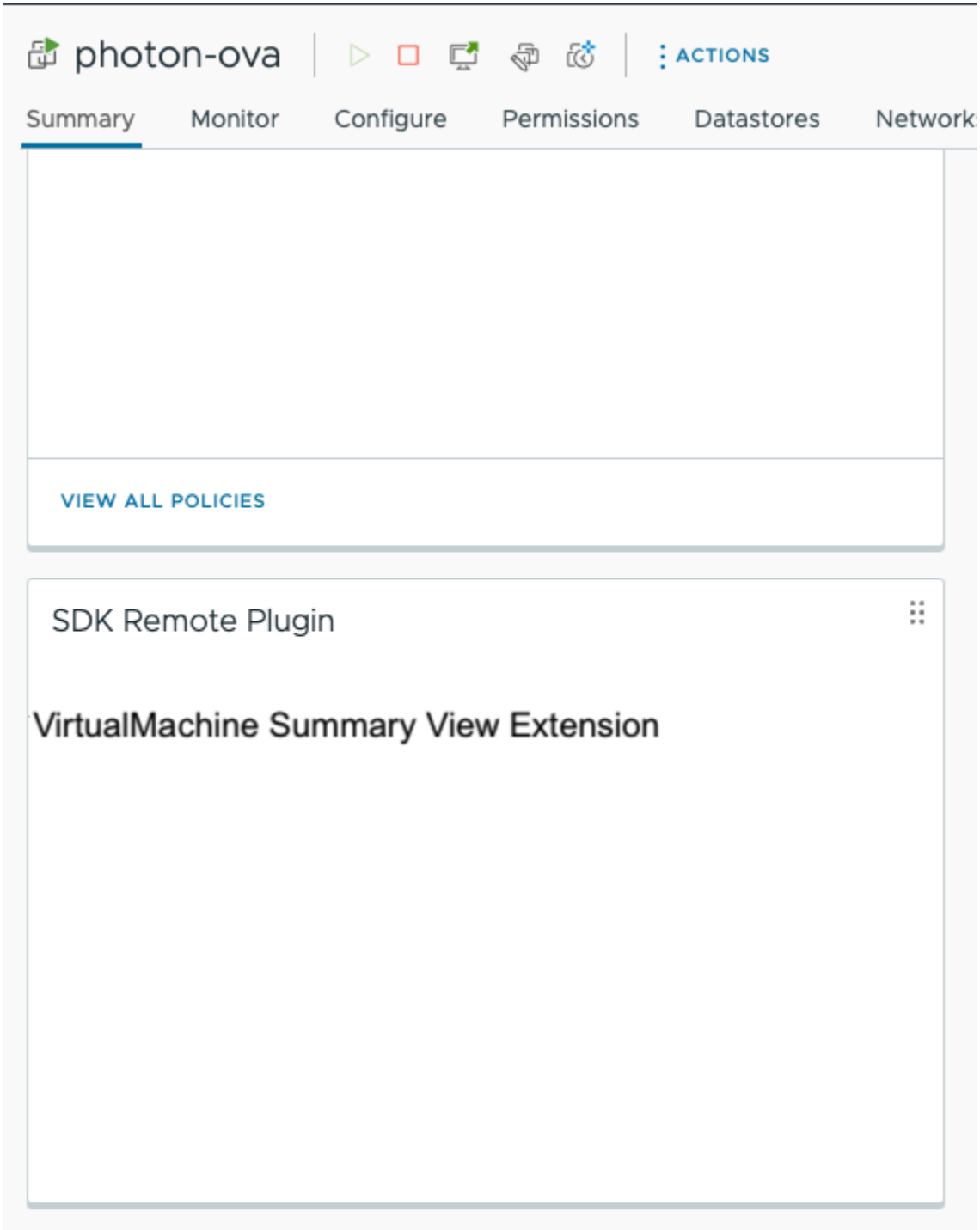
    A Summary section view extension creates a small box in the object's Summary view tab. The box uses the Clarity card template. It should contain primarily simple name-value data. Optionally it can contain action buttons or links to more detailed Monitor or Configure pages.

    For example, this could be a view for a selected host that shows the following:

    - The number of virtual machines that need backup.
    - A button that backs up all virtual machines that need backup.
    - A link to a Monitor view that lists all virtual machines that need backup, and allows a user to back them up separately.

    The following illustration shows the screen context for a VirtualMachine object Summary section view.

Figure 5-2. VirtualMachine Summary Section View Context



Summary section view extensions have fixed height but variable width. If you need more space, you can either select "large" width, create multiple cards in the object's Summary view tab, or link from a Summary view card to a Monitor view that contains more information.

You can create up to five cards in an object's Summary view tab. Each card can have its own title and its own icon. If the title is not specified, it defaults to the plug-in name. If the icon is not specified, it defaults to the plug-in icon.

All cards share a single `dynamicUri`, if used. For more information about `dynamicUri`, see Chapter 6 Dynamic Extensions for Remote Plug-ins.

Specify a single card for the Summary view with the following syntax in the manifest:

```
...
  "summary": {
    "view": {
      "uri": "index.html?view-host-card"
    }
  }
...
```

Specify multiple cards for the Summary view with the following syntax in the manifest. As of vSphere 8.0 U3, card width may be specified as small, medium, or large.

```
...
  "summary": {
    "views": [
      {
        "id": "vmCard1",
        "titleKey": "Title 1",
        "icon": {
          "name": "icon_card1"
        },
        "size": {
          "width": "small"
        },
        "uri": "index.html?view-host-card-card1"
      },
      {
        "id": "vmCard2",
        "titleKey": "Title 2",
        "icon": {
          "name": "icon_card2"
        },
        "size": {
          "width": "large"
        },
        "uri": "index.html?view-host-card-card2"
      }
    ]
  }
...
```

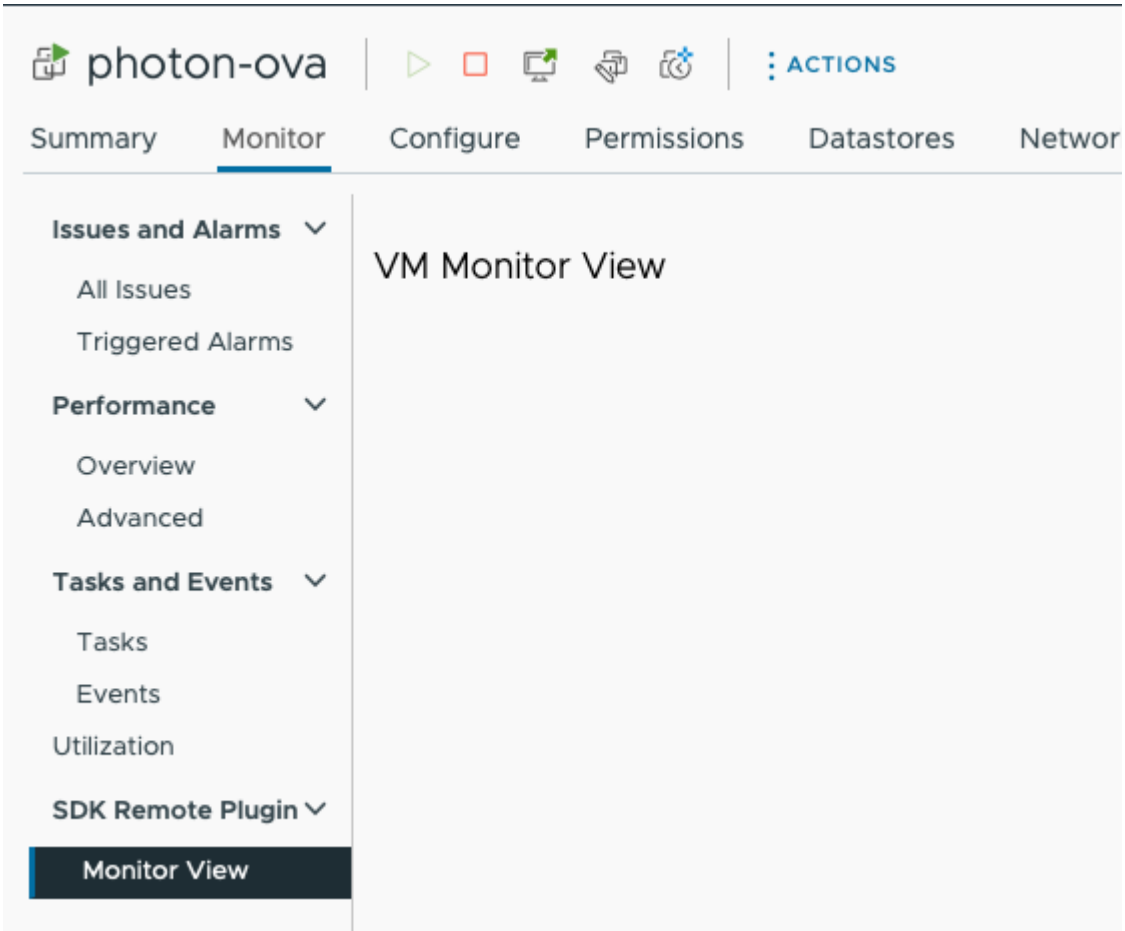
- **Monitor Views**

  This comprises a single Monitor category with one or more views. It can contain detailed monitoring and maintenance data and workflows relevant to the current object.

For example, you could use a Monitor view to show the backup status of all virtual machines on a selected host.

The following illustration shows the screen context for an Object Monitor view

Figure 5-3. Object Monitor View Screen Context



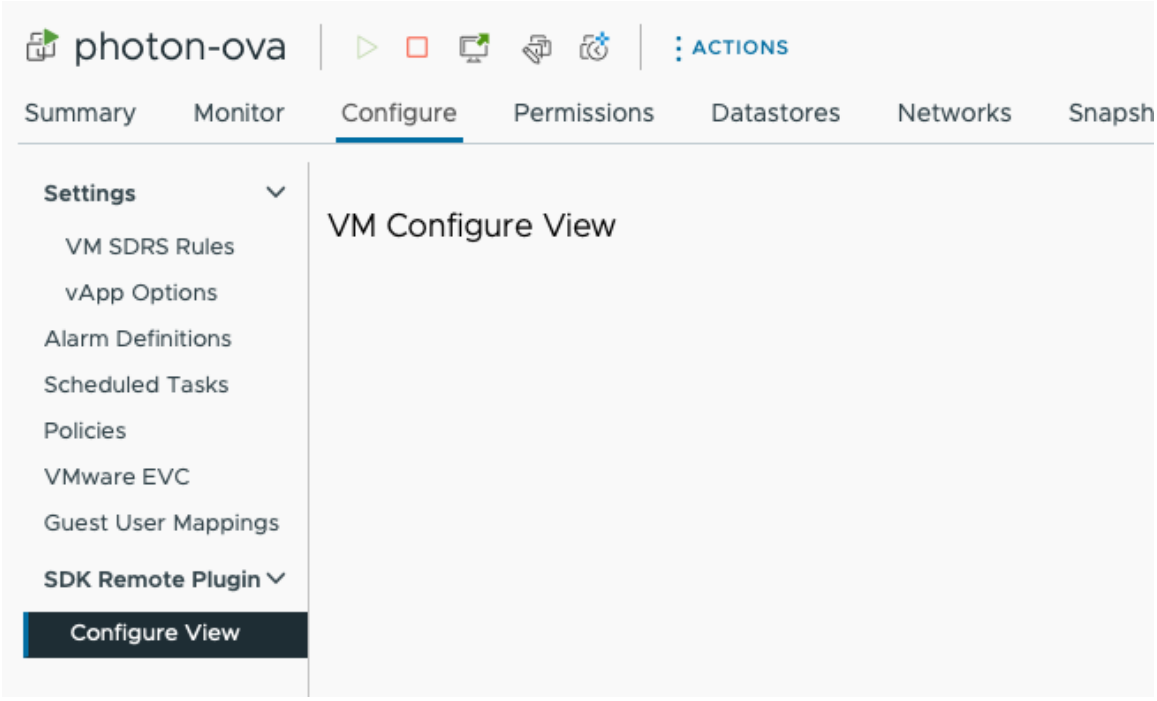
- **Configure Views**

  This comprises a single Configure category with one or more views. It can contain detailed configuration data and workflows relevant to the current object.

  For example, it could show a list of virtual machines on a selected host that should be backed up.

  The following illustration shows the screen context for an Object Configure view

Figure 5-4. Object Configure View Screen Context

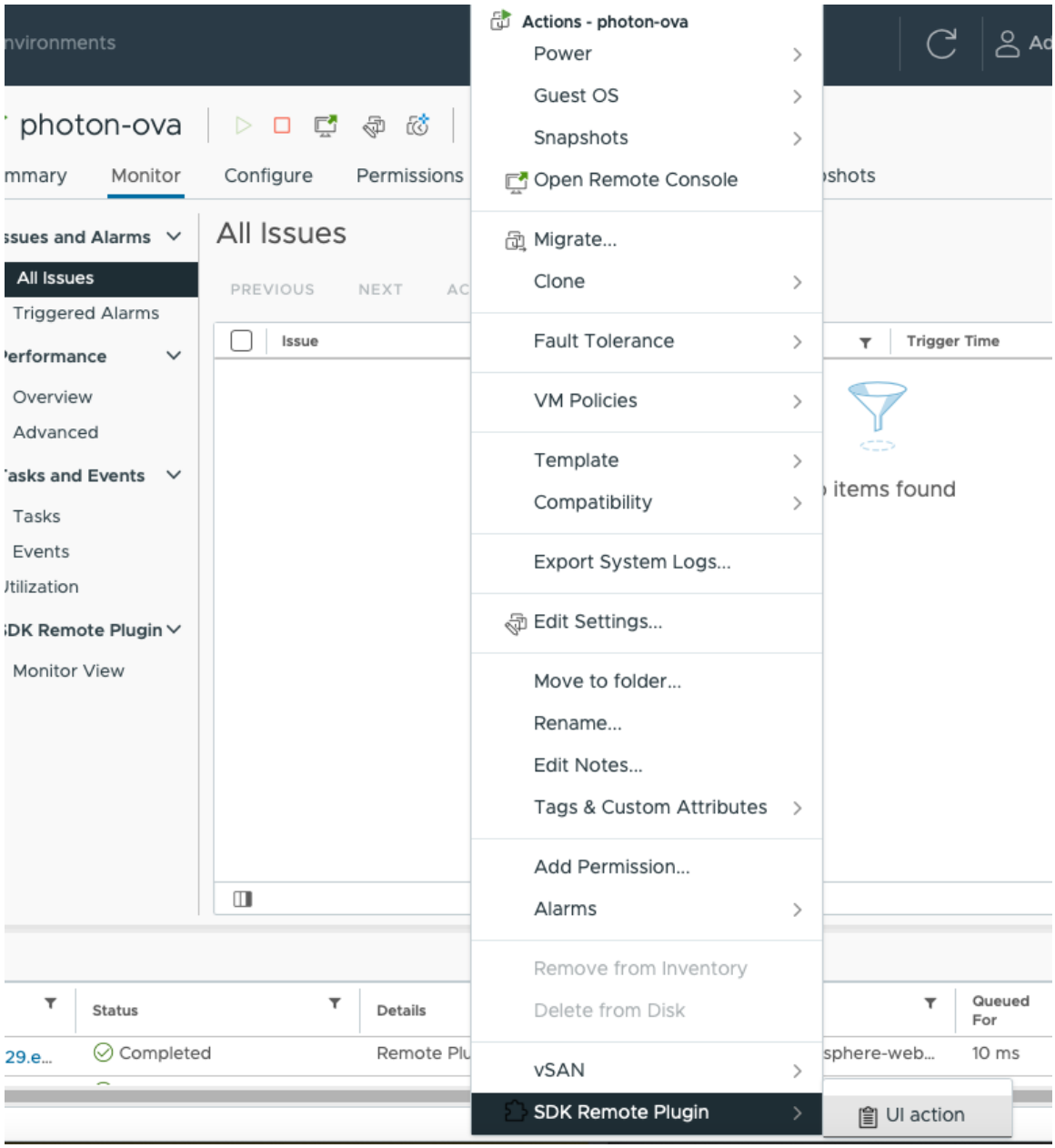


- **Menu**

  This is a single plug-in solution menu with one or more actions. It contains actions that apply to the currently selected object, either navigating directly to a different view of the plug-in or else allowing user input in a modal dialog before the action runs. For example, you could use a Menu action to back up all virtual machines on a selected host.

  The following illustration shows the screen context for a Menu Action.

Figure 5-5. Menu Action Screen Context



A plug-in solution submenu can contain horizontal separators, which allow you to group menu choices. To insert a separator into the submenu, insert the following element into the `menu.actions` array:

```
{ "type": "separator" },
```

Note Separator elements at the beginning or end of the `menu.actions` array are ignored. A pair of adjacent separator elements produces only a single separator displayed on the screen.

Menu actions can also apply to a set of selected objects, under the following conditions:

- All selected objects are managed by the same vCenter Server instance.
- All selected objects have the same type, such as `HostSystem` or `VirtualMachine`.
- The plug-in manifest specifies `acceptsMultipleTargets` in the configuration for the selected object type.

If all of these conditions are true when you select multiple objects, you can apply a menu action to all of the selected objects. If any of the conditions is untrue, for instance, if you select both a `VirtualMachine` object and a `HostSystem` object, the menu shows no actions for the plug-in.

To enable selecting multiple targets, set `acceptsMultipleTargets` to `true` in the menu action configuration for the type or types that should allow it. For example, a configuration for `HostSystem` objects could contain the following lines:

```
...
  "objects": { ...
    "HostSystem": { ...
      "menu": {
        "actions": [
          {
            "id": "TakeAction",
            "labelkey": "Take action",
            "acceptsMultipleTargets": true,
            "trigger": {
              "type": "modal",
            ...
```

Note Headless actions that navigate to a different view for the same object and served by the same remote plug-in are also supported. Specify these headless actions by setting the `trigger.type` to `"navigation"`.

Remote Plug-in Manifest Example

The following JSON code is an example of a plug-in manifest file that demonstrates how to specify some of the vSphere Client SDK extension points for your plug-in.

```
{
  "manifestVersion": "1.0.0",
  "requirements": {
    "plugin.api.version": "1.0.0"
  },
  "configuration": {
    "nameKey": "My Plugin"
  },
  "global": {
    "view": {
```

```
      "navigationId": "myGlobalViewId",
      "uri": "myplugin/globalView.html",
      "navigationVisible": false
    }
  },
  "objects": {
    "Datacenter": {
      "summary": {
        "view": {
          "uri": "myplugin/summary.html",
          "icon": {
            "name": "main"
          }
        }
      },
      "monitor": {
        "views": [
          {
            "navigationId": "myview1",
            "labelKey": "category.view1",
            "uri": "myplugin/view1.html"
          }
        ]
      },
      "configure": {
        "views": [
          {
            "navigationId": "myview1",
            "labelKey": "category.view1",
            "uri": "myplugin/view1.html"
          }
        ]
      },
      "menu": {
        "actions": [
          {
            "labelKey": "action1",
            "icon": {
              "name": "action-1"
            },
            "trigger": {
              "type": "modal",
              "uri": "myplugin/modal-action.html",
              "titleKey": "myplugin modal title",
              "accessibilityTitleKey": "myplugin modal accessibility title",
              "size": {
                "height": 250,
                "width": 600
              }
            }
          },
          {
            "type": "separator"
          },
          {
```

```
            "labelKey": "Action2",
            "icon": {
              "name": "action-2"
            },
            "trigger": {
              "type": "navigation",
              "targetViewId": "myGlobalViewId"
            }
          }
        ]
      }
    }
  },
  "definitions": {
    "iconSpriteSheet": {
      "uri": "myplugin/images/icon-sprite.png",
      "definitions": {
        "main": {
          "x": 0,
          "y": 0
        }
      }
    },
    "i18n": {
      "locales": [
        "en-US",
        "de-DE",
        "fr-FR"
      ],
      "definitions": {
        "category.view1": {
          "en-US": "Monitor View 2",
          "de-DE": "Monitoransicht 2",
          "fr-FR": "Vue Surveiller 2"
        }
      }
    }
  }
}
```

Dynamic Extensions for Remote Plug-ins

6

By default, plug-in views and menu items display in the vSphere Client user interface unconditionally. The vSphere Client supports a Service Provider Interface (SPI) that allows plug-in servers to filter views or menu items so they display conditionally. Conditional extensions are known as dynamic extensions.

Dynamic extensions are supported for Summary, Monitor, and Configure views, and for Menu actions. The vSphere Client SDK does not currently support dynamic extensions for Global views.

Read the following topics next:

Dynamic Extension Use Cases

The vSphere Client supports dynamic extensions for remote plug-ins with an SPI that allows a plug-in server to determine at run time which views display in the vSphere Client user interface. This allows the plug-in to use any basis for the decision.

The dynamic extension SPI gives you flexibility to choose and implement your own filter processing in the plug-in server. For example, a plug-in could choose to display or hide an extension based on user authorization or object state.

How the vSphere Client Displays Dynamic Extensions

The vSphere Client determines whether to show a dynamic extension by asking a plug-in server at the time the context makes the extension relevant. The vSphere Client identifies the relevant extensions and sends a request to each plug-in server that makes visibility choices for the relevant extensions. Each plug-in server collects the information needed to decide whether

to hide or show its extensions, and responds with its visibility choice for each of its relevant extensions.

As of vSphere 8.0 U3, dynamic plug-in actions can be hidden or disabled depending on response from the plug-in server, which can return items with two properties: `visible` or `relevant`.

- Setting the `visible` property to `false` has a different effect on actions than on Cards, Monitor or Configure views. For Cards, Monitor views, or Configure views, `visible=false` means that the view is not shown at all. For actions, `visible=false` makes the action visible but disabled.
- Setting the `relevant` property to `false` makes the action disappear from the menu. The `relevant` property is not required because it is `true` by default. All actions are deemed relevant and get enabled or disabled by the `visible` property. To remove an action, the plug-in should return a response with `relevant=false` and `visible=false`.

Dynamic Views

While the vSphere Client waits for answers, it might display views associated with static extensions, as well as indications that it is waiting for more data. If a plug-in has any dynamic extensions that are relevant in the current context, the vSphere Client will display the plug-in category name in the items list, with a spinner beside it indicating that the display is not yet complete. The vSphere Client will not display the view names in the plug-in category until the list has stabilized.

Dynamic Cards

If a card on the Summary tab is configured to be dynamic, the vSphere Client queries the associated plug-in server for visibility choices. The Summary tab displays only static cards while the vSphere Client waits for a response. After the response arrives, the vSphere Client adds to the display any dynamic cards that the response indicates should be visible.

Dynamic Actions

If a plug-in has dynamic action extensions in the Action menu, the vSphere Client creates place-holders in the menu until the plug-in response indicates whether or not the plug-in actions should be visible. While waiting for a response, the vSphere Client displays the plug-in name in the Actions menu, and puts a brief message in the plug-in submenu that indicates the client code is still loading information. When the plug-in response indicates that an action should be visible, the vSphere Client enables the action in the plug-in submenu. As of 8.0 U3, the plug-in can disable an action with `visible=false` or remove it with `relevant=false` also.

Direct Links to Dynamic Views

If a user clicks a link to a dynamic view, the vSphere Client might need to query the plug-in server for the view's visibility. While waiting for the response, the vSphere Client might open an iFrame for the dynamic view, with only a spinner and a message that the view is not yet ready. If the response indicates that the dynamic view should be hidden, the vSphere Client displays a message saying that the view is not available.

Direct Links to Static Views

If a user clicks a link to a static view, the vSphere Client displays the static view immediately. If there are also dynamic views that might apply, the client code marks the plug-in name in the category list with a spinner while it queries the plug-in server for its visibility choices.

Time-outs

In case the plug-in server does not return a timely response, the vSphere Client will time out and cancel the request. When the vSphere Client times out a request for dynamic extension visibility, it assumes the extension should be hidden.

Caching Dynamic Extension Visibility

When the user navigates to a context (such as a **Monitor** tab) that has the potential to show dynamic extensions, the client code queries the `dynamicUri` for filter choices. The plug-in server returns its visibility choices for the dynamic extensions that apply to the user context. The client code caches the dynamic view choices to speed navigation between the views that pertain to the active vSphere object.

The client code caches visibility choices for only those dynamic views that apply to the current object. For example, if the user navigates to a virtual machine's **Monitor** tab and the plug-in server returns visibility choices for both a view on the **Monitor** tab and a view on the **Configure** tab that apply to the current virtual machine, the client code caches choices for both tabs. The client code does not cache visibility choices for dynamic actions.

The visibility choices for the current object generally remain in cache as long as the user navigates only to tabs for that object. The client code empties the cache when the user navigates to a different vSphere object. Then the client builds up the cache for dynamic views that pertain to the new object.

Note The global refresh button clears the cache and queries the applicable `dynamicUri` for filter choices.

Configure Dynamic Extensions

You configure dynamic extensions for your plug-in by using properties in the `plugin.json` manifest file. These properties identify the dynamic extensions and provide SPI endpoints for queries from the client code.

Dynamic Monitor and Configure extensions are configured in the same way. Dynamic Summary extensions and dynamic Menu extensions are similar. Where the steps differ, the examples show the differences.

Procedure

1 Edit the manifest file, `plugin.json`.

```
{ "manifestVersion": "1.0.0",
  "requirements": {"plugin.api.version": "1.0.0"}
  ...
```

2 Within the `objects` object, locate the type of vSphere managed object for which the plug-in can supply a dynamic extension.

```
{ "manifestVersion": "1.0.0",
  "requirements": {"plugin.api.version": "1.0.0"}
  ...
  "objects": { ...
    "VirtualMachine": {
      ...
```

3 Within the managed object type, locate the extension type for which the plug-in can supply a dynamic extension.

Option	Description

For a dynamic extension in the Summary tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "summary": {`
	`  ...`

For a dynamic extension in the Monitor tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "monitor": {`
	`  ...`

For a dynamic extension in the Configure tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "configure": {`
	`  ...`

For a dynamic extension in the Actions menu.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "menu": {`
	`  ...`

4 Within the object representing the extension type object, add a `dynamicUri` property to identify the endpoint where the client code will query the plug-in server for its filter choices.

If the plug-in manifest server will supply filter choices, you can specify a string containing the URI path. If an auxiliary server will supply filter choices, specify a JSON object containing a `path` property (type `string`) and a `serverType` property (type `string`), where the `serverType` corresponds to the `ServerInfo.type` property in the Extension Manager registration record. The `serverType` value maps to the auxiliary server base URI that the user interface code uses to construct the endpoint for the dynamic extension filter query.

Note When you register an auxiliary server that will supply filter choices, its `ServerInfo.url` property in the `Extension` record must end with a slash (`/`). See Registering Auxiliary Plug-in Servers.

Option	Description

For a dynamic extension in the Summary tab.	`...`
	`"objects": { ...`
	`"VirtualMachine": { ...`
	`"summary": {`
	`"dynamicUri": "rest/dynamics/vm/summary",`
	`...`

(if an auxiliary server handles filter queries)	`...`
	`"objects": { ...`
	`"VirtualMachine": { ...`
	`"summary": {`
	`"dynamicUri": {`
	`"serverType": "DYNAMIC_AUX_SERVER",`
	`"path": "rest/dynamics/vm/summary"`
	`}, ...`

For a dynamic extension in the Monitor tab.	`...`
	`"objects": { ...`
	`"VirtualMachine": { ...`
	`"monitor": {`
	`"dynamicUri": "rest/dynamics/vm/monitor",`
	`...`

(if an auxiliary server handles filter queries)	`...`
	`"objects": { ...`
	`"VirtualMachine": { ...`
	`"monitor": {`
	`"dynamicUri": {`
	`"serverType": "DYNAMIC_AUX_SERVER",`
	`"path": "rest/dynamics/vm/monitor"`
	`},`
	`...`

Option	Description

For a dynamic extension in the Configure tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "configure": {`
	`  "dynamicUri": "rest/dynamics/vm/configure",`
	`  ...`

(if an auxiliary server handles filter queries)	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "configure": {`
	`  "dynamicUri": {`
	`  "serverType": "DYNAMIC_AUX_SERVER",`
	`  "path": "rest/dynamics/vm/configure"`
	`  },`
	`  ...`

For a dynamic extension in the Actions menu.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "menu": {`
	`  "dynamicUri": "rest/dynamics/vm/actions",`
	`  ...`

(if an auxiliary server handles filter queries)	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "menu": {`
	`  "dynamicUri": {`
	`  "serverType": "DYNAMIC_AUX_SERVER",`
	`  "path": "rest/dynamics/vm/actions"`
	`  },`
	`  ...`

For more information about the `ServerInfo.type` property, see Registering Auxiliary Plug-in Servers.

5 At the same level as the `dynamicUri` property, locate the `views` array or the `view` object (for Summary extensions), the `views` array (for Monitor or Configure extensions) or the `actions` array (for Menu extensions)

Option	Description

For a single dynamic extension in the Summary tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "summary": {`
	`  "dynamicUri": "rest/dynamics/vm/summary",`
	`  "view": {`
	`  ...`

For multiple dynamic extensions in the Summary tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "summary": {`
	`  "dynamicUri": "rest/dynamics/vm/summary",`
	`  "views": [`
	`  ...`
	`  ]`
	`  }`
	`  ...`

For a dynamic extension in the Monitor tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "monitor": {`
	`  "dynamicUri": "rest/dynamics/vm/monitor",`
	`  "views": [`
	`  ...`

For a dynamic extension in the Configure tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "configure": {`
	`  "dynamicUri": "rest/dynamics/vm/configure",`
	`  "views": [`
	`  ...`

For a dynamic extension in the Actions menu.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "menu": {`
	`  "dynamicUri": "rest/dynamics/vm/actions",`
	`  "actions": [`
	`  ...`

6 To each view or action that will be treated as dynamic, add the `dynamic` property.

Option	Description

For a single dynamic extension in the Summary tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "summary": {`
	`  "dynamicUri": "rest/dynamics/vm/summary",`
	`  "view": {`
	`  "dynamic": true`
	`  ...`

For multiple dynamic extensions in the Summary tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "summary": {`
	`  "dynamicUri": "rest/dynamics/vm/summary",`
	`  "views": [`
	`  {`
	`  "dynamic": true,`
	`  ...`
	`  },`
	`  {`
	`  "dynamic": true,`
	`  ...`
	`  }`
	`  ]`
	`  }`
	`  ...`

For a dynamic extension in the Monitor tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "monitor": {`
	`  "dynamicUri": "rest/dynamics/vm/monitor",`
	`  "views": [`
	`  {`
	`  "dynamic": true,`
	`  ...`

Option	Description

For a dynamic extension in the Configure tab.	```
	...
	  "objects": { ...
	    "VirtualMachine": { ...
	      "configure": {
	        "dynamicUri": "rest/dynamics/vm/configure",
	        "views": [
	          {
	            "dynamic": true,
	            ...
	``

For a dynamic extension in the Actions menu.	```
	...
	  "objects": { ...
	    "VirtualMachine": { ...
	      "menu": {
	        "dynamicUri": "rest/dynamics/vm/actions",
	        "actions": [
	          {
	            "dynamic": true,
	            ...
	``

7 Add an `id` property to each Summary card or Menu action, and a `navigationId` to each Monitor or Configure view, if not already present.

Option	Description

For a single dynamic extension in the Summary tab.	```
	...
	  "objects": { ...
	    "VirtualMachine": { ...
	      "summary": {
	        "dynamicUri": "rest/dynamics/vm/summary",
	        "view": {
	          "dynamic": true
	          "id": "vmSummaryCard",
	          ...
	``

For multiple dynamic extensions in the Summary tab.	```
	...
	  "objects": { ...
	    "VirtualMachine": { ...
	      "summary": {
	        "dynamicUri": "rest/dynamics/vm/summary",
	         "views": [
	           {
	             "dynamic": true,
	             "id": "summary-card-1",
	             ...
	           },
	           {
	             "dynamic": true,
	             "id": "summary-card-2",
	             ...
	           }
	         ]
	      }
	      ...
	``

Option	Description

For a dynamic extension in the Monitor tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "monitor": {`
	`  "dynamicUri": "rest/dynamics/vm/monitor",`
	`  "views": [`
	`  {`
	`  "dynamic": true,`
	`  "navigationId": "MonitorDynView",`
	`  ...`

For a dynamic extension in the Configure tab.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "configure": {`
	`  "dynamicUri": "rest/dynamics/vm/configure",`
	`  "views": [`
	`  {`
	`  "dynamic": true,`
	`  "navigationId": "ConfigureDynView",`
	`  ...`

For a dynamic extension in the Actions menu.	`...`
	`  "objects": { ...`
	`  "VirtualMachine": { ...`
	`  "menu": {`
	`  "dynamicUri": "rest/dynamics/vm/actions",`
	`  "actions": [`
	`  {`
	`  "dynamic": true,`
	`  "id": "TakeAction",`
	`  ...`

A `navigationId` or `id` string is required to identify the dynamic view or action in the plug-in server's response to a filter query.

Results

The following example shows a `plugin.json` file with a dynamic Summary card, a dynamic Monitor view, a fixed Configure view, and a dynamic action. The dynamic Summary view is filtered by the manifest server, which is the default when a specific server is not named. The dynamic Monitor view is filtered by the the manifest server, configured with an explicit reference. The dynamic menu action is filtered by an auxiliary server.

Example: plugin.json With Dynamic Extensions

```
{ "manifestVersion": "1.0.0",
  "requirements": {"plugin.api.version": "1.0.0"},
  "configuration": {
    "nameKey": "Dynamic Extension Manifest Example"
  }
  "objects": {
    "VirtualMachine": {
      "summary": {
```

```
          "dynamicUri": "rest/dynamics/vm/summary",
          "view": {
            "dynamic": true
            "id": "vmSummaryCard",
            "uri": "index.html#vm-portlet",
          }
        },
        "monitor": {
          "dynamicUri": {
            "serverType": "MANIFEST_SERVER",
            "path": "rest/dynamics/vm/monitor"
          },
          "views": [
            {
              "dynamic": true,
              "navigationId": "MonitorDynView",
              "uri": "rest/views/vm/monitor/index.html"
            }
          ],
        "configure": {
          "views": [
            {
              "navigationId": "ConfigureFixView",
              "uri": "rest/views/vm/configure/index.html"
            },
        "menu": {
          "dynamicUri": {
            "serverType": "DYNAMIC_AUX_SERVER",
            "path": "rest/dynamics/vm/actions"
          },
          "actions": [
            {
              "dynamic": true,
              "id": "TakeAction",
              "labelKey": "Take action",
              "trigger": {
                "type": "modal",
                "uri": "rest/actions/vm/action1.html"
              }
            }
          ]
        }
      }
    }
  }
```

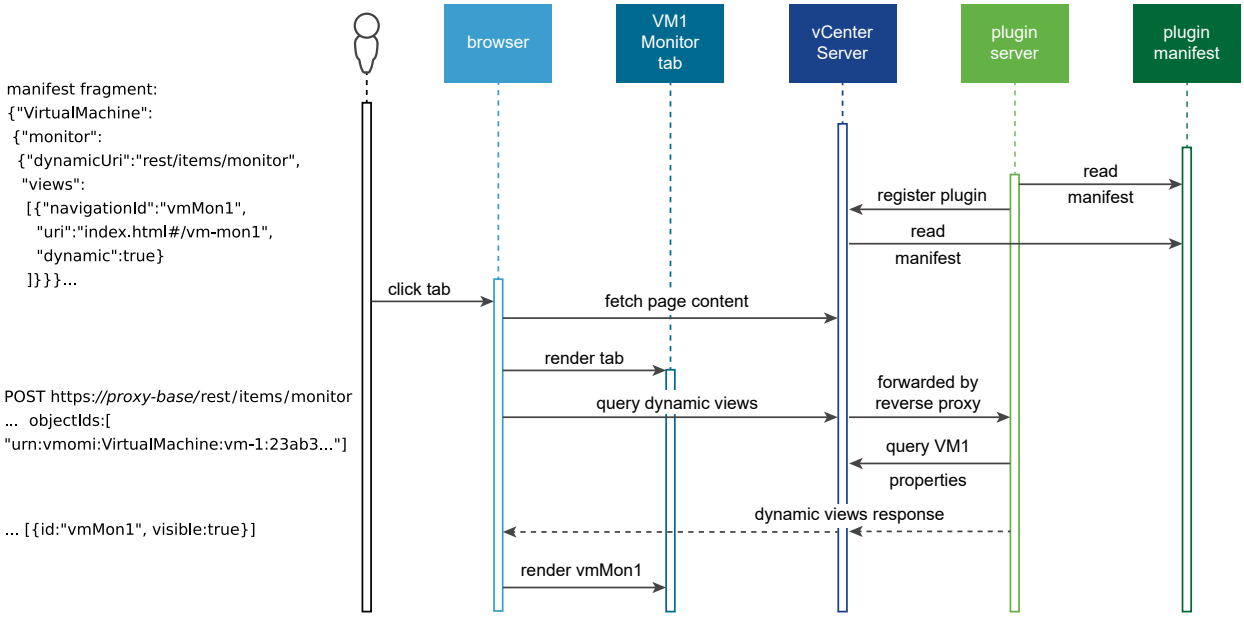
What to do next

Implement server-side code to handle filter queries at the endpoints that you configured in the manifest file. For more information about filter queries, see Dynamic Extensions Filter Query.

Dynamic Extensions Filter Query

The `vsphere-ui` process parses the plug-in manifest when the plug-in is registered wih the Extension Manager of vCenter Server, and stores the information as long as the plug-in remains registered. When a user navigates to a Summary tab, Monitor tab, or Configure tab, or when a user clicks the Actions menu, the user interface code in the browser sends a filter query to a plug-in server to determine its choices for dynamic extensions. The plug-in server responds with a visibility choice for each dynamic extension that applies in the current context.

An example message sequence for a filter query is illustrated in the following diagram.



Notes about the sequence diagram:

- Reading the manifest file is not part of the filter query sequence. The diagram includes the manifest fragment for context.
- The variable *`proxy-base`* in the endpoint URL refers to the proxied plug-in server base URL that the client code prefixes to the `dynamicUri` in the manifest file. All client queries to the plug-in server pass through the reverse proxy for security reasons. For more information, see Client-Server Communications with Remote Plug-ins.

The client filter query consists of an HTTPS POST request to a `dynamicUri` endpoint specified in the plug-in manifest file. The client code chooses the URL configured for the vSphere object type and the extension type that are currently active in the user interface.

Tip A best practice is to maintain a different endpoint for each extension type (Summary, Monitor, Configure, or Menu) and return visibility choices (`true` or `false`) for only the relevant dynamic extensions.

The request header contains:

- The body type specifications (`'application/json'` for both request and response).
- HTTP cache control settings.
- Identity and authentication headers:
  - Node ID (unused)
  - The vsphere-ui endpoint URL This tells the plug-in server which vCenter Server manages the context objects.
  - The session token for the client's authentication with the vsphere-ui endpoint. This allows the plug-in server to authenticate with vCenter Server to retrieve properties needed to make filtering choices.

For example:

```
'content-type': 'application/json',
'Accept': 'application/json',
'Cache-Control': 'no-cache, no-store, max-age=0',
vmware-api-session-id: c621c819-4f65-1b02-2214-c7ac159ad4d4
vmware-api-ui-endpoint-url: https://93.184.216.34/api/ui
vmware-api-ui-node-id: 6079314c-d525-43a1-8a54-735f5417f11e
```

The request body contains:

- The API version of the filter query protocol, used to negotiate message format with the plug-in server.
- The locale code.
- A list of object IDs for the current context objects. For dynamic actions, the list of object IDs may be any length. For dynamic views, the list may contain only the object ID that pertains to the **Summary** tab or **Monitor** tab or **Configure** tab that the client is rendering.

For example:

```
apiVersion: "1.0.0"
objectIds: ["urn:vmomi:VirtualMachine:vm-1005:27a09c68-d1d6-4fe2-a28f-616949f30930"]
locale: "en-US"
```

The response from the plug-in server must be a single anonymous JSON object that contains:

- The latest version of the filter query protocol that the server is prepared to handle, used to negotiate message format with the client.
- A list of dynamic items, where each item is an object that contains:
  - The `id` or `navigationId` of a dynamic view or the `id` of a dynamic menu action. A `navigationId` value is assigned to the `id` property in the response object.
  - Boolean values `visible` to control whether the dynamic extension displays in the user interface, and (as of 8.0 U3) `relevant` to control whether to remove it from the interface.

For example:

```
{
  apiVersion: "1.0.0",
  dynamicItems:
  [
    {
      id: "MonitorDynView",
      visible: true,
      relevant: true
    },
    {
      id: "DeleteAction",
      visible: false,
      relevant: false
    }
  ]
}
```

Example Code for Filtering Dynamic Extensions

Plug-ins that support dynamic extensions must implement the SPI that the client code uses to determine which dynamic extensions to display in the browser. The plug-in server can use any criterion chosen by the plug-in developer. The following example illustrates server code that uses two kinds of criteria to decide whether dynamic extensions should be visible..

Two useful criteria for filtering dynamic extensions are properties of vSphere objects and the authorization level of the vSphere Client user. This example tests the run state of a virtual machine and the user's authorization with respect to the virtual machine.

This example depends on a manifest file that configures two dynamic extensions. One extension provides a dynamic Monitor view of a virtual machine. The other extension provides a dynamic Menu action to run a virtual machine.

The following example code illustrates a controller that serves dynamic extension endpoints. The controller accesses the JSON in the request body by using the DynamicItemsRequestModel object passed in for the specific REST endpoint.

```
package com.example.remote.controllers;
import java.util.ArrayList;
import java.util.List;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.vmware.sample.remote.model.DynamicItem;
import com.vmware.sample.remote.model.DynamicItemsRequestModel;
import com.vmware.sample.remote.model.PluginServerDynamicItemsResponse;
import com.vmware.sample.remote.vim25.services.AuthorizationService;
```

```
/**
* Provide public endpoints for vSphere Client to query about
* UI visibility for dynamic views/actions.
*/
@RestController
@RequestMapping(value = "/dynamicItems",
                method = RequestMethod.POST,
                consumes = MediaType.APPLICATION_JSON_VALUE,
                produces = MediaType.APPLICATION_JSON_VALUE)
public class DynamicItemsController {
   private static final String MANAGE_VM_PRIVILEGE =
"com.vmware.sample.remote.1.0.0.ManageVm";
   private final AuthorizationService authorizationService;
   private final VmRunstateService vmRunstateService;

   public DynamicItemsController(final AuthorizationService authorizationService) {
      this.authorizationService = authorizationService;
      this.vmRunstateService = vmRunstateService;
   }

   /* This action should be visible if user has authorization. */
   @RequestMapping(value = "/vm/actions")
   public PluginServerDynamicItemsResponse retrieveVmActions(
      @RequestBody DynamicItemsRequestModel payload) {
      final boolean hasPrivilege = authorizationService
            .hasPrivilege(payload.objectIds, MANAGE_VM_PRIVILEGE);
      final List<DynamicItem> dynamicItems = new ArrayList<>();
      dynamicItems.add(new DynamicItem("TakeAction", hasPrivilege));
      return new PluginServerDynamicItemsResponse("1.0.0", dynamicItems);
   }

   /* This VM view should be visible only if it is NOT currently running. */
   @RequestMapping(value = "/vm/monitor")
   public PluginServerDynamicItemsResponse retrieveVmMonitorViews(
      @RequestBody DynamicItemsRequestModel payload) {
      final String objectId = payload.objectIds.get(0);
      final boolean makeVisible = !isVmStateRunning(objectId));
      final List<DynamicItem> dynamicItems = new ArrayList<>();
      dynamicItems.add(new DynamicItem("MonitorDynView", makeVisible));
      return new PluginServerDynamicItemsResponse("1.0.0", dynamicItems);
   }
}
```

vSphere Client Plug-in User Interface Modules

7

The vSphere Client provides several JavaScript interfaces that your plug-in can use to communicate with the vSphere Client 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.

The plug-in web application runs in a separate iFrame which is part of the vSphere Client. The iFrame content is rendered from the web application server of the plug-in back end.

Note Do not access the `window.parent`, which belongs to the vSphere Client. Do not access the internal JavaScript or CSS resources of the vSphere Client. Such access is unsupported and could cause your plug-in to fail in a future release of the vSphere Client.

Read the following topics next:

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

Bootstrapping the JavaScript API

The vSphere Client loads plug-in resources in a tenant iFrame. The plug-in must load a thin JavaScript library to support communication with the parent window in the vSphere Client. The library implements a JavaScript API that the plug-in front-end code uses to manage resources outside its iFrame.

To bootstrap the Client Library the following script should be added to all HTML pages in the plug-in:

```
<script type="text/javascript"src="/api/ui/htmlClientSdk.js"></script>
```

After you load the script, you initialize the `htmlClientSdk` object, by invoking the `htmlClientSdk.initialize()` method. Before you initialize, you can only invoke the methods of the `htmlClientSdk` interface. After you initialize the `htmlClientSdk` object, you can invoke any of the methods in the JavaScript API.

If you use frameworks such as jQuery, or zone.js with Angular, you only need to initialize the `htmlClientSdk` object once. You should initialize as early as possible, so that the `htmlClientSdk` functions will be available to all plug-in user interface components.

Note Do not use any communication method not provided by the APIs. Do not access any internal JavaScript or CSS resources of the vSphere Client. Doing so is unsupported because the implementation of the `htmlClientSdk` functions can change in future releases of the vSphere Client.

vSphere Client JavaScript API: htmlClientSdk Interface

The htmlClientSdk object provides access to all the JavaScript API interfaces. You load and initialize the htmlClientSdk first in your plug-in views.

`htmlClientSdk.getProxiedPluginServerOrigin()`

Signature	`htmlClientSdk.getProxiedPluginServerOrigin():string`
Description	Returns a proxy URL for the plug-in manifest server root. The URL contains the protocol, domain, and port of the vCenter Server, with a proxy path that routes to the plug-in manifest server root. This URL is useful to set a root-relative `base` tag for relative URLs in a front-end framework such as Angular. This method can be safely called before the SDK has been initialized.

`htmlClientSdk.initialize()`

Signature	`htmlClientSdk.initialize(callback:function):void`
Description	Initializes the htmlClientSdk object and invokes the callback function when initialization is complete and the other JavaScript interfaces are available. After the first call, subsequent calls act to register additional callbacks.
Parameter: `callback`	The optional callback function must have the following signature: `function callback():void` The callback function has access to all the functions in the JavaScript API. You use the callback to code logic for initializing plug-in state.

`htmlClientSdk.isInitialized()`

Signature	`htmlClientSdk.isInitialized():boolean`
Description	Tests whether the htmlClientSdk object has been initialized.

vSphere Client JavaScript API: Modal Interface

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

Note This SDK uses the Structured Clone Algorithm, which has limitations as described in the Mozilla Developer Network (MDN) web doc. Limitations affect Modal, Confirmation Modal, and Shared Modal APIs.

`modal.AlertLevel`

Description	Enum: ■ `SUCCESS` ■ `INFO` ■ `WARNING` ■ `DANGER` Clarity adds a standard icon, depending on the alert level. See https://v2.clarity.design/alerts.

Used by: `modal.ConfirmationModalConfig`

`modal.ButtonStyle`

Description	Enum: ■ `SUCCESS` ■ `INFO` ■ `WARNING` ■ `DANGER` Clarity adds default CSS tags, depending on the action importance: ■ `btn-success` ■ `btn-primary` ■ `btn-warning` ■ `btn-danger` For examples of Clarity button styling, see the `Solid Buttons` illustrations at https://v2.clarity.design/buttons/#examples.

Used by: `modal.ModalButton`

`modal.ButtonType`

Description	Enum: ■ `PRIMARY` ■ `SECONDARY` Clarity styles PRIMARY buttons with greater emphasis. See https://v2.clarity.design/buttons.

Used by: `modal.ModalButton`

modal.close()

Signature	Closes the modal dialog box in the parent iFrame.
Description	Optional data that will be passed to callback function specified by `onClosed` property at dialog open.
Parameter: `data`	Optional data that will be passed to callback function specified by `onClosed` property at dialog open.

Note No data is passed to the callback function if the user clicks the dialog close box.

modal.ConfirmationModalConfig

Description	Specifies the properties of a confirmation modal dialog box.

Property	Type	Required?	Notes
`content`	`string`	yes	Confirmation message displayed in the dialog.
`buttons`	`modal.ModalButton[]`	yes	Buttons displayed in the dialog footer. (min 1, max 4)
`title`	`string`	no	Dialog main title. May not contain an icon. (default=`''`)
`AccessibilityTitle`	`string`	no	Used when dialog title is not displayed, such as in a wizard dialog.
`size`	`modal.ModalSize`	no	Width of the dialog box. (Default width is chosen by Clarity. Height is fixed.)
`closable`	`boolean`	no	Whether the dialog displays a close button. (default=`true`)
`onClosed`	`function(result:any): void`	no	Function runs when user closes the dialog. If `modal.close()` is called, its parameter is in turn passed to the `onClosed()` function.
`secondaryTitle`	`string`	no	Optional subtitle for the dialog.
`alertLevel`	`modal.AlertLevel`	no	Causes Clarity to add an icon to the dialog.

Used by: `modal.openConfirmationModal()`

modal.DynamicModalConfig

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

Property	Type	Required?	Notes
`title`	`string`	no	Dialog title. May not contain an icon. (If not present, no change to dialog title.)
`accessibilityTitle`	`string`	no	Used when dialog title is not displayed, such as in a wizard dialog.
`height`	`number`	no	Dialog height, specified in pixels. (If not present, no change to dialog height.)

Used by: `modal.setOptions()`

modal.getCustomData()

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.getSharedModalsMap()

Signature	`modal.getSharedModalsMap(targetPluginId:string, callback:function):void`
Description	Creates listings of modal dialog boxes shared by a specific plug-in and accessible to the calling plug-in. The callback function must have the following signature: `function callback(sharedModalsMap:modal.SharedModalsMap):void`

modal.ModalButton

Description	Describes button properties.

Property	Type	Required?	Notes
`label`	`string`	yes	
`type`	`modal.ButtonType`	no	
`style`	`modal.ButtonStyle`	no	
`callback`	`function(): void`	no	

Used by: `modal.ConfirmationModalConfig`

modal.ModalConfig

Description	Specifies the properties of a modal dialog box.

Property	Type	Required?	Notes
`url`	`string`	yes	Location of HTML content for the dialog.
`title`	`string`	no	Dialog title. May not contain an icon. (default=`''`)
`accessibilityTitle`	`string`	no	Used when dialog title is not displayed, such as in a wizard dialog.
`size`	`(width:number, height:number)`	no	Specify in pixels. Default is chosen by Clarity. See https://v2.clarity.design/modals.
`closable`	`boolean`	no	Whether the dialog displays a close button. (default=`true`)
`onClosed`	`function(result:any): void`	no	Function runs when the dialog closes. If `modal.close()` is called, its parameter is in turn passed to the `onClosed()` function. Function runs when the dialog closes. If `modal.close()` is called, its parameter is in turn passed to the `onClosed()` function.
`customData`	`any`	no	Data the calling module passes to the dialog.
`contextObjects`	`any[]`	no	IDs of relevant objects the calling module passes to the dialog.

Used by: `modal.open()`

`modal.ModalSize`

Description	Width of a modal dialog box.

Property	Type	Required?	Notes
`width`	`number`	yes	Specified in pixels.

Used by: `modal.ConfirmationModalConfig`

`modal.open()`

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

modal.openConfirmationModal()

Signature	`modal.openConfirmationModal(configObj:modal.ConfirmationModalConfig):void`
Description	Opens a lightweight modal dialog box designed to present information and confirmation buttons.
Parameter: `configObj`	Specifies the properties of this modal dialog box.

modal.openSharedModal()

Signature	`modal.openSharedModal(configObj:modal.SharedModalConfig):void`
Description	Opens a modal dialog box that is defined and shared by another plug-in.

modal.SharedModalConfig

Description	Specifies the properties for opening a shared modal dialog.

Property	Type	Required?	Notes
`pluginId`	`string`	yes	ID of the plug-in that defines the dialog.
`sharedModalId`	`string`	yes	ID of the shared modal dialog.
`vcGuid`	`string`	yes	The GUID of the vCenter Server instance where the plug-in that defines the dialog is registered.
`objectId`	`string`	no	
`onClosed`	`function(result:any):void`	no	Function runs when user closes the dialog. If `modal.close()` is called, its parameter is in turn passed to the `onClosed()` function.
`contextObjects`	`any[]`	no	IDs of relevant objects the calling module passes to the dialog.
`customData`	`any`	no	Additional data the calling module passes to the dialog.

Used by: `modal.openSharedModal()`

modal.setOptions()

Signature	`modal.setOptions(configObj:modal.DynamicModalConfig):void`
Description	Called by the parent view to modify some properties for a modal dialog box in the parent iFrame.
Parameter: `configObj`	Specifies values for some dialog box properties.

modal.SharedModalsMap

Description	Information about modal dialog boxes shared by a specific plug-in instance registered with one or more vCenter Server instances.

Property	Type	Required?	Notes
sharedModalsInfoByVcGuid	[VcGuid:string]:modal.VcSharedModalsInfo	yes	Map of modal dialog boxes shared by a plug-in instance, listed by each vCester Server instance where the plug-in is registered.
sharedModalsInfoByVcObjects		no	Reserved for future use.

Used by: modal.getSharedModalsMap()

modal.VcSharedModalsInfo

Description	Information about modal dialog boxes shared by a plug-in instance registered with a specific vCenter Server instance.

Property	Type	Required?	Notes
pluginVersion	string	yes	The plug-in version registered with the vCenter Server instance.
sharedModalsIds	string[]	yes	List of modal dialog boxes shared by the plug-in.

Used by: modal.SharedModalsMap

vSphere Client JavaScript API: Application Interface

The app interface provides context object information and helps your plug-in navigate and control the vSphere Client user interface.

app.ApiEndpoints

Description	Holds a nested object that contains the parsed elements of the plug-in URL.

Property	Type	Required?	Notes
uiApiEndpoint	app.UiApiEndpoint	info only	

Used by: app.getApiEndpoints()

app.ClientInfo

Description	Documents type and version of vSphere Client.

Property	Type	Required?	Notes
type	string	info only	The vSphere Client type (must be HTML).
version	string	info only	The vSphere Client version string.

Used by: app.getClientInfo()

app.ClientViewNavigationOptions

Description	Specifies a destination view that is owned by the vSphere Client.

Property	Type	Required?	Notes
targetviewid	string	no	Navigation ID of the destination view. ■ administration.ceip ■ cluster.configure.settings.drs ■ cluster.configure.settings.ha ■ cluster.configure.settings.quickstart ■ cluster.monitor.allIsues ■ folder.configure.keyProviders ■ host.configure.networking.physicalAdapters ■ host.summary ■ namespace.storage.persistentVolumeClaims ■ network.summary ■ profilesAndPolicies.vmStoragePolicies ■ vm.monitor.performance.overview ■ vm.summary ■ folder.configure.settings.general ■ workload.platform.supervisor.summary
objectid	string	no	ID of any object associated with the view. (For a global view, this field is not required.).

Used by app.navigateToClientView()

app.formatDateTime()

Signature	app.formatDateTime(instant:number, options:app.PluginDateTimeFormatOptions):string
Description	Returns a formatted string that contains a human-readable representation of the instant parameter. The instant parameter holds the number of milliseconds since the start of 1 January 1970 UTC.

app.getApiEndpoints()

Signature	`app.getApiEndpoints():app.ApiEndpoints`
Description	Returns the URLs of the vsphere-ui service API endpoints available to plug-ins. For an example, see the sample remote plug-ins available in the vSphere Client SDK.

app.getClientInfo()

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

app.getClientLocale()

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

app.getContextObjects()

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

app.getNavigationData()

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

app.getPluginBackendInfo()

Note Applies to remote plug-ins only.

Signature	`app.getPluginBackendInfo(callback:function):void`
Description	Creates listings of service endpoints registered for the running plug-in instance, enabling plug-in front ends to use the vCenter Server `ExtensionManager` as a service registry. The callback function must have the following signature: `Function callback(backendInfo:app.PluginBackendInfo):void` If the method throws an exception, the `callback` parameter will be `null`.

app.getRemotePluginNavigationMap()

Signature	`app.getRemotePluginNavigationMap(targetPluginId:string, callback:function):void`
Description	Returns view IDs that belong to a plug-in specified by the `targetPluginId` parameter. If the targetPluginId is not the ID of the caller, the method returns IDs of only the public views for the target plug-in. Returns information for all vCenter Server instances where the target plug-in is registered and enabled. The callback function must have the following signature: `function callback(navigationInfo:app.RemotePluginNavigationMap):void` If the method throws an exception, the `callback` parameter will be `null`.

app.getSessionInfo()

Signature	`app.getSessionInfo(callback:function):void`
Description	Retrieves and processes information about the client's authentication session. The callback function must have the following signature: `function callback(info:app.SessionInfo):void` If the method throws an exception, the `callback` parameter will be `null`.

app.getTheme()

Signature	`app.getTheme():app.PluginTheme`
Description	Retrieves information about the UI theme that is currently selected.

app.navigateTo()

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

app.navigateToClientView()

Signature	`app.navigateTo(configObj:app.ClientViewNavigationOptions):void`
Description	Navigates to a specified vSphere Client view.
Parameter: `configObj`	Specifies the destination view.

app.navigateToRemotePluginView()

Signature	`app.navigateTo(configObj:app.RemotePluginViewNavigationOptions):void`
Description	Navigates to a view implemented by another remote plug-in, or by another instance of the same plug-in deployed by a different vCenter Server instance. Optionally passes custom data to the view.
Parameter: `configObj`	Specifies the destination view and custom data.

Note A best practice is to call app.getRemotePluginViewNavigationMap() before this procedure, to ensure that the navigation target is registered and enabled.

app.NavigationOptions

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

Property	Type	Required?	Notes
`targetViewId`	`string`	no	Navigation ID of the destination view. (For a remote plug-in, this property must identify a view created by the same plug-in.) Omit this to navigate to the last-used view of the object specified by `objectId`.
`objectId`	`string`	no	ID of any object associated with the view. Omit this to navigate to a global view.
`customData`	`any`	no	A custom data structure passed to the view.

Used by: app.navigateTo()

app.PluginBackendInfo

Note Applies to remote plug-ins only.

Description	Contains two objects that list endpoint descriptors available to a given plug-in instance.

Property	Type	Required?	Notes
`allPluginBackendServers`	`Array<app.PluginBackendServerInfo>`	info only	A list of plug-in server endpoint descriptors registered for a given plug-in instance. The list includes all registrations within the same link group. The list is unordered and contains no duplicates.
`backendServersPerVc`	`{[vcGuid:string]:Array<app.PluginBackendServerInfo>}`	info only	A one-to-many mapping: vCenter Server GUID to a list of plug-in server endpoint descriptors registered with the vCenter Server instance on behalf of the running plug-in instance.

Used by: app.getPluginBackendInfo()

app.PluginBackendServerInfo

Note Applies to remote plug-ins only.

Description	A descriptor for a plug-in server endpoint registered for a plug-in instance. Choose the server by the `type` value, then form a resource URL from the endpoint descriptor. To access a resource belonging to the server, prefix the `proxiedBaseUrl` value to the root-relative path of the resource on the plug-in server: /<proxiedBaseUrl>/<path to resource from server root>

Property	Type	Required?	Notes
`proxiedBaseUrl`	`string`	info only	The path component of the server root URL, as seen on the reverse proxy service port.
`type`	`string`	info only	The `type` of the server, as specified in its registration record (`Extension` data object) with a vCenter Server instance. Plug-ins can use this property to identify auxiliary servers that are part of the same plug-in instance. For more information, see Using Auxiliary Plug-in Servers.

Used by: app.PluginBackendInfo

app.PluginDateTimeFormatOptions

Description	Specifies which part of a timestamp to format.

Property	Type	Required?	Notes
`format`	`string`	no (default: `DATE_AND_TIME`)	Possible values: ■ `DATE` ■ `TIME` ■ `DATE_AND_TIME`

Used by: app.formatDateTime()

app.PluginTheme

Description	Indicates a choice of UI theme.

Property	Type	Required?	Notes
`name`	`string`	info only	Possible values: `light` or `dark`.

Used by: app.getTheme()

app.QueryParam

Description	Holds a single query parameter of a URL.

Property	Type	Required?	Notes
`name`	`string`	info only	Name of query parameter, as in `?name=value`.
`value`	`string`	info only	Value of query parameter, as in `?name=value`.

Used by: app.UiApiEndpoint

app.refreshPluginItemsState()

Signature	`app.refreshPluginItemsState():void`
Description	Triggers the refresh of all dynamic UI content hosted by the current plug-in. For example, the plug-in introduces a view that is dynamically filtered based on a property value of some object. If the property value is changed by the plug-in using the public vSphere APIs, then the plug-in must call this API to signal the UI to re-evaluate the state of the initiator's plug-in related items.

app.RemotePluginNavigationMap

Description	Lists remote plug-in destination views, belonging to a single vCenter Server instance, that are exposed by plug-ins using the `isPublic` property in the plug-in manifest. Use the map key as the value of the `vcGuid` property in the parameter to `app.navigateToRemotePluginView()`.

Property	Type	Required?	Notes
navigationInfoByVcGuid	Map<string, app.RemotePluginVcNavigationInfo>	info only	A one-to-many mapping of vCenter Server instances to view IDs that are served by a specified plug-in instance registered with each vCenter Server instance. Each map entry contains the GUID of a vCenter Server instance within the same link group, and a corresponding list of view descriptors.

Used by: app.getRemotePluginNavigationMap()

app.RemotePluginVcNavigationInfo

Description	Descriptor for views served by a given plug-in. Use a view ID as the value of the targetViewId property in the parameter to app.navigateToRemotePluginView()

Property	Type	Required?	Notes
pluginVersion	string	info only	The version of a plug-in instance registered with a given vCenter Server instance.
viewIds	string[]	info only	All the view IDs, served by a given plug-in instance, that are accessible to the current plug-in instance.

Used by: app.RemotePluginNavigationMap

app.RemotePluginViewNavigationOptions

Description	Specifies a destination view served by a different plug-in instance, and custom data for the view.

Property	Type	Required?	Notes
pluginId	string	yes	ID of the plug-in that owns the destination view.
targetViewId	string	yes	Navigation ID of the destination view. Use only values returned by htmlClientSdk.app.getRemotePluginNavigationMap() in the viewsIds array.
vcGuid	string	yes	The GUID of the vCenter Server instance that determines the context for the target view. If objectId is specified, the object must be managed by the specified vCenter Server instance.
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.

Used by: app.navigateToRemotePluginView()

app.SessionInfo

Description	Holds information about the current session of the vSphere Client.

Property	Type	Required?	Notes
sessionToken	string	info only	Identifier of the plug-in authentication session with vCenter Server.
nodeId	string	info only	Reserved for internal use only.

Used by: app.getSessionInfo()

app.UiApiEndpoint

Description	Holds the parsed elements of a plug-in URL.

Property	Type	Required?	Notes
origin	string	info only	<protocol>://<hostname><port>
pathname	string	info only	
queryParams	Array<app.QueryParam>	info only	<name>=<value>
fullUrl	string	info only	<origin>/<pathname>?<queryParams>

Used by: app.ApiEndpoints

vSphere Client JavaScript API: Event Interface

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

event.onDateTimeFormatChanged()

Signature	event.onDateTimeFormatChanged(callback:function):void
Description	Registers an event handler for changes to the datetime format preferences in the vSphere Client.
Parameter: callback	A reference to a function that responds to changes in datetime preferences. The callback function must have the following signature: `function callback():void`

`event.onGlobalRefresh()`

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

`event.onThemeChanged()`

Signature	`event.onThemeChanged(callback:function):void`
Description	Registers an event handler that the vSphere Client will call when the vSphere Client changes the current theme.
Parameter: `callback`	A reference to a theme change handler. The callback function must have the following signature: `function callback(theme:app.PluginTheme):void` The `theme` parameter identifies the new vSphere Client theme.

Example Using the `modal` API

This example shows some basic features of the `modal` interface of the Client API.

modal.html

```
<html>
  <head>
    <script src="http://code.jquery.com/jquery-latest.min.js"
            type="text/javascript"></script>
    <script src="/api/ui/htmlClientSdk.js"
            type="text/javascript"></script>
    <script type='text/javascript'>
      function handler(event)
      {
        var choice = $('input[name=heads_or_tails]:checked').val();
        htmlClientSdk.modal.setOptions({title: choice});
        settimeout(function(){htmlClientSdk.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

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

  # Create callback function.
  checker = function(choice){
    var correct = htmlClientSdk.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.
  htmlClientSdk.modal.open(config);
}

# Initialize Javascript API.
$(document).ready(htmlClientSdk.initialize(flipper));
```

Using Themes with vSphere Client Plug-ins

8

The vSphere Client SDK provides the means for a plug-in to integrate with the themes supported by the vSphere Client. Modifying a plug-in to support themes requires changes to the plug-in style sheets and front-end code to switch style sheets whenever the user changes the theme in the vSphere Client.

To integrate with the vSphere Client themes, a plug-in uses these methods of the JavaScript API:

- `app.getTheme()`
- `event.onThemeChanged(callback)`

To prepare your code for a theme change, you must identify and isolate theme-dependent styles and icons, then create variables with which to manage the style changes and overrides.You can merge your styles with the standard Clarity styles to improve performance. Finally, you can use examples in this book to load new styles in response to user theme changes.

The following procedures assume that the plug-in's front-end code is built using Angular and Clarity Design System. For other frameworks and build tools, the approach is similar but you will need to adapt the approach to suit the chosen tools. The examples in this guide are based on the HTML Plug-in Sample provided as part of the vSphere Client SDK.

Read the following topics next:

- Using Style Variables in Plug-In CSS
- Building Output Style Sheets for vSphere Client Plug-Ins
- Configuring and Loading Theme Style Sheets in vSphere Client Remote Plug-Ins
- Configuring Theme-Dependent Icons for vSphere Client Remote Plug-ins

Using Style Variables in Plug-In CSS

If a plug-in uses custom styles that depend on the theme colors, the plug-in style sheets (CSS or SASS or LESS) need to be parameterized. This enables the plug-in to adapt when the user switches themes in the vSphere Client user interface.

In this procedure you copy any custom colors that depend on the current theme into variables in separate style sheets that are specific to the `light` or `dark` theme. You replace the colors in the original style sheets with instances of CSS variables. This is done to avoid style sheet duplication and to easily integrate theming with any custom Angular components the plug-in has defined. For more information about CSS variables, see https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_variables.

Prerequisites

Ensure that the plug-in's Clarity version supports the `dark` theme. The first Clarity version to support the `dark` theme is 0.10.16.

Procedure

1. Identify any theme-dependent colors or styles in your plug-in.
2. Factor out theme-dependent colors or styles into two new style sheets as CSS variables.

   The SDK includes the following sample file at `html-client-sdk/samples/remote-plugin-sample/src/main/ui/src/styles-light.css`.

   ```
   :root {
           --border-color: rgb(204, 204, 204);
           --overlay-color: rgba(255, 255, 255, 0.2);
           --info-icon-color: darkblue;
         }
   ```

   The SDK includes the following sample file at `html-client-sdk/samples/remote-plugin-sample/src/main/ui/src/styles-dark.css`.

   ```
   :root {
           --border-color: rgb(72, 87, 100);
           --overlay-color: rgba(0, 0, 0, 0.2);
           --info-icon-color: darkblue;
         }
   ```

3. Replace the theme-dependent colors or styles in the original style sheets with variable references.

   The SDK includes the following code in the sample file at `html-client-sdk/samples/remote-plugin-sample/src/main/ui/src/app/views/list/list.component.scss`.

   ```
   .splitter {
               flex: 0 0 auto;
               width: 1px;
               margin: 0 20px;
               background-color: var(--border-color);
             }
   ```

4 For Internet Explorer 11, which does not include support for CSS variables, include a polyfill library to provide support for CSS variables.

   The vSphere Client SDK includes a remote plug-in sample that uses `css-vars-ponyfill`. The following example is borrowed from `html-client-sdk/samples/remote-plugin-sample/src/main/ui/src/index.html`.

```
<script type="text/javascript" src="scripts/css-vars-ponyfill.js"></script>
```

What to do next

- Configure theme-dependent icons in the plug-in manifest file.
- Use the modified input style sheets to build the output style sheets for your plug-in.

Building Output Style Sheets for vSphere Client Plug-Ins

After you isolate theme-dependent colors or styles as CSS variables, you can merge the resulting style sheets with the standard Clarity styles to produce a set of output style sheets for optimized performance.

Angular applications which use `webpack` and `angular-cli` place the style sheet declarations inline by default, when in development mode. Inline style declarations interfere with dynamic CSS loading. When you build the output style sheets, always configure the build to output and use external CSS:

To build external style sheets, add the **`--extract-css`** parameter to the **`ng build`** command. The vSphere Client SDK has examples of this usage in `html-client-sdk/samples/remote-plugin-sample/src/main/ui/package.json`.

You must deactivate any output file name hashing in the development and production builds. Otherwise the names of the style sheet files will change whenever the code changes, and the plug-in will not be able to load them.

To deactivate file name hashing when you build style sheets, use this syntax: **`ng build --prod --output-hashing none`**.

Prerequisites

Refactor the input style sheets for the plug-in so that they isolate theme-dependent colors and styles in separate style sheets as CSS variables.

Procedure

1 Create a base output style sheet that is independent of the themes.

   The base style sheet contains the Clarity icons style sheet and the base input style sheet for the plug-in, which uses CSS variables. The vSphere Client SDK builds this output style sheet by using Angular to compile the SCSS.

The following example comes from the vSphere Client SDK file `html-client-sdk/samples/remote-plugin-sample/src/main/ui/angular-cli.json`.

```
"styles": [
  {
    "input": "../node_modules/clarity-icons/clarity-icons.min.css",
    "output": "styles",
    "lazy": true
  },
  {
    "input": "styles.css",
    "output": "styles",
    "lazy": true
  }
  ...
]
```

This step combines the contents of `html-client-sdk/samples/remote-plugin-sample/src/main/ui/node_modules/remote-plugin-sample/src/main/ui/styles.css` and `html-client-sdk/samples/remote-plugin-sample/src/main/ui/styles.css` into `html-client-sdk/samples/remote-plugin-sample/target/classes/ui/styles.bundle.css`.

2 Create an output style sheet file for the `light` theme.

This style sheet includes the Clarity style sheet for the `light` theme and the plug-in style sheet for the `light` theme, which contains the CSS variable definitions for the `light` theme.

The following example comes from the vSphere Client SDK file `html-client-sdk/samples/remote-plugin-sample/src/main/ui/angular-cli.json`.

```
"styles": [
  ...
  {
    "input": "../node_modules/clarity-ui/clarity-ui.min.css",
    "output": "theme-light",
    "lazy": true
  },
  {
    "input": "styles-light.css",
    "output": "theme-light",
    "lazy": true
  }
  ...
]
```

This step combines the contents of `html-client-sdk/samples/remote-plugin-sample/src/main/ui/node_modules/clarity-ui/clarity-ui.min.css` and `html-client-sdk/samples/remote-plugin-sample/src/main/ui/styles-light.css` into `html-client-sdk/samples/remote-plugin-sample/target/classes/ui/theme-light.bundle.css`.

3 Create an output style sheet file for the `dark` theme.

This style sheet includes the Clarity style sheet for the `dark` theme and the plug-in style sheet for the `dark` theme, which contains the CSS variable definitions for the `dark` theme.

The following example comes from the vSphere Client SDK file `html-client-sdk/samples/remote-plugin-sample/src/main/ui/.angular-cli.json`.

```
"styles": [
  ...
  {
    "input": "../node_modules/clarity-ui/clarity-ui-dark.min.css",
    "output": "theme-dark",
    "lazy": true
  },
  {
    "input": "styles-dark.css",
    "output": "theme-dark",
    "lazy": true
  }
  ...
]
```

This step combines the contents of `html-client-sdk/samples/remote-plugin-sample/src/main/ui/node_modules/clarity-ui/clarity-ui-dark.min.css` and `html-client-sdk/samples/remote-plugin-sample/src/main/ui/src/styles-dark.css` into `html-client-sdk/samples/remote-plugin-sample/target/classes/ui/theme-dark.bundle.css`.

What to do next

Write front-end code to load style sheets that match the theme selected by the user.

Configuring and Loading Theme Style Sheets in vSphere Client Remote Plug-Ins

After you compile the output style sheets for your plug-in user interface, you write front-end code to load the style sheets that cause your plug-in to conform to the style selected in the vSphere Client.

Prerequisites

- Refactor the input style sheets for the plug-in so that they isolate theme-dependent colors and styles in separate style sheets as CSS variables.
- Configure theme-dependent icons in the plug-in manifest file.
- Build output style sheets into a base style sheet and a style sheet for each theme.

Procedure

1 Load and configure polyfill libraries to provide CSS variable support in Internet Explorer 11.

If you use `css-vars-ponyfill`, consider whether to configure options to create a MutationObserver and whether to remove CSS rulesets and declarations that do not reference a CSS custom property value. For more information about configuring `css-vars-ponyfill`, see https://github.com/jhildenbiddle/css-vars-ponyfill/tree/v1.17.1#optionswatch and https://github.com/jhildenbiddle/css-vars-ponyfill/tree/v1.17.1#optionsonlyvars.

The vSphere Client SDK includes a remote plug-in sample that uses `css-vars-ponyfill`. The following example is borrowed from the file `html-client-sdk/samples/remote-plugin-sample/src/main/ui/src/index.html`.

```
<script type="text/javascript" src="scripts/css-vars-ponyfill.js"></
script>
// Initialize CSS vars to configure polyfill.
cssVars({
          watch: true,
          onlyVars: true
});
```

The following example is borrowed from the file `html-client-sdk/samples/remote-plugin-sample/src/main/ui/.angular-cli.json`.

```
  "assets": [
    "assets",
    {
      "glob":
      "css-vars-ponyfill.js",
      "input": "../node_modules/css-vars-ponyfill/dist/",
      "output": "scripts/
    },
   …
  ]
```

2 Load the base style sheet initially.

The following example is borrowed from `html-client-sdk/samples/remote-plugin-sample//src/main/ui/src/index.html`.

```
<link rel="stylesheet" type="text/css" href="styles.bundle.css">
```

3 Load and initialize the vSphere Client JavaScript API.

```
<script type="text/javascript" src="/api/ui/htmlClientSdk.js"></script>
<script type="text/javascript">
  htmlClientSdk.initialize(init_plugin_view());
</script>
```

For examples in the SDK, see `html-client-sdk/samples/remote-plugin-sample//src/main/ui/src/index.html` and `html-client-sdk/samples/remote-plugin-sample//src/main/ui/src/app/app.component.ts`.

4 Load the style sheet for the current theme initially and whenever the style changes.

The following example is adapted from `html-client-sdk/samples/remote-plugin-sample/src/main/ui/src/app/app.component.ts`.

```
if (this.globalService.htmlClientSdk.app.getTheme &&
  this.globalService.htmlClientSdk.event.onThemeChanged) {
    this.loadTheme(true, this.globalService.htmlClientSdk.app.getTheme());
    this.globalService.htmlClientSdk.event.onThemeChanged(
                         this.loadTheme.bind(this, false));
  } else {
    this.loadTheme(true, { name: 'light' });
  }

private loadTheme(firstLoad: boolean, theme: any): void {
  let themeName: string = theme.name;
  let supportedThemeNames: string[] = ['light', 'dark'];
  if (supportedThemeNames.indexOf(themeName) === -1) {
    themeName = supportedThemeNames[0];
  }
  let styleSheetLinkElement =
    (<HTMLLinkElement> document.getElementById('theme-stylesheet-link'));
  let themeCssUrl = `theme-${themeName}.bundle.css`;

  if (firstLoad) {
    let initialThemeLoadCompleteListener = (event: Event) => {
      this.initialThemeLoadComplete = true;
      styleSheetLinkElement.removeEventListener('load',
                      initialThemeLoadCompleteListener);
      styleSheetLinkElement.removeEventListener('error',
                       initialThemeLoadCompleteListener);
    };

    styleSheetLinkElement.addEventListener('load',
                initialThemeLoadCompleteListener);
    styleSheetLinkElement.addEventListener('error',
                 initialThemeLoadCompleteListener);
  }

  styleSheetLinkElement.setAttribute("href", themeCssUrl);
  document.documentElement.setAttribute("data-theme", themeName);
}
```

Configuring Theme-Dependent Icons for vSphere Client Remote Plug-ins

When you refactor style sheets for your plug-in to accommodate theme changes, you can also specify icons suited for alternative themes. You specify theme-dependent icons in the plug-in manifest file.

Configuring theme-dependent icons is an optional step that can improve the user experience after a theme change.

Procedure

1 Edit the plug-in manifest file, `plugin.json`, and locate the `iconSpriteSheet` property at the second level of the JSON, within the `definitions` property.

```
...
"definitions": {
  "iconSpriteSheet": {
    "uri" : "assets/images/sprites.png",
    "definitions": {
      "main" : {
        "x": 0,
        "y": 96
      }
    }
  },
  "i18n": {
    "locales": ["en-US"],
    "definitions": {
      "plugin.name": {
        "en-US": "Theme Example"
      }
    }
  }
},
...
```

2 Within the `iconSpriteSheet` property, add a `themeOverrides` property that maps each theme name to a URI and the coordinates of its theme-dependent icons.

The structure of each theme property inside the `themeOverrides` property is identical to the structure of the `iconSpriteSheet` property, which contains a `uri` and a `definitions` property.

```
...
"definitions": {
  "iconSpriteSheet": {
    "uri" : "assets/images/sprites.png",
    "definitions": {
      "main" : {
        "x": 0,
        "y": 96
      }
```

```
    },
  "themeOverrides": {
    "dark": {
      "uri": "assets/images/sprites_dark.png",
      "definitions": {
        "main": {
          "x": 0,
          "y": 96
        }
      }
    }
  },
  "i18n": {
    "locales": ["en-US"],
    "definitions": {
      "plugin.name": {
        "en-US": "Theme Example"
      }
    }
  }
},
...
```

A best practice is to maintain a separate style sheet for each theme, with corresponding icons in the same positions. When you do this, you do not need to override the default coordinates, and you can omit the `definitions` from the `themeOverrides` element.

3 Save your changes and close the manifest file.

What to do next

Write front-end code to load style sheets that match the theme selected by the user.

Integrated Solution Installer for the vSphere Client

9

When you create a plug-in server encapsulated in an Open Virtual Format package (OVF), you can configure the OVF to automate the registration of the plug-in. This avoids the need to create a custom registration tool or to use the manual registration script in the SDK.

Read the following topics next:

- Conceptual Overview of the Preparation Process
- Deployment with the Integrated Solution Installer
- Integrated Installer Solution Metadata

Conceptual Overview of the Preparation Process

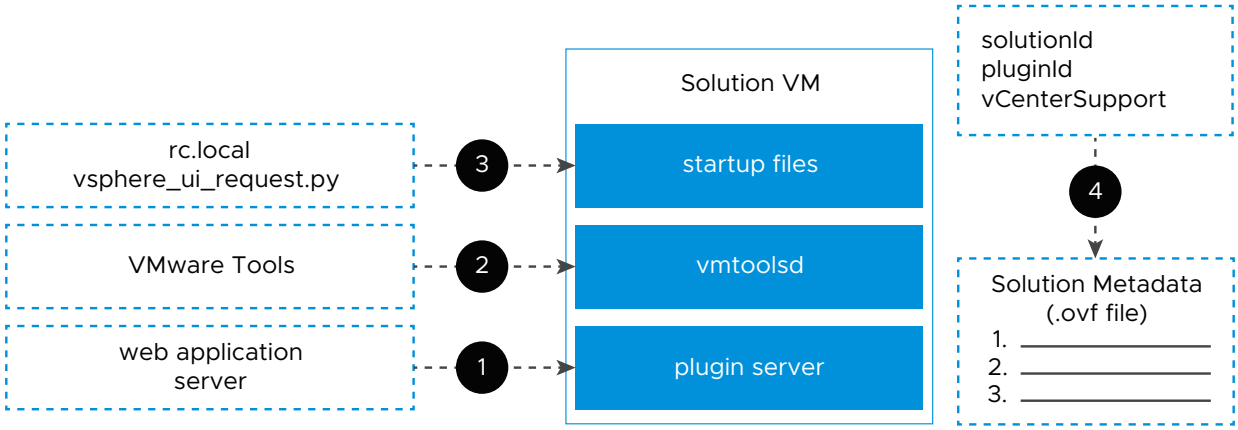
A self-registering plug-in solution requires the coordination of several software functions. The following procedure shows how to prepare an OVF package for self-registration by installing needed software and configuring metadata that guides the registration process when the virtual machine first starts up after deployment to a vCenter Server instance.

In brief, these are the needed software and metadata components:

- The virtual machine needs a web application server. The following instructions show how to install the Remote Sample Starter from the vSphere Client SDK, which you can customize to support your application. Alternatively, you could install `nginx` or another server that is capable of serving the plug-in manifest during the registration process.
- The virtual machine needs VMware Tools installed. VMware Tools provides the mechanism to inject environmental parameters into the virtual machine. The following instructions begin with downloading a Photon OS virtual machine that has VMware Tools already installed. If you choose a different virtual machine as a starting point, you might need to install VMware Tools.
- The virtual machine also needs files that run during startup to assemble the environmental parameters and send a registration request to vCenter Server. Sample registration files intended for a Linux guest OS are provided in the vSphere Client SDK. A bash script queries VMware Tools for the parameters needed to build the registration request and launch a Python program to do the registration.

- The registration request includes solution identifying details. During deployment of the OVF files, the vSphere Client passes these details to the guest OS by means of VMware Tools..

Figure 9-1. Preparing the Solution OVF for the Integrated Installer



Preparing a Solution OVF to Use the Integrated Solution Installer

You can prepare a virtual machine with software and configuration metadata that work in conjunction with the vSphere Client to register a plug-in solution when the virtual machine starts up in a production environment.

Prerequisites

- vCenter Server 6.7 or later.
- An OVA package that contains a virtual machine with virtual hardware version 11 or later. PhotonOS, as used in this procedure, is recommended.
- `root` or `administrator` access to the guest operating system.
- (If you choose to use Docker to download `nginx`) access to the Docker Hub.

Outline of Steps to Prepare a Plug-in Server for Self-Registration and Installation

1 Acquire the OVA File for Photon OS, virtual hardware version 11

2 Use the vSphere Client to Install the OVA in a vSphere Environment

3 Configure vApp Options for deployment in a production environment

4 Configure Solution Properties in the OVF

5 Run the virtual machine and Set the root Password in the Guest OS

6 Install Your Web Application Server in the Guest OS

7 Install VMware Tools if needed

8 Copy Startup Files into the Guest OS to the `/etc/rc.d/` directory

9 Export a New OVF Template

10 Deploy the OVF template in a production environment

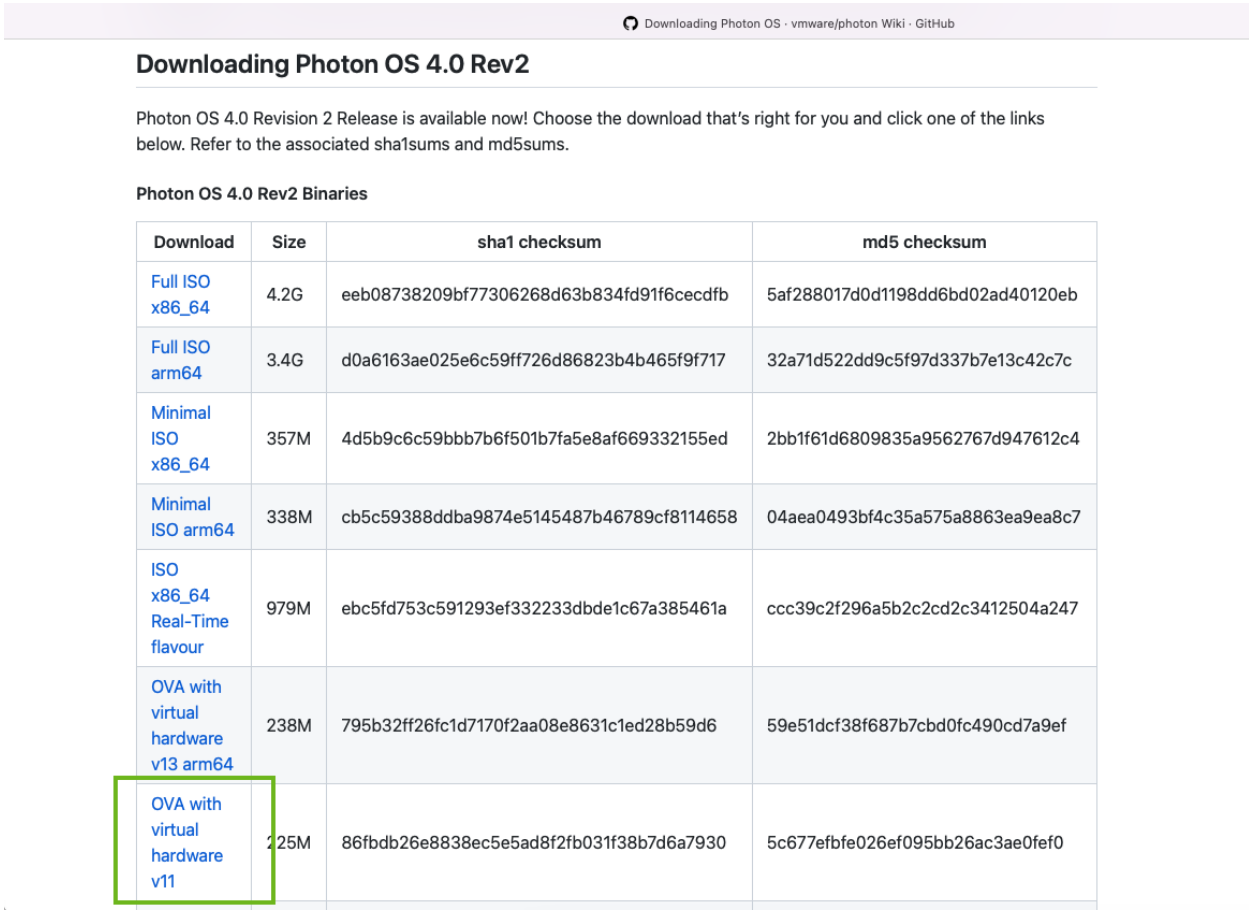
When a user deploys a virtual machine from the finished OVF template, the virtual machine starts the plug-in web server in the guest OS and registers it with vCenter Server.

Acquire the OVA File for Photon OS

Photon OS is an open-source, security-hardened, enterprise-grade Linux distribution designed for Cloud and Edge applications.

You can download a Photon OS OVA with virtual hardware v11 from https://github.com/vmware/photon/wiki/Downloading-Photon-OS

Figure 9-2. Download Photon OS from GitHub



Install the OVA in a vSphere Environment

You need to install the virtual machine in an environment where you can modify its configuration and make some changes in the guest OS. To install the OVA, use the Deploy OVF Template wizard in the vSphere Client.

1 Right-click any inventory object that is a valid parent object of a virtual machine, such as a data center, folder, cluster, resource pool, or host.

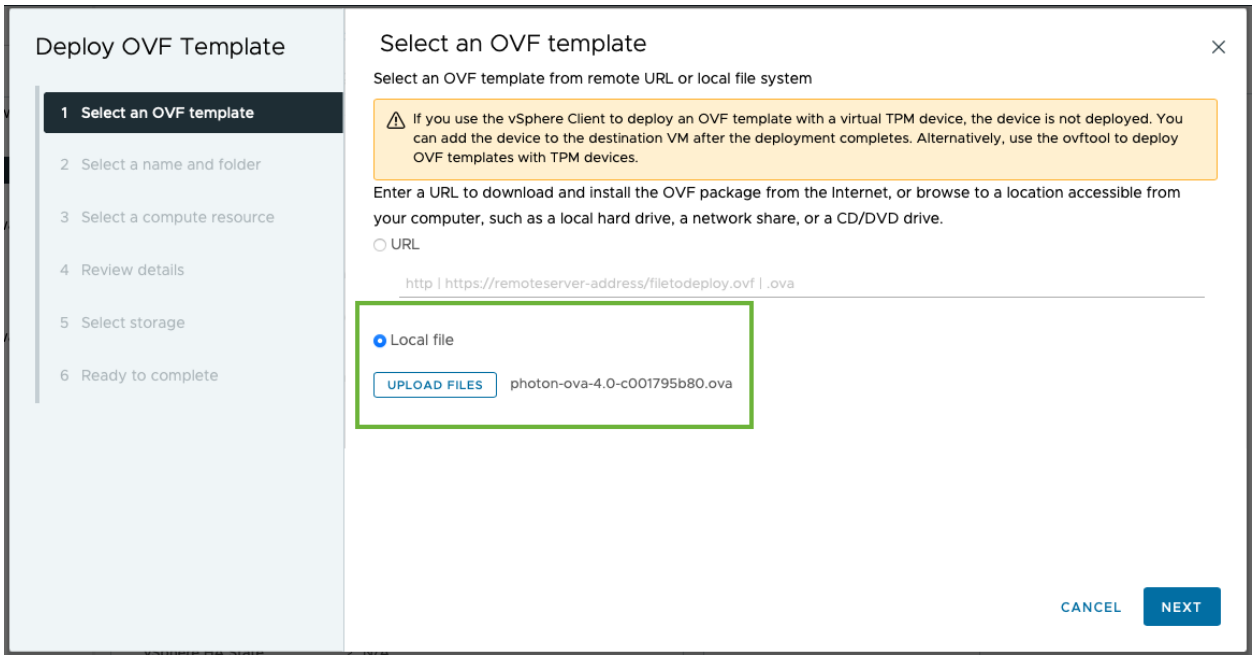
2 Select **Deploy OVF Template**.

3 In the first step of the wizard, **Select an OVF template**, choose **Local file**.

4 Click **UPLOAD FILES** and choose the OVA file you downloaded.

5 Click **NEXT** and complete the Deploy OVF Template wizard.

Figure 9-3. The Deploy OVF Template wizard



Configure vApp Options

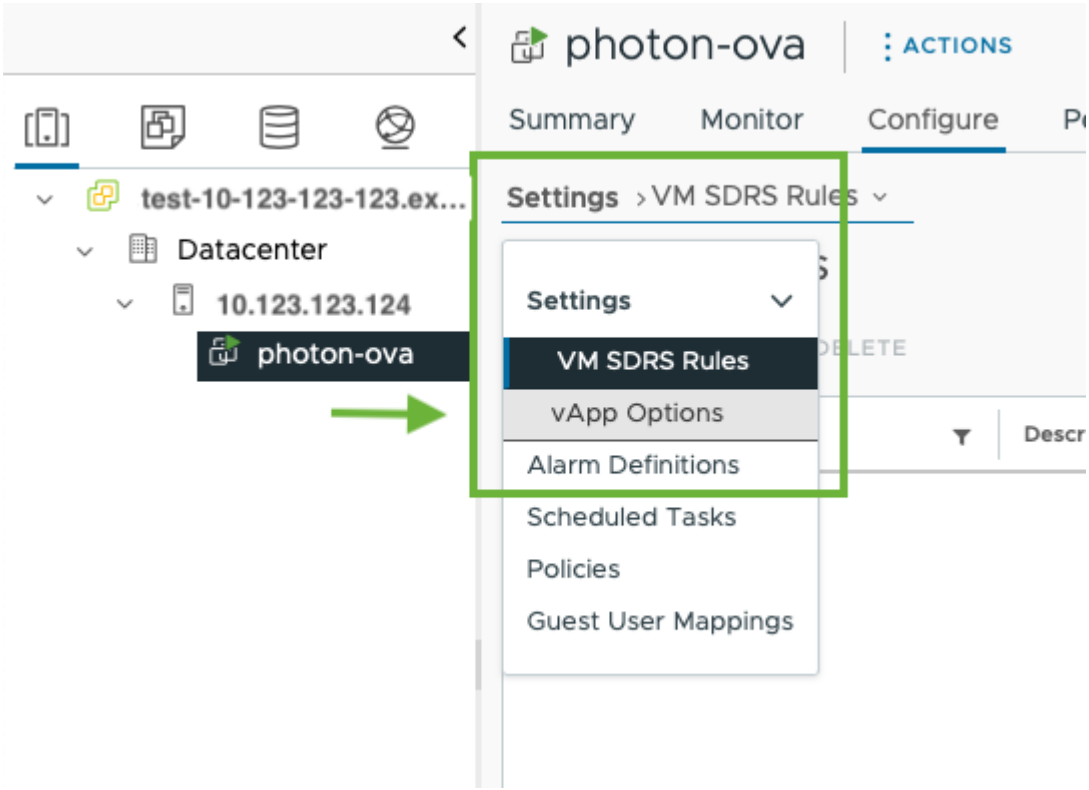
After you install the virtual machine in your development environment, you configure certain vApp properties that will affect the virtual machine when it is deployed to a production environment.

1 Navigate to the deployed virtual machine name in the navigation pane on the left.

2 Click the **Configure** tab.

3 Select **Settings** > **vApp Options**.

Figure 9-4. Settings > VApp Options



4 Click the **EDIT** button near the top of the VApp Options panel.

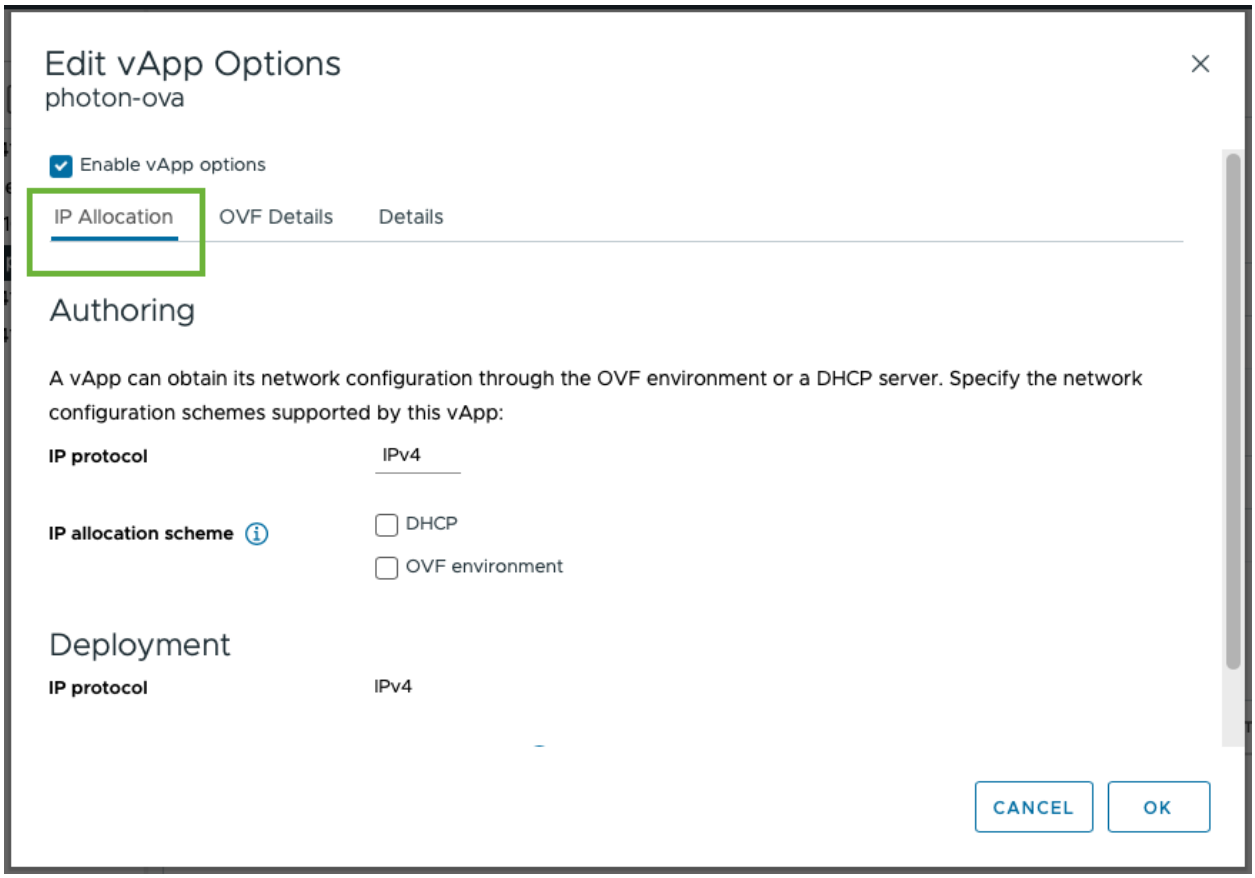
The Edit vApp Options panel contains three tabs: IP Allocation, vApp Details, and Details.

On the IP Allocation tab, select network constraints for the virtual machine.

- In the Authoring section, choose one or both of the network protocols (IPv4, IPv6, or both) for the vApp production environment. The choices you make in the Authoring section constrain the choices available in the Deployment section.

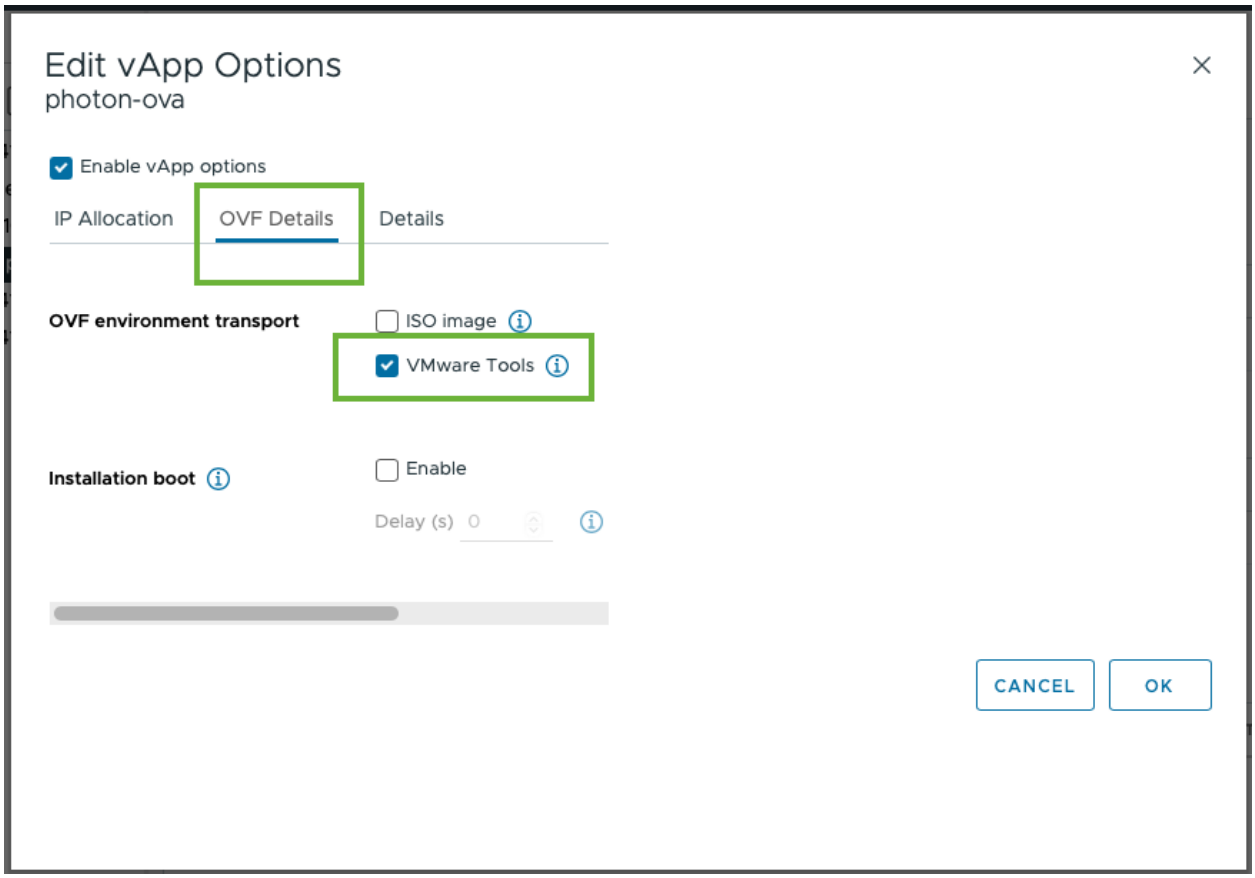
If you select OVF Environment in the Authoring section...	Choose IP pool settings in the Deployment section.
If you choose DHCP in the Authoring section...	DHCP service must be available on the vSphere networks where the vApp will be deployed.
If you make no selections in the Authoring section...	IP addresses must be assigned manually when the vApp is deployed.

Figure 9-5. Selecting IP Allocation Choices



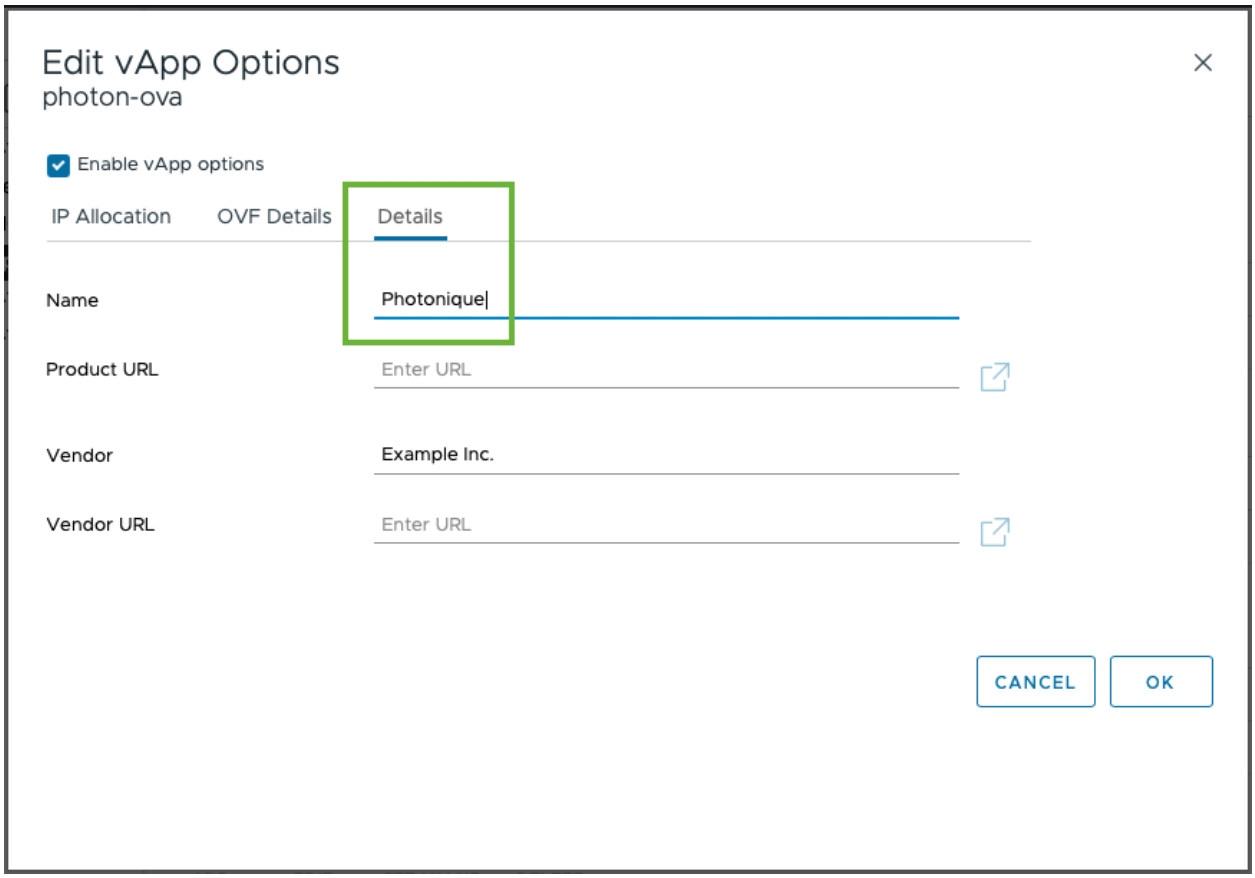
On the **OVF Details** tab of the Edit vApp Options window, in the **OVF environment transport** section, enable VMware Tools. When the virtual machine is deployed in a production environment, it will use VMware Tools to read solution properties that were configured in the OVF template. These properties will be injected into the plug-in registration request.

Figure 9-6. Selecting OVF Details



On the **Details** tab of the Edit vApp Options window, specify a vendor-supplied product name for the plug-in solution.

Figure 9-7. Entering Vendor Information



Configure Solution Properties in the OVF

When the virtual machine starts as a plug-in in a production environment, it needs to register itself with vCenter Server. Before you deploy the plug-in, you must configure the OVF template with several solution properties that will be injected into the registration record during startup. These include the solution ID, the plug-in ID, and other properties.

1 Navigate to the deployed virtual machine name in the navigation pane on the left

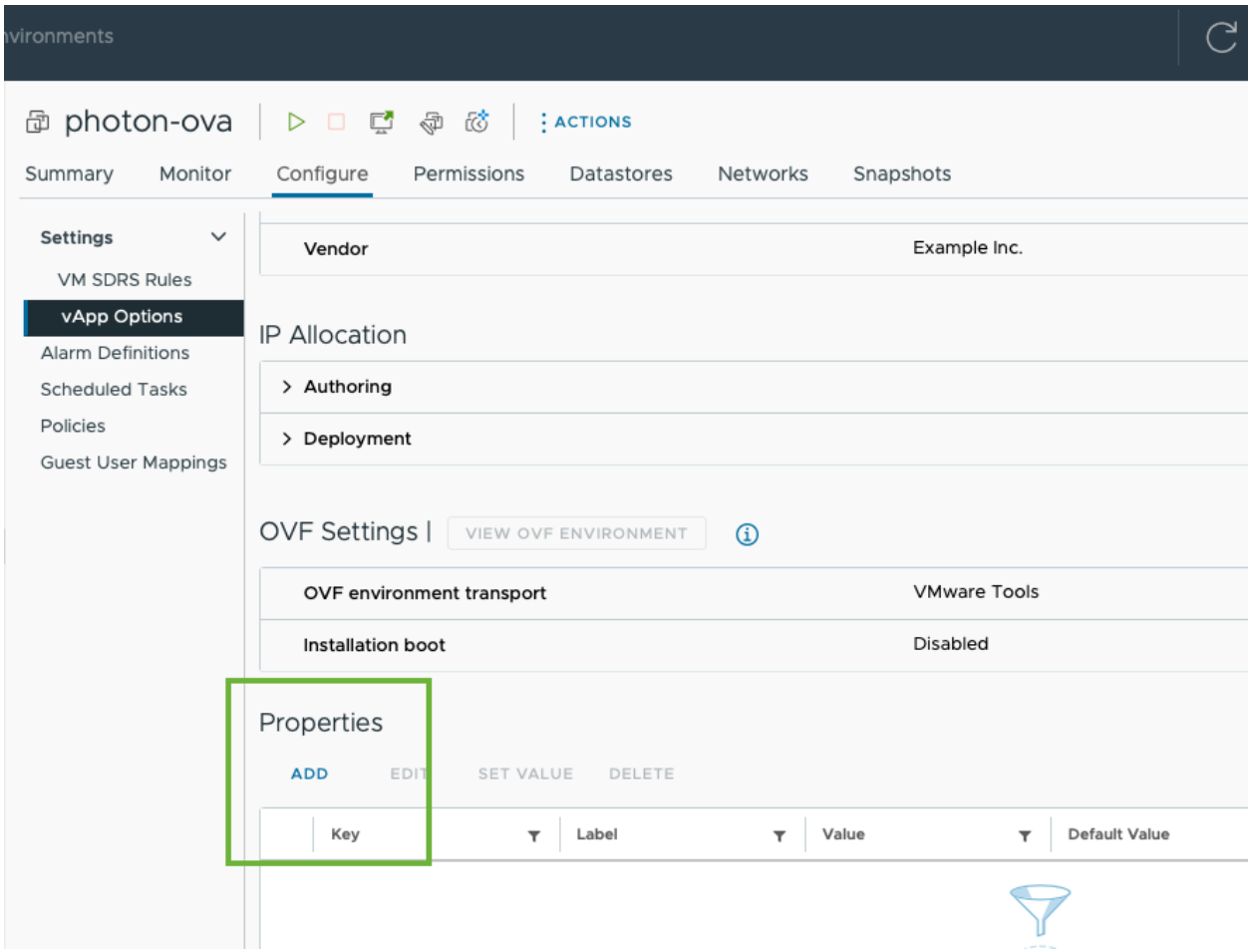
2 Power off the virtual machine if it is powered on.

3 Select the **Configure** tab in the selection pane on the left.

4 Select **Settings > vApp Options**.

5 In the vApp Options pane, scroll down to the Properties section.

Figure 9-8. Adding VApp Properties



Click **ADD** to add each new property. Configure the OVF environment with required properties, as shown in the following table. Default values are required, and must match the choice list value.

Note Optional values and best practices are described in Integrated Installer Solution Metadata.

Property Key	Type	Example Choice List	Example Default Value
vmw.vsphereui.solutioninstall.solutionId	string choice	"com.example.solution.id"	com.example.solution.id
Choice list contains a single vendor-defined element. For example: "com.example.solution.id"			
vmw.vsphereui.solutioninstall.pluginId	string choice	"com.example.plugin.id"	com.example.plugin.id
Choice list contains a single vendor-defined element. For example: "com.example.plugin.id". Value may be same as solution ID.			

Property Key	Type	Example Choice List	Example Default Value
vmw.vsphereui.solutioninstall.vCenterSupport	string choice	"Single"	Single
Choice list contains either "Single" or "Multiple" to indicate whether the solution supports one or many vCenter Server instances.			

Note Property keys are case sensitive.

Note The OVF properties will be assigned values the next time the virtual machine is powered on. The values must be assigned before you export the OVF template.

For more information about these properties and other optional properties, see Integrated Installer Solution Metadata.

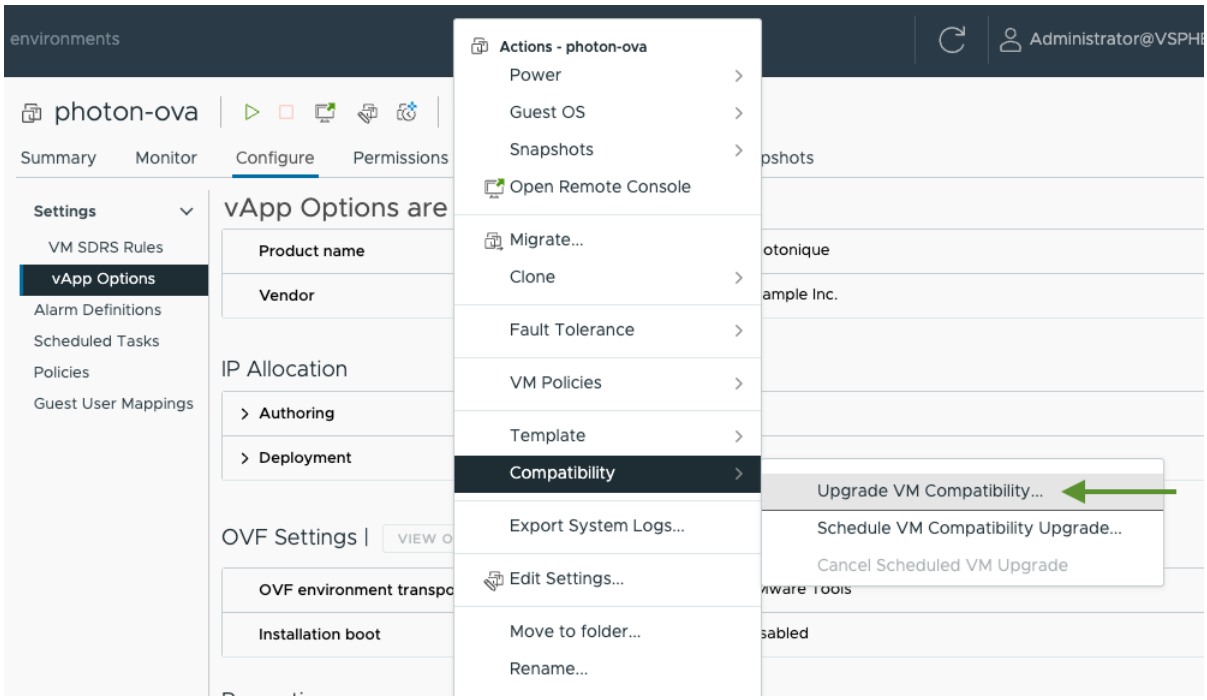
Upgrade VM Compatibility

To be sure that the guest OS will be supported by the host when it is deployed, upgrade compatibility to **ESXi 6.7 Update 2 and later**.

1 Power off the virtual machine, if running.

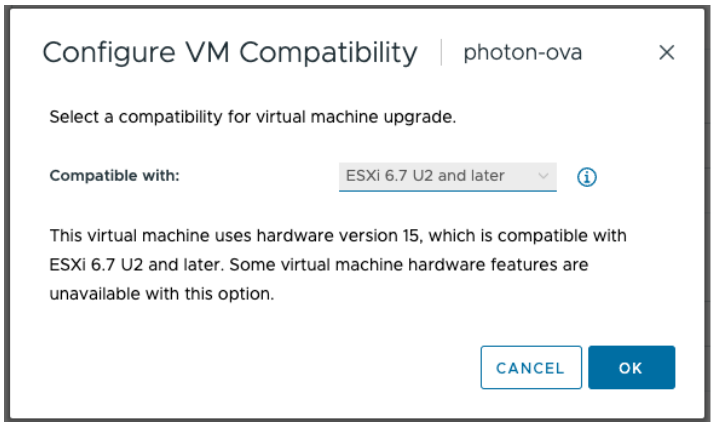
2 From the ACTIONS menu, choose **Compatibility > Upgrade VM Compatibility**.

Figure 9-9. Upgrading Compatibility for the Virtual Machine



3 In the Configure VM Compatibility dialog, choose **ESXi 6.7 U2 and later** from the **Compatible with** pop-up menu.

Figure 9-10. Upgrade VM Compatibility Dialog



Set the `root` Password in the Guest OS

When you download the OVA, the root password for the guest OS is insecure. You must change the password before you deploy the virtual machine in a production environment.

1 Power on the virtual machine.

2 Open a Web Console or Remote Console window to the guest OS.

- The default root password for the Photon OS guest OS is `changeme`.

3 Set a new, secure, root password.

- Make a note of the new password, in case you need to update the software or alter the configuration in the future.

Install Your Web Application Server in the Guest OS

The virtual machine needs a plug-in server to handle REST queries from the plug-in user interface. You can install the SDK sample starter or other web server, according to your preference.

To install the remote plug-in sample starter provided in the vSphere Client SDK:

1 Customize the sample starter as needed for your solution.

2 Build the remote-plugin-sample-starter from the SDK.

See Build the Remote Plug-in Sample Starter.

3 Copy the remote-plugin-sample-starter-*VERSION*.jar into the guest OS. (for this example, we use the /root/ folder of the VM).

```
# cd /root
# scp myuserid@mydevbox.example.com:/html-client-sdk/samples/remote-plugin-sample-starter/
target/*.jar .
```

Install VMware Tools

If you are using the Photon OS OVA downloaded from github as described in the first step of this procedure, VMware Tools is already installed in the guest. Otherwise, you might need to install VMware Tools manually. In the vSphere Client Hosts and Clusters view, right-click your virtual machine and select **Guest OS > Install VMware Tools**. For more detailed instructions, see the knowledge base article https://kb.vmware.com/s/article/2004754.

Copy Startup Files into the Guest OS

When the virtual machine starts up in a production environment, it needs to run startup scripts to register itself with vCenter Server. You must install the scripts in the Linux guest OS before you deploy the virtual machine in a production environment.

1 Copy the files `rc.local` and `vsphere_ui_request.py` from the SDK directory `/samples/solutioninstall/` to the directory `/etc/rc.d/` in the guest OS.

```
# cd /etc/rc.d
# scp myuserid@mydevbox.example.com:/html-client-sdk/samples/solutioninstall/rc.local .
# scp myuserid@mydevbox.example.com:/html-client-sdk/samples/solutioninstall/
vsphere_ui_request.py .
```

2 Set execute permissions on the files in the guest OS.

```
# chmod +x /etc/rc.d/rc.local
# chmod +x /etc/rc.d/vsphere_ui_request.py
```

When you deploy the virtual machine as a plug-in in vCenter Server, the guest OS runs `rc.local`, which in turn runs `vsphere_ui_request.py`. The request script uses VMware Tools to read the solution properties, then builds a registration request and sends it to vCenter Server.

If you install a web server other than the remote plug-in sample starter, you can customize `rc.local`, which is a bash script, or write your own script along similar lines. The script does the following actions:

- Reads solution metadata
- Installs a JDK
- Opens a firewall port for incoming HTTPS requests
- Starts the web server
- Launches `vsphere_ui_request.py` to send the plug-in registration request

Note: If you customize the script, it is important that the vCenter Server thumbprint is verified as part of making the request to the vSphere UI.

Export a New OVF Template

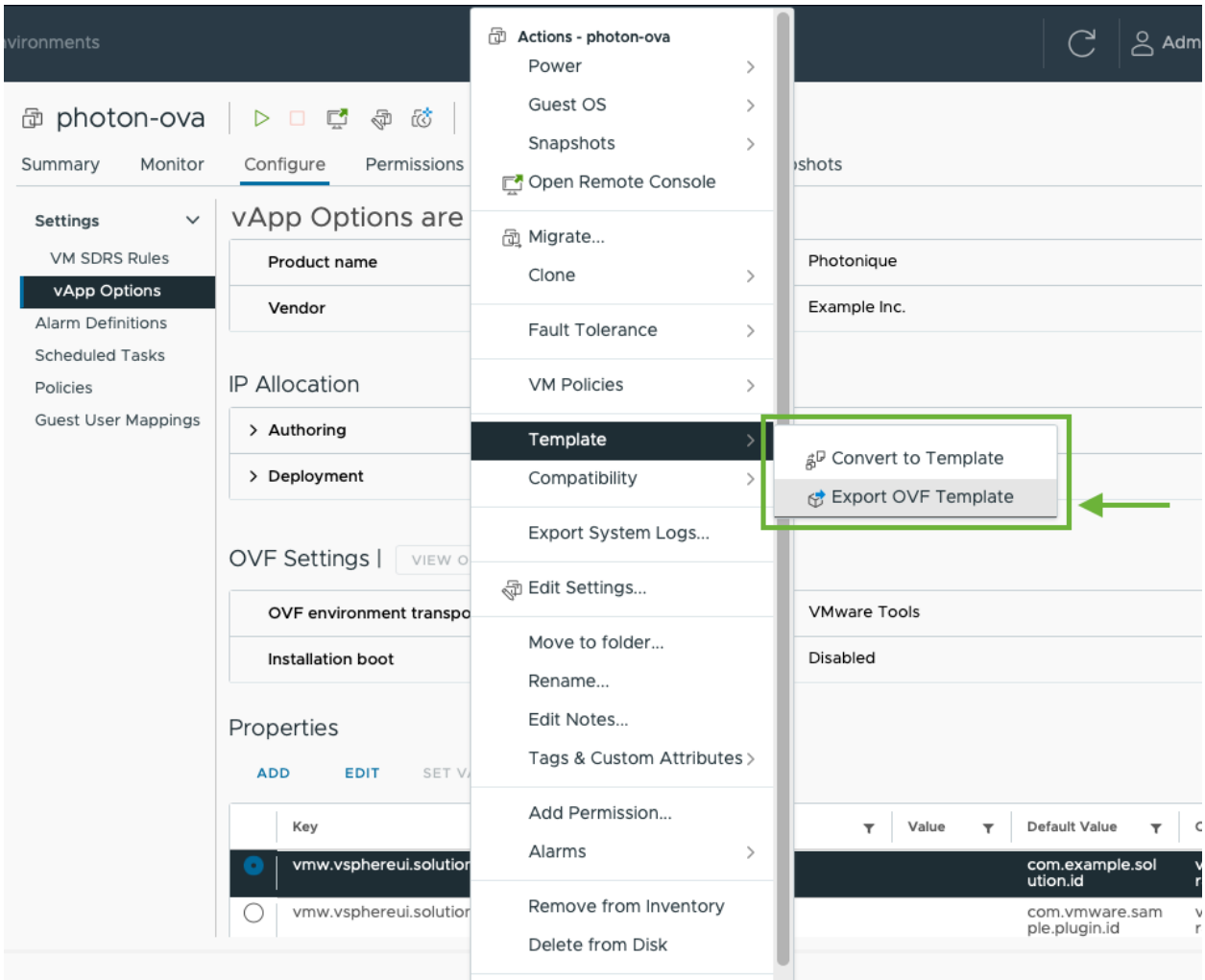
After you configure VMware Tools, install the startup script, configure injectable solution properties, and install your plug-in web server, you can export the virtual machine as an OVF template that is available to deploy in your production environment.

1 Power off the virtual machine.

2 Export the virtual machine as an OVF, using the **Template > Export OVF Template** action.

In the Export OVF Template dialog, supply a name and description for the template.

Figure 9-11. Exporting an OVF Template



Related Resources

- PhotonOS documentation: https://vmware.github.io/photon/docs/
- nginx downloads: https://hub.docker.com/_/nginx/

Deployment with the Integrated Solution Installer

After you complete the configuration steps and export the virtual machine as an OVF template, it is ready to install as a plug-in server in your production environment.

Deploy the OVF template in a production environment

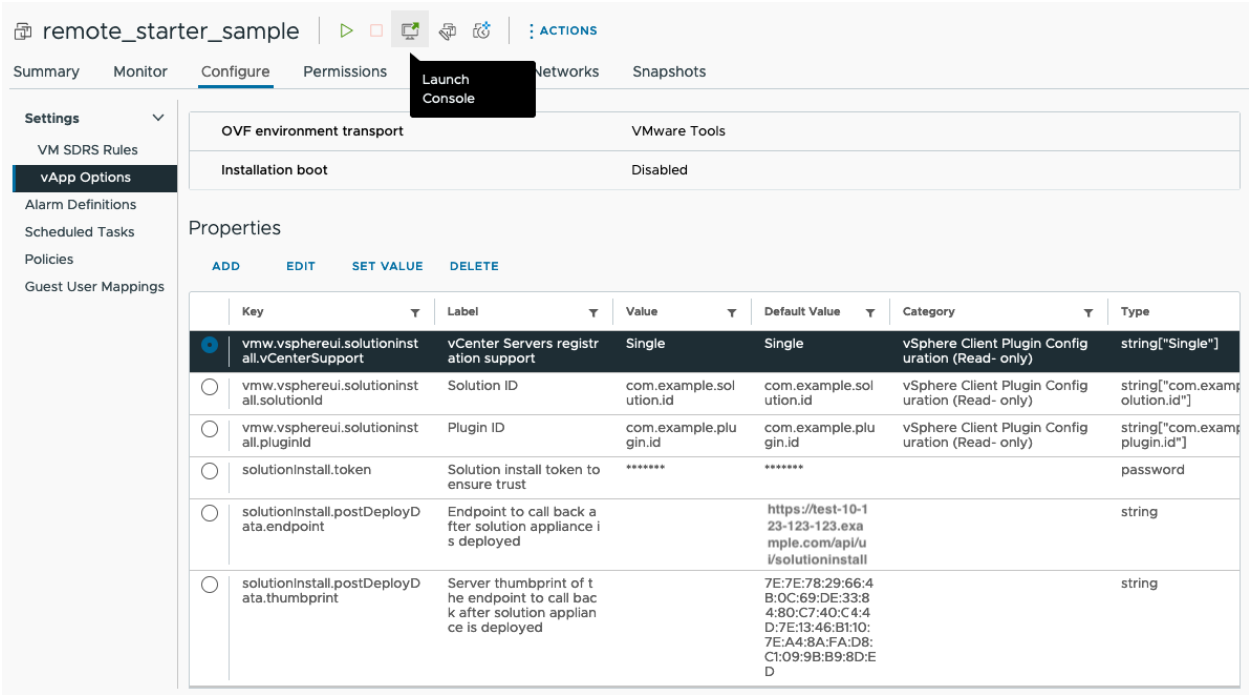
1 If you prepared the virtual machine in a development environment, copy the OVF file to a location accessible to your production system.

2 Use the vSphere Client, connected to your production system, to add the OVF file to your plug-in deployment.

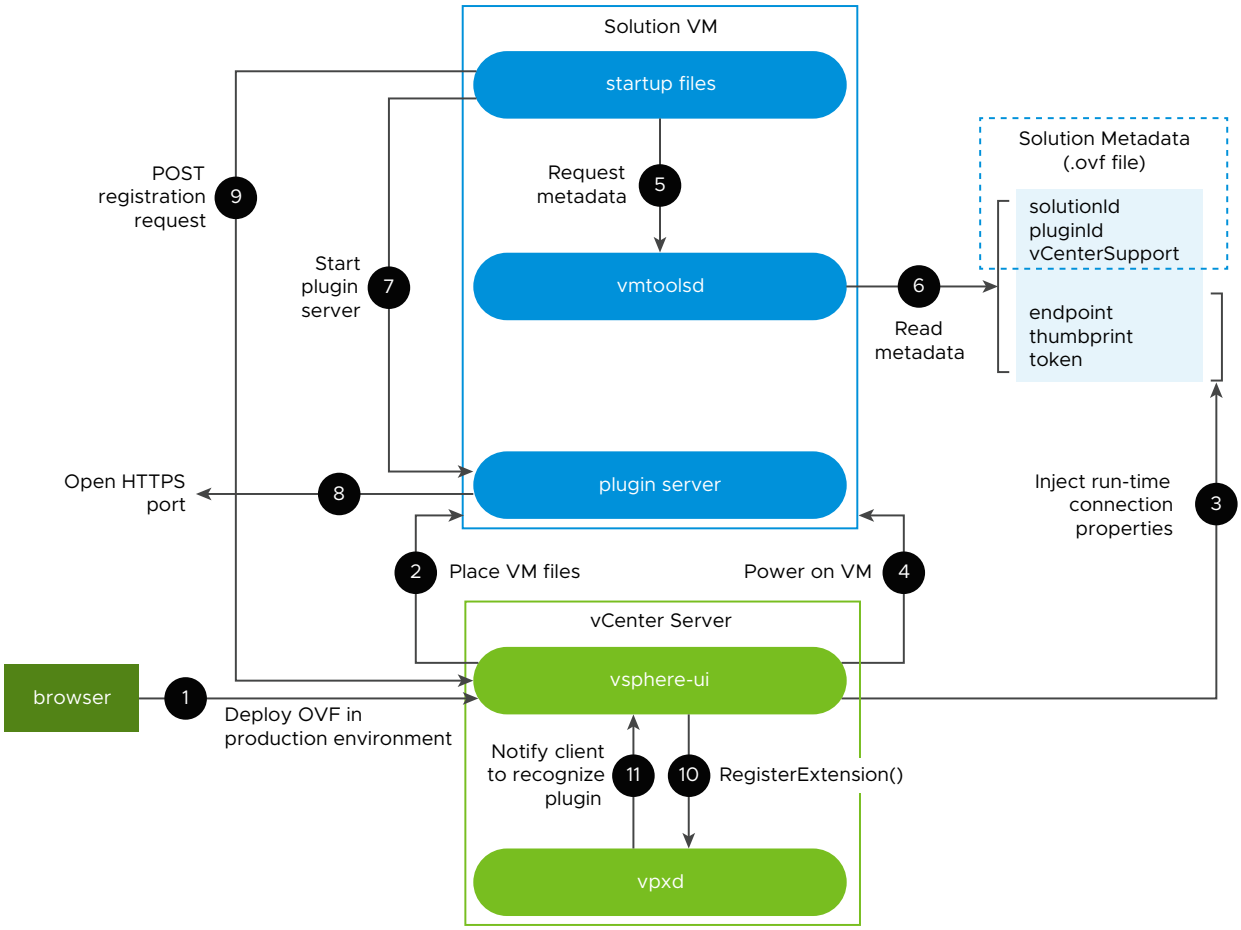
- Navigate to **Administration > Client Plugins** and click the ADD button to install and launch the plug-in in the vCenter Server environment.

3 There is a specific solution installation task in the Tasks console. As of vSphere 8.0 U2, the specific solution install task shows its progress and can be cancelled.

Figure 9-12. Remote Sample Starter Plug-in After Deployment



When the plug-in server deploys to the destination compute resource, the vSphere Client inserts additional connection properties into the OVF. When the virtual machine starts up, these properties enable the plug-in server to connect and authenticate with vCenter Server for the self-registration operation.



Injected OVF Properties

Here are properties injected into the OVF:

- `solutionInstall.token` – Solution install token to ensure trust. A simple base64 token that is issued by vSphere UI, cached and then validated upon receiving a request from the newly deployed Solution Manager. It has 30 minutes timeout after the VM has been deployed and is one-time use.
- `solutionInstall.postDeployData.endpoint` – Endpoint to call back after solution appliance is deployed.
- `solutionInstall.postDeployData.thumbprint` – Server thumbprint of the endpoint to call back after solution appliance is deployed. Must be verified when making a call to the endpoint.

Since vSphere 8.0 U2 (`<index>` = zero indicates first vCenter and increases to N-1):

- `solutionInstall.targetVcenterServers.count` – number of associated vCenter Servers
- `solutionInstall.targetVcenterServers.<index>.hostname` – vCenter FQDN

- `solutionInstall.targetVcenterServers.<index>.guid` – vCenter GUID
- `solutionInstall.targetVcenterServers.<index>.thumbprint` – vCenter thumbprint

As of vSphere 8.0 U3:

- `solutionInstall.targetVcenterServers.<index>.version` – vCenter version number
- `solutionInstall.postDeployData.version` – Version of the endpoint to call back after solution appliance is deployed (vSphere Client version).

Solution Install API

The solution's appliance VM is responsible for providing information to the vSphere Client about its plug-in extension. Once the special parameters are injected and the VM is powered on, the vSphere Client waits to receive a call from the solution appliance on the `solutionInstall.postDeployData.endpoint` (that is, *vCenterIP*`/api/ui/solutioninstall`). The `solutionInstall.token` should also be present in the headers (`solution-install-token header`) of the request. Endpoint definition:

```
POST /api/ui/solutioninstall
```

Header	Value	Summary
solution-install-token	String	One-off base64-encoded token issued by vSphere UI, used to establish trust

Body payload JSON follows. For more details, see data object `Extension` in the *Web Services API Reference*. All fields down to `serverThumbprint` are required, including ones not so listed in the API reference, such as `company`, `client`, and `server`. Description `summary` is displayed in the Client Plug-ins view; `label` is currently not used. Server `type` and `adminEmail` are optional. If the solution should be available in the Solution Manager UI, set `shownInSolutionManager` true. Format of `privilegeList` and `resourceList module` are as prescribed in the `Extension` data object. If you want to add use cases and payload types later, set `additionalProperties` true.

```
{
  "vcenterExtensions": [
    {
      "extension": {
        "key": "<plugin id>",
        "version" : "<plugin version in format a.b.c.d>",
        "company" : "<company name>",
        "description" : {
          "label": "<plugin extension name>",
          "summary": "<plugin extension summary>"
        },
        "client": [
          { "type" : "vsphere-client-remote",
            "url": "<plugin manifest uri>" },
          ...
        ],
        "server": [
          { "url": "<server1 uri>",
```

```
            "serverThumbprint": "<server1 thumbprint>",
            "type": "<server purpose type>"
            "adminEmail": "<email>" },
          ...
        ],
        "shownInSolutionManager": <true|false>,
        "faultList": [
          { "faultID": "<fault-id>" },
          { "faultID": "<fault-id>" }
        ],
        "taskList": [
          { "taskID": "<task-id>" },
          { "taskID": "<task-id>" }
        ],
        "eventList": [
          { "eventID": "<event-id>" },
          { "eventID": "<event-id>" }
        ],
        "privilegeList": [
          { "privID": "<priv-group-name>.<priv-id>",
            "privGroupName": "<priv-group-name>" },
          { "privID": "<priv-group-name>.<priv-id>",
            "privGroupName": "<priv-group-name>" }
        ],
        "resourceList": [
          { "data": [
              { "key": "key-1", "value": "value-1" },
              { "key": "key-2", "value": "value-2" } ],
            "locale": "EN_US",
            "module": "<module>"
          }
        ]
      },
      ...
      additionalProperties: true
    }
  ]
}
```

As of vSphere 8.0 U3, body payload `vcenterExtensionsPerGuid` indicates which extensions to register for a given vCenter GUID. You can specify a list of extensions for each vCenter and have different extensions for every vCenter if desired. Be sure to provide extensions for every vCenter that a user can select. Body payload `vcenterExtensionsPerGuid` extends `vcenterExtensions` – do not use both. Similar format is encapsulated in GUID designation.

```
{
  "vcenterExtensionsPerGuid": {
    "vc-guid01": [
      "extension" section similar to vcenterExtensions
    ]
    "vc-guid02": [
```

```
      "extension" section like vcenterExtensions
    ]
  }
}
```

Integrated Installer Solution Metadata

When you prepare an OVF for a plug-in solution, the metadata available to the virtual machine includes solution properties that control how the virtual machine registers itself with a vCenter Server instance. These properties, with their subproperties and recommended practices, are as follows.

vmw.vsphereui.solutioninstall.solutionId

Subproperty	Required?	Value	Best Practice
Category	no	vendor-defined	vSphere Client Plugin Configuration (read-only)
	Purpose: Distinguish groups of related properties		
Label	no	vendor-defined	`Solution ID`
	Purpose: User-friendly property label		
Key class ID	no		
Key ID	Fixed value	`vmw.vsphereui.solutioninstall.solutionId`	
	Purpose: Unique property identifier		
Key instance ID	no		
Description	no	vendor-defined	Include vendor identification
	Purpose: User-friendly brief description of solution purpose		
Static property	yes	true (checked)	
	Purpose: Distinguish a fixed property of the solution		
Type	Fixed value	`String choice`	
	Purpose: Data format		
User configurable	Fixed value	true (checked)	

Subproperty	Required?	Value	Best Practice
Choice list	yes	vendor-defined	Reverse domain name notation, in quotes, single value
	Purpose: Uniquely identify solution (for single plug-in solution can be same as plug-in ID)		
Default value	yes	vendor-defined	Value from choice list
	Purpose: Assign ID value to deployed solution		

vmw.vsphereui.solutioninstall.pluginId

Subproperty	Required?	Value	Best Practice
Category	no	vendor-defined	vSphere Client Plugin Configuration (Read-only)
	Purpose: Distinguesh groups of related properties		
Label	no	vendor-defined	`Plugin ID`
	Purpose: User-friendly property label		
Key class ID	no		
Key ID	Fixed value	`vmw.vsphereui.solutioninstall.pluginId`	
	Purpose: Unique property identifier		
Key instance ID	no		
Description	no	vendor-defined	Include vendor identification
	Purpose: User-friendly brief description of plug-in or solution purpose		
Static property	yes	true (checked)	
	Purpose: Distinguish a fixed property of the solution		
Type	Fixed value	`String choice`	
	Purpose: Data format		
User configurable	Fixed value	true (checked)	
Choice list	yes	vendor-defined	Reverse domain name notation, in quotes, single value

Subproperty	Required?	Value	Best Practice
	Purpose: Uniquely identify plug-in		
Default value	yes	vendor-defined	Value from choice list
	Purpose: Assign ID value to deployed plug-in		

vmw.vsphereui.solutioninstall.vCenterSupport

Subproperty	Required?	Value	Best Practice
Category	no	vendor-defined	vSphere Client Plugin Configuration (Read-only)
	Purpose: Distinguesh groups of related properties		
Label	no	vendor-defined	`vCenter Servers registration support`
	Purpose: User-friendly property label		
Key class ID	no		
Key ID	Fixed value	`vmw.vsphereui.solutioninstall.vCenterSupport`	
	Purpose: Unique property identifier		
Key instance ID	no		
Description	no	vendor-defined	Include link to solution documentation
	Purpose: User-friendly brief description of plug-in topology		
Static property	yes	true (checked)	
	Purpose: Distinguish a fixed property of the solution		
Type	Fixed value	`String choice`	
	Purpose: Data format		
User configurable	Fixed value	true (checked)	
Choice list	yes	`'Single'` or `'Multiple'`	Specify `'Multiple'` if shared server-side context or low request volume
	Purpose: `'Single'` restricts plug-in to registering with a single vCenter Server instance		

Subproperty	Required?	Value	Best Practice
Default value	yes	vendor-defined	Value from choice list
	Purpose: Assign chosen value to deployed plug-in		

vmw.vsphereui.solutioninstall.requiredVcVersion

Note As of vSphere 8.0 U1. Neither vSphere 8.0 nor vSphere 7.0 U3 support this property.

(optional property)

Subproperty	Required?	Value	Best Practice
Category	no	vendor-defined	vSphere Client Plugin Configuration (Read-only)
	Purpose: Distinguish groups of related properties		
Label	no	vendor-defined	`vCenter Servers Minimum Version`
	Purpose: User-friendly property label		
Key class ID	no		
Key ID	Fixed value	`vmw.vsphereui.solutioninstall.requiredVcVersion`	
	Purpose: Unique property identifier		
Key instance ID	no		
Description	no	vendor-defined	Include link to solution documentation
	Purpose: User-friendly brief explanation of reason for minimum version		
Static property	yes	true (checked)	
	Purpose: Distinguish a fixed property of the solution		
Type	Fixed value	`String choice`	
	Purpose: Data format		
User configurable	Fixed value	true (checked)	
Choice list	yes	vendor-defined	
	Purpose: Specify the minimum vCenter Server version that can support this plug-in		

Subproperty	Required?	Value	Best Practice
Default value	yes	vendor-defined	Value from choice list
	Purpose: Assign chosen value to deployed plug-in		

vmw.vsphereui.solutioninstall.legacyExtensionId

(optional property)

Subproperty	Required?	Value	Best Practice
Category	no	vendor-defined	vSphere Client Plugin Configuration (Read-only)
	Purpose: Distinguesh groups of related properties		
Label	no	vendor-defined	Legacy Extension ID
	Purpose: User-friendly property label		
Key class ID	no		
Key ID	Fixed value	`vmw.vsphereui.solutioninstall.legacyExtensionId`	
	Purpose: Unique property identifier		
Key instance ID	no		
Description	no	vendor-defined	Include vendor identification, link to documentation
	Purpose: Brief description of legacy plug-in replacement strategy		
Static property	yes	true (checked)	
	Purpose: Distinguish a fixed property of the solution		
Type	Fixed value	`String choice`	
	Purpose: Data format		
User configurable	Fixed value	true (checked)	
Choice list	yes	vendor-defined	Reverse domain name notation, in quotes, single value
	Purpose: Identify the plug-in that this plug-in replaces; facilitate check for legacy plug-in already installed		

Subproperty	Required?	Value	Best Practice
Default value	yes	vendor-defined	Value from choice list
	Purpose: Assign chosen value to deployed plug-in		

vmw.vsphereui.solutioninstall.preventInstallationWhenLegacyExtension

(optional property)

Subproperty	Required?	Value	Best Practice
Category	no	vendor-defined	vSphere Client Plugin Configuration (Read-only)
	Purpose: Distinguish groups of related properties		
Label	no	vendor-defined	Prevent installation if legacy extension
	Purpose: User-friendly property label		
Key class ID	no		
Key ID	Fixed value	`vmw.vsphereui.solutioninstall.preventInstallationWhenLegacyExtension`	
	Purpose: Unique property identifier		
Key instance ID	no		
Description	no	vendor-defined	Include vendor identification, link to documentation
	Purpose: Brief description of legacy plug-in replacement strategy		
Static property	yes	true (checked)	
	Purpose: Distinguish a fixed property of the solution		
Type	Fixed value	`String choice`	
	Purpose: Data format		
User configurable	Fixed value	true (checked)	
Choice list	Fixed value	`'true'` or `'false'`	
	Purpose: Indicate condition for replacing legacy plug already installed		

Subproperty	Required?	Value	Best Practice
Default value	yes	vendor-defined	Value from choice list
	Purpose: Assign chosen value to deployed plug-in		

Plug-in Properties XML Example

An example of the `<ProductSection>` of an OVF file, containing the three required plug-in properties with example values, follows:

```
    <ProductSection>
      <Info>Information about the installed software</Info>
      <Product>Photonique</Product>
      <Vendor>Example Inc.</Vendor>
      <Version>4.0</Version>
      <FullVersion>4.0</FullVersion>
      <Category>vSphere Client Plugin Configuration (Read-only)</Category>
      <Property ovf:qualifiers="ValueMap{&quot;Single&quot;}" ovf:userConfigurable="true"
ovf:value="Single" ovf:type="string" ovf:key="vmw.vsphereui.solutioninstall.vCenterSupport">
        <Label>vCenter Servers registration support</Label>
        <Description/>
      </Property>
      <Property ovf:qualifiers="ValueMap{&quot;com.example.solution.id&quot;}"
ovf:userConfigurable="true" ovf:value="com.example.solution.id" ovf:type="string"
ovf:key="vmw.vsphereui.solutioninstall.solutionId">
        <Label>Solution ID</Label>
        <Description/>
      </Property>
      <Property ovf:qualifiers="ValueMap{&quot;com.example.plugin.id&quot;}"
ovf:userConfigurable="true" ovf:value="com.example.plugin.id" ovf:type="string"
ovf:key="vmw.vsphereui.solutioninstall.pluginId">
        <Label>Plugin ID</Label>
        <Description/>
      </Property>
    </ProductSection>
```

Advanced Considerations for Remote Plug-in Servers

10

A remote plug-in for the vSphere Client has both a server portion and a user interface portion. You can use any coding language or framework you choose for the server portion. Your plug-in server must generally provide the following functionality:

- A web application server for both plug-in manifest file and plug-in user interface files.
- A fixed Service Provider Interface that responds to vSphere Client requests for dynamic view content such as menus.
- Data access and computation services on behalf of the plug-in user interface, if needed.
- Session cloning service to authenticate with the vCenter Server Web Services API, if needed.

Read the following topics next:

Registering Auxiliary Plug-in Servers

When you register a plug-in with a vCenter Server instance, information about the plug-in server is stored in an `Extension` record with the ExtensionManager. Registration information includes endpoint addresses of the plug-in's primary server and of any auxiliary servers that the plug-in comprises.

The `Extension` record stores descriptors for the plug-in servers in the `server` property, which is an array of `ServerInfo` objects. The first element of the array represents the manifest server. All other array elements represent auxiliary servers, in any order. Several properties of the `ServerInfo` object must be set in specific ways:

type

Contains a service identifier, which is a string value chosen by the developer. The string enables client code to distinguish between auxiliary servers.

A best practice is to assign the value `MANIFEST_SERVER` to the first element of the array, to identify the manifest server for the plug-in.

url

Contains the server endpoint address. The vCenter Server reverse proxy translates the server endpoints into proxy endpoints, which the JavaScript API makes available to front-end code. In the descriptor for the primary server, which also serves the manifest file, the `url` property must contain the full path to the plug-in manifest file. Relative URLs for HTML content resolve relative to the directory that contains the manifest.

Note In the descriptor for an auxiliary server that supplies filter choices for dynamic extensions, the `uri` property must end with a slash ('`/`'). See Configure Dynamic Extensions.

serverThumbprint

Each server descriptor must contain either the thumbprint or the server certificate (below) or both.

serverCertificate

The SSL certificate for the server endpoint in strict PEM format (string). In the plug-in server certificate, the server host name must be specified according to subjectAltName extension (RFC 2818, Section 3.1 Server Identity).

Note When registering a plug-in, it's highly recommended to provide SSL certificates of the plug-in servers in addition to the thumbprints of the SSL certificates of the plug-in servers. Performing a full SSL certificate check during SSL handshake is more secure than performing an SSL certificate thumbprint check. Furthermore, in future vSphere releases, support for SSL certificate thumbprints will be dropped and full SSL certificates will be required.

In addition to the `server` property, the `Extension` record contains a `client` property, which is an array of `ClientInfo` objects. When you register the plug-in, you use the first element of this array as a descriptor for the plug-in as a whole. Three properties of the descriptor must be set in specific ways:

type

Must contain the value `'vsphere-client-remote'`

url

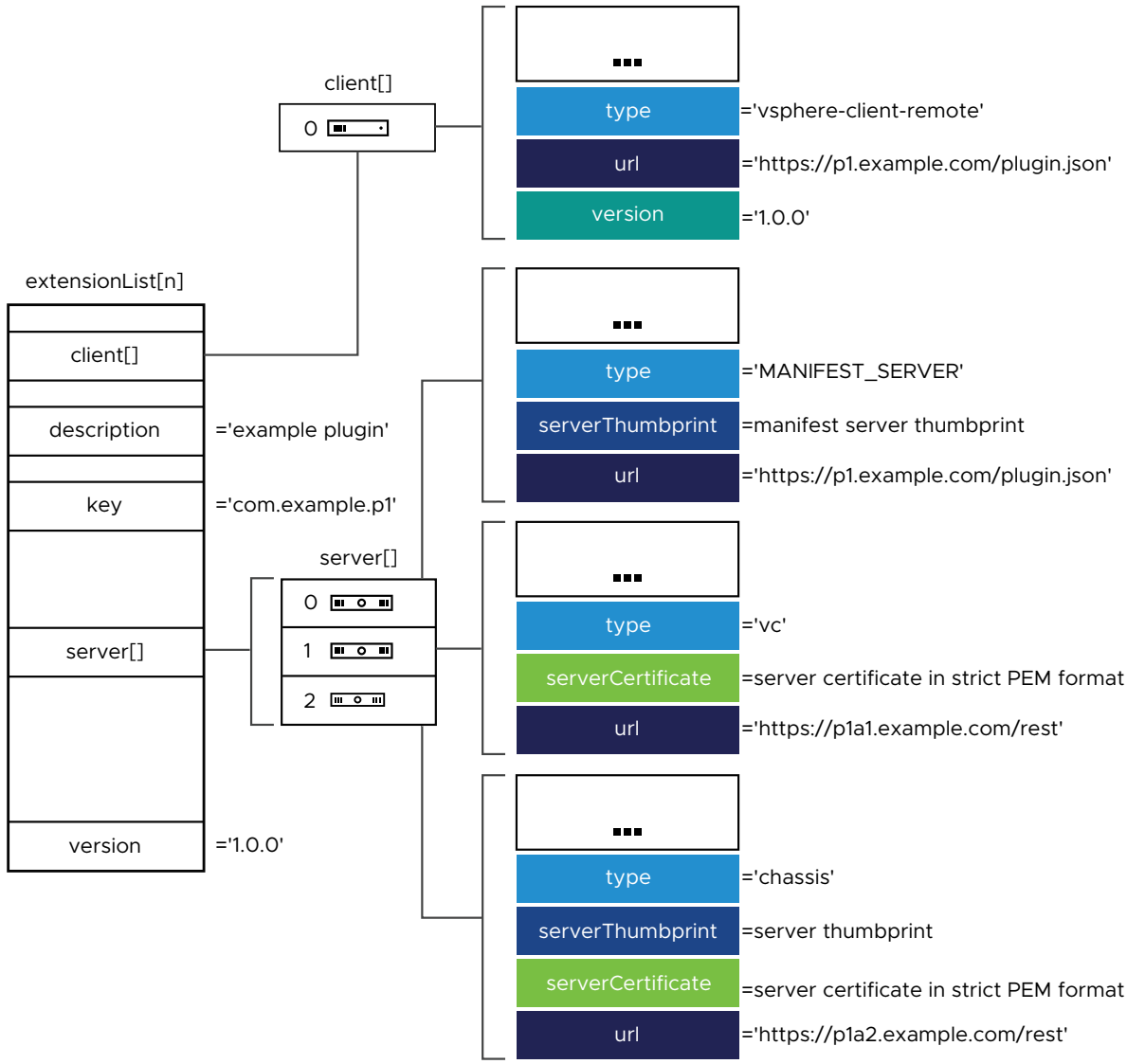
Must contain URL of the plug-in manifest file, which must match the `url` property in the first `ServerInfo` record.

version

Must contain a developer-assigned version string, which distinguishes a new plug-in version from a new plug-in instance of the same version. The version string must be an integer or a series of dot-separated integers: *n*`[.`*n*`[.`*n*`]]`

The following illustration shows some of the basic structure of the Extension record used to register a plug-in.

Figure 10-1. Plug-in Registration Data



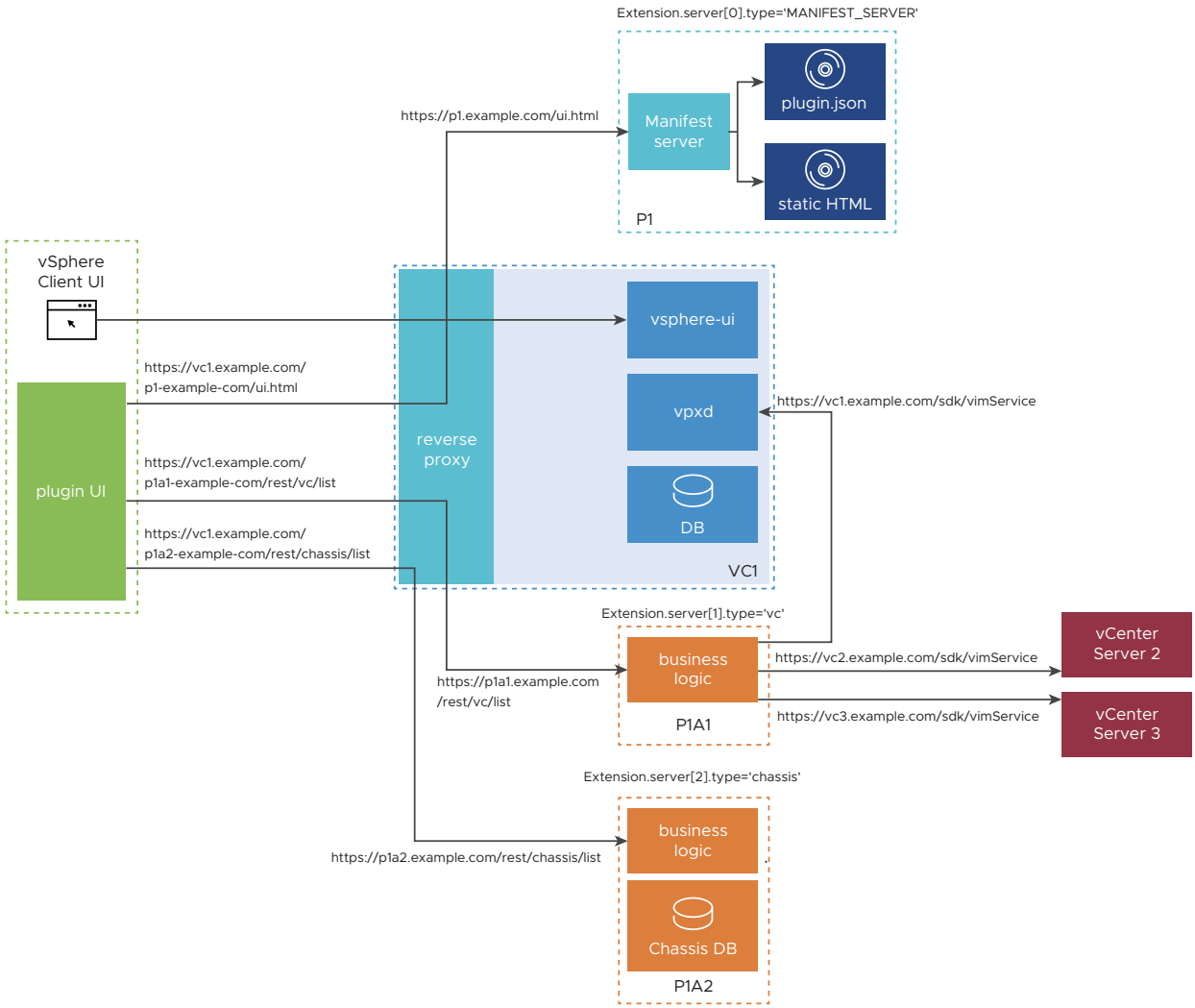
For information about how to register auxiliary plug-in servers by using the registration tool in the SDK, see vSphere Client Plug-in Registration Tool.

Auxiliary Server Data Paths

When a plug-in includes auxiliary servers, the auxiliary servers must be registered with vCenter Server, in the same Extension record as the primary server. When the plug-in user interface sends a request to an auxiliary server, the request passes through the vCenter Server reverse proxy, so that the browser treats it as having the same origin as the primary server.

The following illustration shows a hypothetical use of auxiliary plug-in servers that provide specific services supplemental to the functions of the primary server. The primary server provides the overall plug-in configuration in the manifest file, as well as the HTML components for the plug-in user interface. Separate concerns are handled by the auxiliary servers.

Figure 10-2. Auxiliary Server Data Paths



This hypothetical plug-in implements a Chassis abstraction, which represents a physical enclosure that contains a number of blade servers. The blades run instances of vCenter Server. The plug-in user interface presents a Chassis and its vCenter Server instances as related objects in the inventory.

Auxiliary server 1 (P1A1, in orange) interfaces with the vCenter Server instances and data centers encompassed by the plug-in. The auxiliary server uses the Web Services API to collect data from the vCenter Server instances, which it returns to the plug-in user interface. The plug-in user interface uses instance-specific data to supplement graphic and tabular displays in the UI.

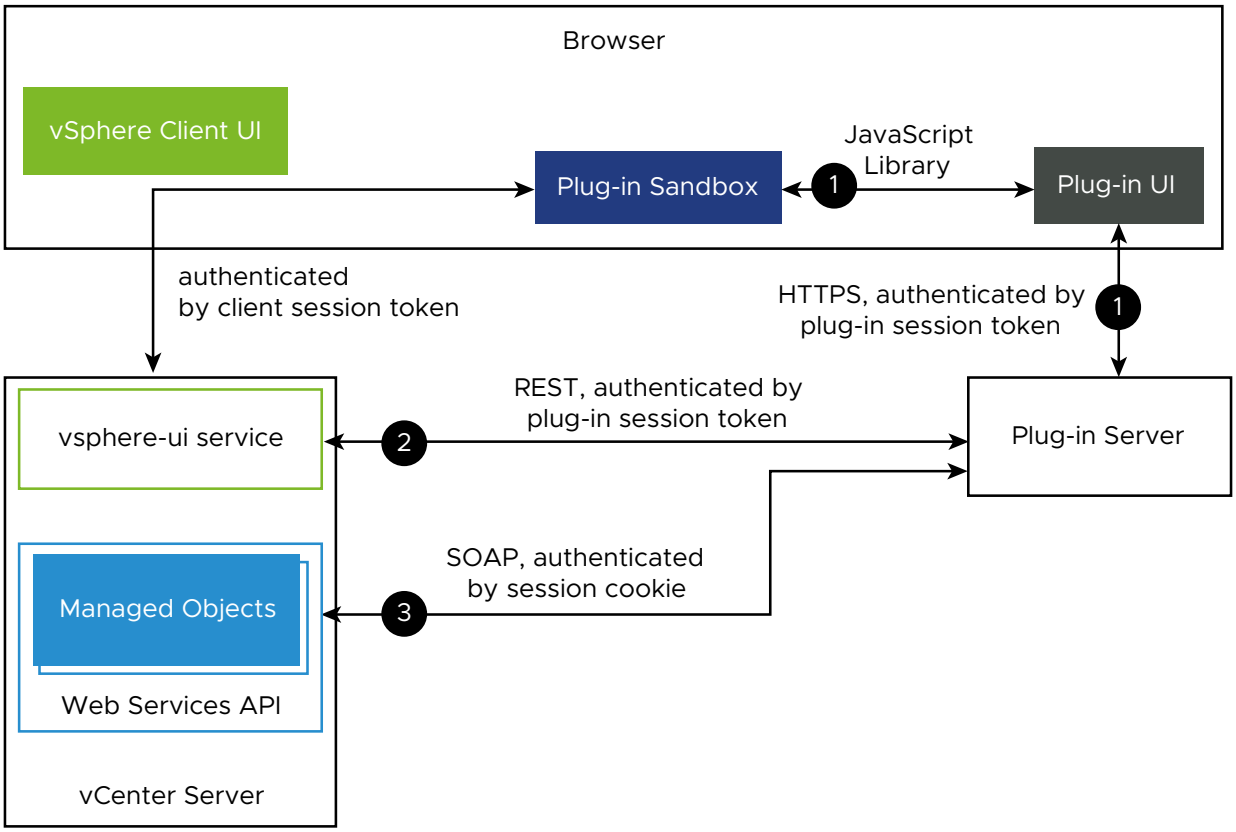
Auxiliary server 2 (P1A2, in orange) maintains a high-performance interface to a database of Chassis objects and links to related objects that represent the physical blades and VMware products that run on them. The plug-in uses this data to fill out a display of Chassis objects with related blades, vCenter Server instances, and other managed entities that populate VMware data centers.

Communication Paths for Authentication in the Remote Plug-in Server

The remote plug-in server operates outside the vCenter Server instance, and must authenticate with the Web Services API to identify and authorize its access to vSphere resources. The authentication procedure requires several steps, summarized below.

The plug-in user interface communicates with the `vsphere-ui` service through a plug-in sandbox in the browser. The plug-in sandbox uses the vSphere Client session token to authenticate with the `vsphere-ui` service in vCenter Server. The plug-in server needs to get a SOAP client session token to authenticate its operations with the Web Services API. The following diagram shows the basic communication paths involved in converting the vSphere Client session token to a plug-in server SOAP session token.

Figure 10-3. Plug-in Server Communication Paths for Authentication



Cloning a session consists of three stages of interactions involving the plug-in server:

1 The plug-in user interface retrieves its session ID and the GUID of the context object, then sends them to the plug-in server.

2 The plug-in server sends a REST request to vCenter Server to acquire a ticket that allows it to clone the user session.

3 The plug-in server sends a SOAP request to vCenter Server to clone the user session and acquire a new session ID.

These interactions are described in more detail in How to Delegate Session Authority to the Plug-in Server.

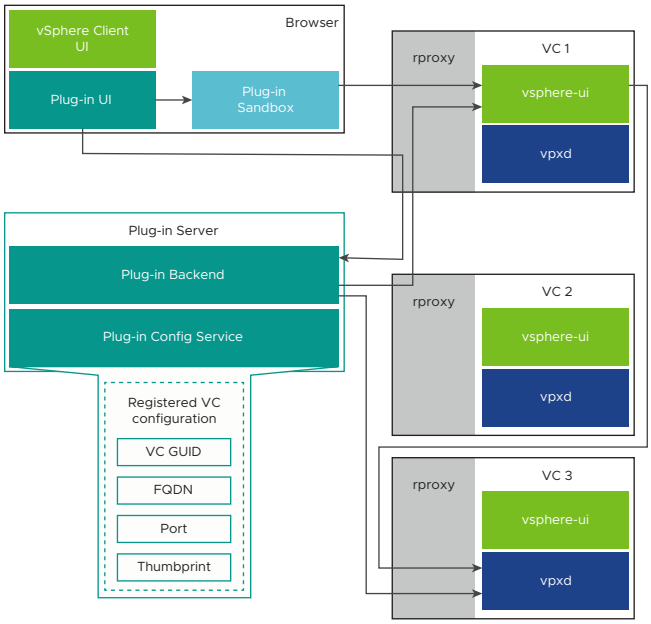
How to Delegate Session Authority to the Plug-in Server

When a plug-in server communicates with vCenter Server, it must authenticate to act on behalf of the user. This procedure shows how to delegate authority from the plug-in client code, running in a browser, to the remote plug-in server.

Before the plug-in server can establish a session with a vCenter Server, it must know the endpoint attributes for that instance. A best practice is to maintain a configuration file, for each plug-in server, that stores the attributes for each vCenter Server instance where the plug-in is registered. For example, a JSON representation of endpoint attributes might look similar to this:

```
{
  "vc1-guid": {
    "fqdn": "vc1.example.com",
    "thumbprint": "vc1-thumbprint",
    "port": 443
  },
  "vc2-guid": {
    "fqdn": "vc2.example.com",
    "thumbprint": "vc2-thumbprint",
    "port": 443
  }
}
```

In its simplest form, the delegation procedure involves communications between a plug-in front end and its sandbox environment, between the plug-in front end and the plug-in back end, and between the back end and an instance of vCenter Server. In some situations, two or three vCenter Server instances participate in the delegation procedure. The following orientation diagram shows all the processes that might work together to delegate session authority securely to the plug-in back end.



Use the following steps to delegate authority from the plug-in front end code to the plug-in server. The plug-in server can use this procedure to establish a session with any vCenter Server instance in the same ELM link group.

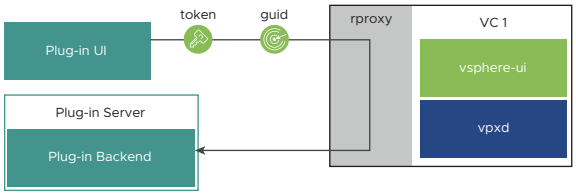
1 The plug-in user interface calls the `app.getSessioninfo()` method in the client JavaScript library, which in turn contacts the plug-in sandbox to request session information. The sandbox returns an object containing a `sessionToken` string, which contains a new plug-in session token that can be used for authentication by the plug-in server. The token represents the authority of the session the user established between the vSphere Client and the vCenter Server instance to which it is connected.

2 The plug-in user interface calls the `app.getContextObjects()` method in the client JavaScript library, which returns the ID of the object currently selected in the vSphere Client. The object ID is appended with the GUID of the vCenter Server instance that manages the object. The vCenter Server GUID is the substring following the last colon separator (:).

Note If a global view is active instead of an object view, `app.getContextObjects()` returns an empty list.

3 The plug-in user interface extracts both the session key and the vCenter Server GUID, then sends them to the plug-in server, using a custom API.

Note If no GUID is available, the plug-in user interface sends only the session key.



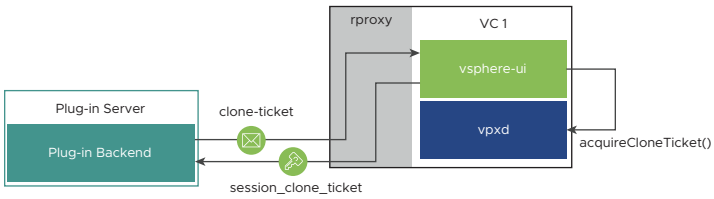
4 The plug-in back end builds a REST request to the vsphere-ui service of a vCenter Server instance. The request contains the following:

- A `POST` verb
- The clone-ticket endpoint path: `/api/ui/vcenter/session/clone-ticket`
- The `Content-type` and `Accept` headers both set to `application/json`.
- A custom header named `vmware-api-session-id`, with the session token as its value
- A JSON object body, containing a `vc-guid` property whose value is the GUID of the vCenter Server instance

The request will look similar to this:

```
POST /api/ui/vcenter/session/clone-ticket
Content-type: application/json
Accept: application/json
vmware-api-session-id: 12345678

{
  "vc_guid": "223b94f2-af15-4613-5d1a-a278b19abc09"
}
```

5 The plug-in back end sends the REST request to the clone-ticket REST endpoint of any vCenter Server instance, by using the endpoint attributes in the plug-in server configuration data.

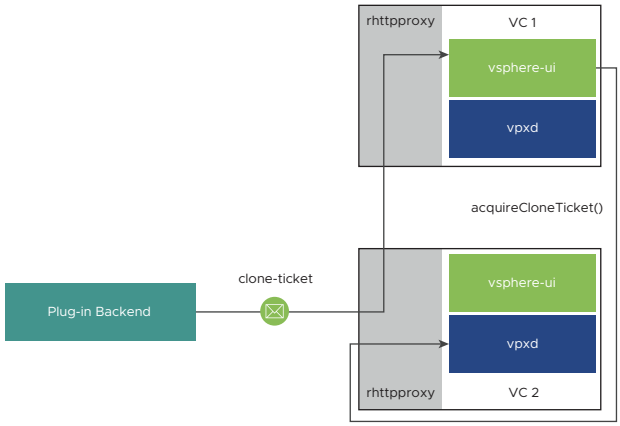


In some cases, this can trigger extra back-end communications before the plug-in back end receives a reply. For example:

a The plug-in back end might be sending the request to a vCenter Server instance that does not hold the user session, particularly if the user's browser is connected to an instance with which the plug-in is not registered. In that case, the reverse HTTP proxy forwards the request to the vCenter Server instance that holds the user session.

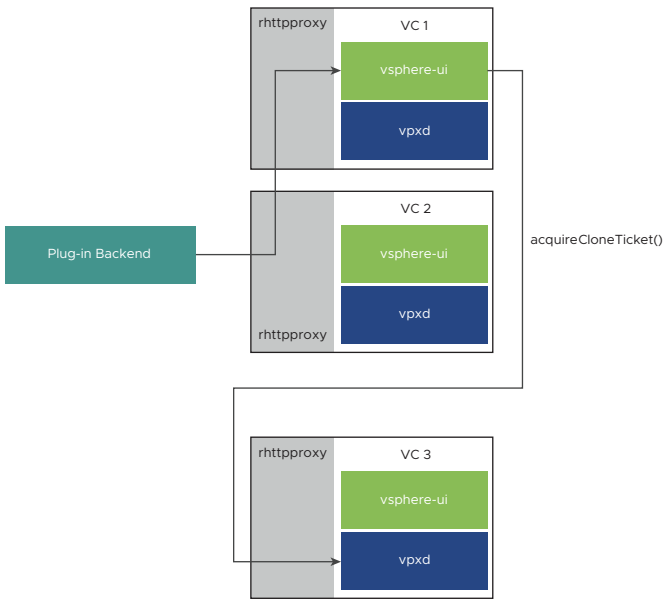
In the following diagram, the browser is connected to VC1, which serves the vSphere Client in the browser. The plug-in back end chooses to contact VC2 with the clone-ticket request. VC2 forwards the request to VC1, which holds the session key.

.

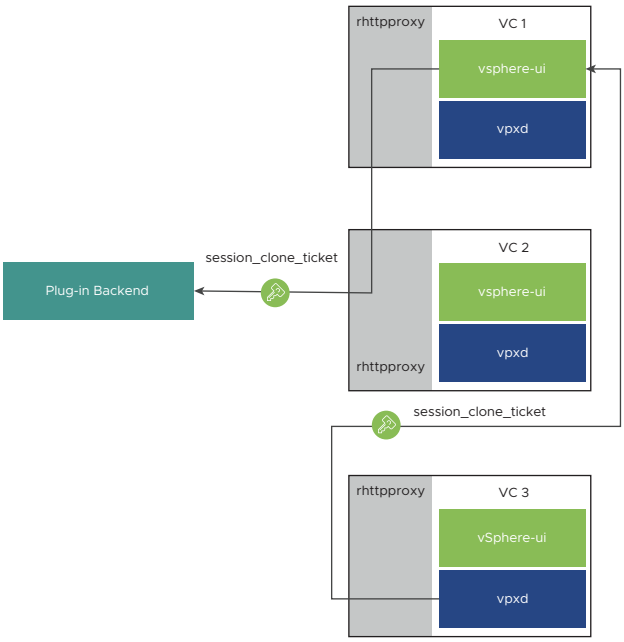


b The context object the user selected might be on a third vCenter Server instance. In that case, the vSphere Client back end (the vsphere-ui process) issues a Web Services API call to the vCenter Server instance identified by the GUID in the clone-ticket request. The vsphere-ui process uses the `acquireCloneTicket()` method of the SessionManager managed object to request delegation of session authority to the process that presents the ticket to the vCenter Server instance.

In the following diagram, VC2 forwards the clone-ticket request to VC1, which holds the session with the vSphere Client in the browser. VC1 discovers that the context object is managed by VC3, so VC1 requests a clone ticket that will allow the plug-in back end to authenticate with VC3.



c The call chain returns the ticket to the plug-in back end. The ticket gives the plug-in back end a means to establish a Web Services session with VC3.



The clone ticket is valid for the Web Services API of the vCenter Server instance associated with the context object. This is a single-use key to authenticate a call to the SessionManager.

The response will look similar to this:

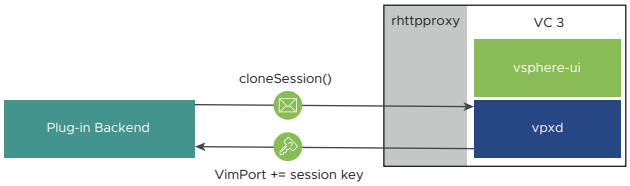
```
{
  "session_clone_ticket": "cst-VCT-82cbd981-5f52-0a67-fe55-d995a7347f82--tp-B6-BC-CB-B8-59-89-C0-F2-E4-F0-C2-91-8F-28-C1-DE-10-5E-24-69"
}
```

6 The plug-in back end constructs a SOAP request to obtain a regular session ID from the Web Services API, by using the `cloneSession()` operation on the Session Manager.

The code for the SOAP request will be similar to this Java example:

```
VimService vimService = new VimService();
VimPortType client = vimService.getVimPort();
ManagedObjectReference siRef = new ManagedObjectReference();
siRef.setType("ServiceInstance");
siRef.setValue("Serviceinstance");
ServiceInstance si = client.createStub(ServiceInstance.class, siRef);
ServiceInstanceContent sic = si.RetrieveContent();
SessionManager mgr = client.createStub(SessionManager.class, sic.getSessionManager());
UserSession wsSession = mgr.cloneSession(sessionCloneTicket);
```

7 The `cloneSession()` method retrieves a new multi-use session key and applies it to the linked `VimPort` object. Subsequent SOAP requests sent by means of the same `VimPort` object authenticate with the new session key.



Response Codes to session/clone-ticket Request

The REST request for a clone-ticket from the vsphere-ui service can produce the following response codes.

Table 10-1. Response Codes to clone-ticket Request

Response Code	Description

201	Session clone ticket was successfully created and returned.
	The body type is `application/json`, and the response body has this format:
	`{`
	`"session_clone_ticket": "string"`
	`}`

401	The request is not properly authenticated.
403	The remote plug-in is not registered with the vCenter Server instance identified by the specified UUID.
404	There is no vCenter Server instance identified by the specified UUID.

Best Practices for vSphere Client Remote Plug-ins

11

Sections below give tips on implementing, localizing, and deploying remote client plug-ins.

Read the following topics next:

- Best Practices for Implementing Plug-in Workflows
- Best Practices for Localization
- Best Practices for Deploying and Testing Remote Plug-ins
- Additional Resources

Best Practices for Implementing Plug-in Workflows

The following practices are recommended to ensure that your plug-in is stable and performs well.

- If you need to temporarily store data, use the browser cache or your own back-end server.
- Do not send calls to the topmost browser window, `window.top`, or to the parent object of your current window, `window.parent`.
- To increase security of your extensions, limit the users who can access your plug-ins, and control user access to your extensions based on their privileges. For example, you can make your extensions available only to users who have privileges to create or delete Datastore objects. For more information, see Dynamic Extension Use Cases.

Best Practices for Localization

For a partner's remote client plug-in to pass certification for vSphere 8.0, it must support all locales that vCenter supports, as below. Language localizations should be done during the development cycle to prevent delay during the certification process.

- en_US
- de_DE
- es_ES
- fr_FR
- ja_JP

- ko_KR
- zh_CN
- zh_TW
- it_IT

Best Practices for Deploying and Testing Remote Plug-ins

Apply the following advice to minimize difficulties when you deploy your plug-in or plug-ins.

- To prevent deployment issues when you try to deploy a new version of a registered plug-in, make sure that you modify the version property of your plug-in extension.
- To prevent deployment issues when you try to deploy a plug-in with the same version, make sure that you unregister the plug-in by removing the plug-in as a vCenter Server extension. You must also manually delete the cached files of the plug-in that are stored in one of the following locations:

Environment	Location of Cached Packages
vCenter Server Appliance	`/etc/vmware/vsphere-ui/vc-packages/vsphere-client-serenity/`

- To avoid performance issues, make sure that your plug-in 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 the deployment of your plug-in and monitor for any issues related to your plug-in, see one or more of the following resources:
  - The **Administration > Client Plug-ins** view in the vSphere Client.
  - The `Download Plugin` task in the Tasks console.
  - The `Deploy Plugin` task in the Tasks console.
  - The Tomcat Server log files.

  You can find the Tomcat server log files in one of the following locations:

Environment	Tomcat Log Files Location
vCenter Server Appliance 8.0 installation vSphere Client	`/var/log/vmware/vsphere-ui/logs/`

  The `vsphere_client_virgo.log` file contains the log information that the Tomcat server generates. Problems usually start with the `[ERROR]` tag. Use your plug-in name or the bundle symbolic name to detect errors caused by your plug-in.

Additional Resources

The following additional resources can help you to design or upgrade your plug-ins for the vSphere Client.

UX Design Guidelines for vSphere Client Plug-ins	Provides details on vSphere Client plug-in extensibility and UX recommendations.
vSphere Client SDK Forum	A community forum for questions on plug-in development and SDK topics.
HOL-1911-07: Hands On Lab on building a vSphere Client Plug-in	Guides the plug-in developer through all the stages of developing a local plug-in.
Upgrading Plug-ins from vSphere 6.5 to vSphere 6.7	Explains Spring upgrade requirements for plug-ins and how to adapt plug-ins to support all release versions.
Virgo Application Server Replacement	Provides information about the replacement of the Virgo application server with a standard Tomcat server.
vSphere Client Remote Plug-in Extensions Reference	Provides details on the remote plug-in manifest JSON format.
vSphere Client Plug-in Manifest Conversion Tool	Migrates a local plug-in XML manifest to a remote plug-in JSON manifest.
JavaScript API Migration Guide	Migrates a plug-in with Bridge JavaScript APIs to a plug-in with new JavaScript APIs.

Troubleshooting Remote Plug-ins for the vSphere Client

12

These topics describe common problems that users see when deploying and operating remote plug-ins. The topics describe characteristic symptoms and recommend solutions.

Some problems might be hard to distinguish because of similar symptoms. For example, several different underlying problems can cause a situation where a plug-in user interface fails to display. If a troubleshooting topic matches your symptom but the troubleshooting advice does not lead to a matching cause or a working solution, check other topics for similar symptoms.

Read the following topics next:

- Plug-in Does Not Appear in vSphere Client
- Missing Entry in the Instance Selector
- Unable to Change Plug-in Manifest File
- Troubleshooting: Problems with Registration Script in SDK

Plug-in Does Not Appear in vSphere Client

The plug-in does not appear in the vSphere Client.

Problem

One or more of these symptoms is evident:

- A plug-in does not appear in the object navigator.
- A plug-in view does not display in the browser.

Cause

Several causes can prevent a plug-in from appearing. Try the following troubleshooting topics:

- Plug-in Manifest URL Unreachable
- Troubleshooting: Plug-in Thumbprint Incorrect
- Manifest Cannot Be Parsed
- Plug-in Already Registered
- Wrong Plug-in Type

- Plug-in Marked As Incompatible
- Plug-in Registered with Incompatible Version
- Plug-in View is missing in the vSphere Client

Wrong Plug-in URL

The remote plug-in does not display in the vSphere Client user interface.

Problem

The plug-in does not appear in the object navigator, but it is listed in the vSphere Client under **Admin > Client plugins**.

Cause

This problem can occur if the plug-in URL is not reachable. The plug-in registration command might have supplied an incorrect value for the `extension.url` property, which causes deployment to fail.

Note vCenter Server supports only the HTTP protocol for the plug-in URL. Secure HTTP (HTTPS) is recommended for production environments.

To verify the cause, use one of the following methods:

- In the Management UI, find the plug-in in the list under **Admin > Client plugins**. Next to the list entry, look for a signpost which contains an error message saying that the plug-in failed to download:

  Error downloading plug-in. Make sure that the URL is reachable and the registered thumbprint is correct.

- If you do not find the signpost, you can search for the sample name in the log file at `/var/log/vmware/vsphere-ui/logs/vsphere-client-virgo.log` to find an error message.

Solution

1. Connect a web browser to the Managed Object Browser (MOB) of the vCenter Server with which you attempted to register the plug-in.

   The MOB URL is **`https://vcenter_server_FQDN/mob`**.

2. To view the ServiceContent data object click the **content** link.
3. To view the ExtensionManager managed object click the **ExtensionManager** link.
4. In the **extensionList** values, search for the extension key that you used to register the plug-in.

   The **more...** link enables you to see more of the list of extension keys.

5 Check that the `client.url` property is correct and specifies HTTP or HTTPS.

   Enter the URL in a browser address bar and verify that the browser displays the contents of the `plugin.json` file.

   If the URL used in the registration command was correct, check the thumbprint by using the topic Troubleshooting: Plug-in Thumbprint Incorrect.

6 If the URL used in the registration command was incorrect, reregister the plug-in as follows:

   a Unregister the plug-in.

      You can use the registration tool in the SDK to unregistor a plug-in. For syntax information, see vSphere Client Plug-in Registration Tool.

   b Repeat the plug-in registration step with the correct URL.

Troubleshooting: Plug-in Thumbprint Incorrect

The remote plug-in does not display in the vSphere Client user interface.

Problem

The plug-in does not appear in the object navigator, but it is listed in the vSphere Client under **Admin > Client plugins**.

Cause

This problem can occur if the `extension.thumbprint` is not valid. The plug-in registration command might have supplied an incorrect server thumbprint, which causes deployment to fail.

To verify the cause, use one of the following methods:

- In the Management UI, find the plug-in in the list under **Admin > Client plugins**. Next to the list entry, look for a signpost which contains an error message saying that the plug-in failed to download:

  Error downloading plug-in. Make sure that the URL is reachable and the registered thumbprint is correct.

- If you do not find the signpost, you can search for the sample name in the log file at `/var/log/vmware/vsphere-ui/logs/vsphere-client-virgo.log` to find an error message.

Solution

1 Connect a web browser to the Managed Object Browser (MOB) of the vCenter Server with which you attempted to register the plug-in.

   The MOB URL is **`https://`*`vcenter_server_FQDN`*`/mob`**.

2 To view the ServiceContent data object click the **content** link.

3 To view the ExtensionManager managed object click the **ExtensionManager** link.

4 In the **extensionList** values, search for the extension key that you used to register the plug-in.

   The **more...** link enables you to see more of the list of extension keys.

5 Check that the `server.serverThumbprint` property is correct.

   - The thumbprint (fingerprint) should match the characters shown in the plug-in server certificate. To find the plug-in server thumbprint, see Get Thumbprint or Certificate of Remote Plug-in Server. Character pairs should be colon separated.
   - Check for hidden characters.
   - Pairs of digits must be separated by colon separators.
   - You can register the plug-in using a SHA-256 fingerprint or a SHA-1 fingerprint. A best practice is to use SHA-256, which is more secure. SHA-1 is deprecated in favor of SHA-256.

   If the thumbprint is correct, check the URL by using the topic Wrong Plug-in URL.

6 If the thumbprint used in the registration command was incorrect, reregister the plug-in as follows:

   a Unregister the plug-in.

      You can use the registration tool in the SDK to unregistor a plug-in. For syntax information, see vSphere Client Plug-in Registration Tool.

   b Repeat the plug-in registration step with the correct thumbprint.

Manifest Cannot Be Parsed

The remote plug-in does not display in the vSphere Client user interface.

Problem

The plug-in does not appear in the object navigator, but it is listed in the vSphere Client under **Admin > Client plugins**.

Cause

The plug-in manifest file is not valid and the plug-in deployment fails due to unsuccessful schema validation.

You can distinguish this problem by searching for this error message in the `vsphere-client-virgo.log`:

Ignoring plugin extension.key because its JSON manifest could not be parsed.

Solution

1 According to the exception, locate the failure in the plug-in manifest file.

2 Trigger another plug-in discovery/deployment cycle. If the redeployment functionality is not enabled, restart the vSphere Client service, vsphere-ui, to get the new plug-in manifest file changes.

3 Verify that the plug-in has been deployed, if not look for other errors in the plug-in manifest file.

Wrong Plug-in Type

The remote plug-in does not display in the vSphere Client user interface.

Problem

The plug-in does not appear in the object navigator, but it is listed in the vSphere Client under **Admin > Client plugins**.

Cause

The plug-in was not registered correctly as a remote plug-in, which causes the vSphere Client to assume that it is a local plug-in. When the vSphere Client tries to download the plug-in manifest, it expects a `.zip` file but finds a `.json` file instead.

You can verify the cause of the problem by searching the `vsphere-client-virgo.log` file for an error message saying that the vSphere Client did not find a ZIP file at the plug-in manifest location:

Couldn't open plugin zip file when trying to verify the signature of plugin extension.key:extension.version java.util.zip.ZipException: error in opening zip file.

Solution

1 On the machine that runs the plug-in server, change to the directory that contains the plug-in manifest.

For example: `cd samples/remote-plugin-sample/target/classes/static`

2 Unregister the plug-in.

For example: `$tools/extension-registration.sh -action unregisterPlugin -k sample.plugin -url https://vc-svr.example.com/sdk -u administrator@vsphere.local -p secret`

3 Reregister the plug-in, specifying that it is a remote plug-in.

For example, you can use the plug-in registration tool in the SDK: `$tools/extension-registration.sh -action registerPlugin -remote -k sample.plugin -v 1.0.0 -url https://vc-svr.example.com/sdk -u administrator@vsphere.local -p $secret -pluginUrl $pluginurl -serverThumbprint $thumbprint -c 'Example Inc.' -n 'Example Plug-in' -s 'This plug-in is registered with the remote keyword.'`

Plug-in Marked As Incompatible

The remote plug-in does not display in the vSphere Client user interface.

Problem

The plug-in does not appear in the object navigator, but it is listed in the vSphere Client under **Admin > Client plugins**.

Cause

The plug-in is marked as incompatible in the compatibility matrix and is filtered out.

Note Plug-ins can be marked as incompatible with wild characters. The plug-in itself might not be marked as incompatible, but it matches a pattern in the compatibility matrix.

In the Management UI there is a signpost for the plug-in which contains an error message saying that the plug-in is marked as incompatible and can not be deployed:

The plug-in does not claim compatibility with the current vSphere Client version. Check plug-in's interoperability matrix.

Solution

1. Open the `/etc/vmware/vsphere-ui/compatibility-matrix.xml`.
2. Mark the plug-in with `extension.key` as compatible by adding: *<PluginPackage id="extension.key" status="compatible"/>*.

Plug-in Registered with Incompatible Version

The remote plug-in does not display in the vSphere Client user interface.

Problem

The plug-in does not appear in the object navigator, but it is listed in the vSphere Client under **Admin > Client plugins**.

Cause

The plug-in did not deploy because the `extension.version` is not well formatted.

The `vsphere-client-virgo.log` file contains an error message saying that the plug-in version is not well formatted:

DEPLOYMENT_FAILED: Error deploying plugin package extension.key:extension.version. Reason: Deployment error. java.lang.NumberFormatException: For input string: "some string".

Solution

- Verify that the `extension.version` is in the format "*A.B.C.D*", where:
  - *A*, *B*, *C*, and *D* are numbers.
  - *A* is mandatory.
  - *B*, *C*, and *D* are optional.

The following examples form valid version strings:

- "1"
- "2.3"
- "5.4.2"
- "12.3.22.4"

Plug-in View is missing in the vSphere Client

A specific view is missing in the vSphere Client.

Problem

Navigating to some extension point shows `Page Not Found`.

Cause

The view was not loaded by the vSphere Client.

Solution

Verify that the view URL is correctly defined in the plug-in manifest file.

Verify directly that the URL is reachable from the plug-in server, bypassing the vCenter Server reverse proxy.

Missing Entry in the Instance Selector

A plug-in instance is missing from an Instance Selector.

Cause

This problem can occur only in an ELM environment, where multiple vCenter Server instances are linked. The plug-in instance entry is missing because the plug-in from the vCenter Server instance did not download or deploy the plug-in correctly.

Solution

- Verify that the plug-in is registered with the vCenter Server instance.
- Use other troubleshooting topics in Plug-in Does Not Appear in vSphere Client to find out why the plug-in did not download or deploy.

Unable to Change Plug-in Manifest File

The vSphere Client does not respond to changes in a remote plug-in manifest file.

Problem

After a plug-in fails to deploy due to an invalid plug-in manifest file, the vSphere Client marks it as broken and does not attempt to retry the download even if the plug-in manifest file is later updated on the plug-in server.

Cause

After download the plug-ins are cached locally and a new download will happen if the cached manifest file is deleted. This is expected production workflow. However, development environments require additional configuration to avoid the cached plug-in manifest files.

Solution

Use the Redeploy feature to bypass the plug-in cache during development. For more information, see Redeploying Plug-ins During Development.

Troubleshooting: Problems with Registration Script in SDK

The vSphere Client SDK provides a script that you can use to register plug-ins during development. Errors in the registration command can cause several kinds of plug-in failures. Most plug-in registration failures go undetected until run time, but some failures cause the registration script to report an error.

Plug-in Already Registered

The remote plug-in registration script fails.

Problem

The plug-in registration script reports an error. If you overlook the error, the plug-in does not appear in the object navigator and the plug-in is not listed in the vSphere Client under **Admin > Client plugins**.

Cause

The extension key was previously registered with the Extension Manager. The Extension Manager rejects a new registration with a duplicate value in the extension key property.

The `extension-registration.sh` or `extension-registration.bat` script reports a message similar to the following:

```
Client received SOAP Fault from server: A specified parameter was not correct:
extension.key
```

Solution

1. Connect a web browser to the Managed Object Browser (MOB) of the vCenter Server with which you attempted to register the plug-in.

   The MOB URL is **`https://vcenter_server_FQDN/mob`**.

2 To view the ServiceContent data object click the **content** link.

3 To view the ExtensionManager managed object click the **ExtensionManager** link.

4 In the **extensionList** values, search for the extension key that you used in your registration command.

   The **more...** link enables you to see more of the list of extension keys.

5 If you find the extension key in use, use a different key or unregister the extension key before you retry the extension registration script.

Unable To Unregister Plug-in

A plug-in unregistration command fails.

Problem

You use the plug-in registration tool to unregister a plug-in, but the tool reports a failure to unregister the plug-in.

Cause

The **`--key`** argument did not match the `extension.key` property of the registration record in the ExtensionManager.

Solution

1 Use the **`--action unregisterPlugin`** argument with a key that matches the key you used to register the plug-in. If you no longer have access to the registration command you used, you can find the extension key value from the registration record in the ExtensionManager by using the following procedure.

   a Connect a web browser to the Managed Object Browser (MOB) of the vCenter Server with which you attempted to register the plug-in.

      The MOB URL is **`https://`*`vcenter_server_FQDN`*`/mob`**.

   b To view the ServiceContent data object click the **content** link.

   c Click the **ExtensionManager** link to view the ExtensionManager managed object.

   d In the **extensionList** values, search for the extension key that you used in your registration command.

      The **more...** link enables you to see more of the list of extension keys.

   e Locate the `extension.key` property of the registration record for your plug-in.

2 Retry the unregistration command, supplying a corrected **`--key`** argument.