Developing Remote Plug-ins with the vSphere Client SDK

vmware®

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

https://docs.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 vSphere Client user interface.

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

Intended Audience

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

VMware Technical Publications Glossary

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

Revision History

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

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

Revision	Description
05 DEC 2022	Minor cosmetic cleanup.
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.
02 APR 2020	vSphere Client SDK 7.0 release. ■ Extended and updated JavaScript API. ■ SHA-256 thumbprint support. ■ Support for auiliary 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.

Revision	Description
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.

About the vSphere Client SDK

1

The VMware vSphere® Client provides a means for connecting to VMware vCenter Server® systems and managing the objects in the vSphere infrastructure. The VMware vSphere Client is an HTML5-based web application with a modular architecture that 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.

Using the vSphere Client Remote Plug-in Sample

2

For an introduction to the vSphere Client Remote Plug-in SDK, you can install and run the remote plug-in sample. The remote plug-in sample demonstrates a secure, efficient, remote plug-in that was developed according to recommended practices. The sample illustrates 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.

This chapter includes the following topics:

- About the Remote Plug-in Sample
- Components in the Remote Plug-in Sample
- Build the vSphere Client Remote Plug-in Sample
- Find the SSL Thumbprint and GUID of vCenter Server
- Find the SSL Thumbprint of the Remote Plug-in Server
- Register the vSphere Client Remote Plug-in Sample
- Viewing the vSphere Client Remote Plug-in Sample
- Remote Plug-in Sample Directory Structure

About the Remote Plug-in Sample

The Remote Plug-in sample supports a Global View extension to the vSphere Client. The user interface component creates a modal dialog and several portlets that extend a vSphere inventory object 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 a context object view.

- 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 2-1. Remote Plug-in Sample Environment
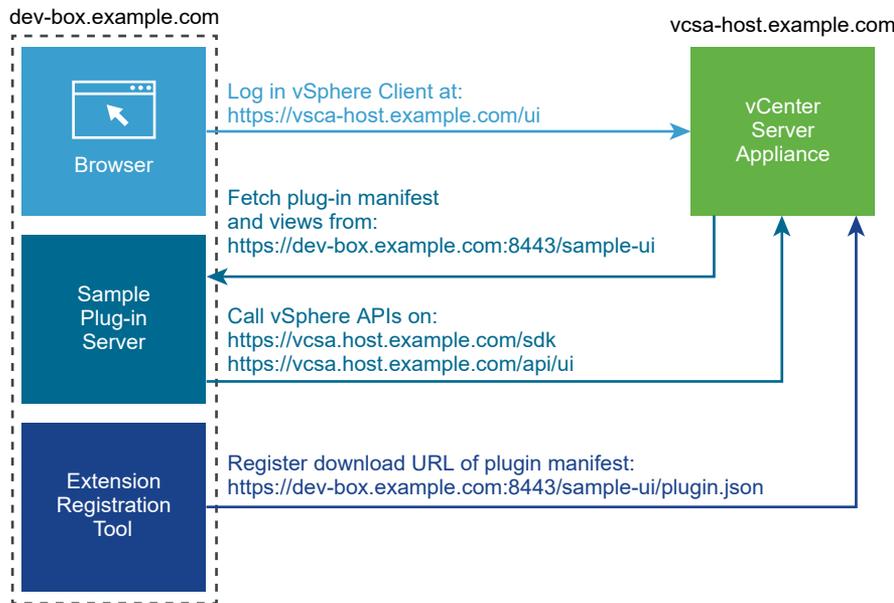
Build the vSphere Client Remote Plug-in Sample

You deploy the remote plug-in sample with a Maven build directed by the sample's `pom.xml` file. This procedure builds the sample and deploys it on the plug-in server.

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

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.

- You need to download and unzip the SDK. See Chapter 1 About the vSphere Client SDK for information about the structure of the SDK archive.

Procedure

1 Change to the root directory of the remote plug-in sample.

   For example: `cd /HD/sdk/html-client-sdk/samples/remote-plugin-sample`

2 `mvn validate`

3 `mvn clean install`

For a production build, substitute the command `mvn clean install`
`-Dproduction.build=true`

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 run 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 thumbprint and the GUID of the vCenter Server where you want to register the plug-in.

You need to find out the certificate thumbprint and the GUID of a vCenter Server instance.

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 either the SHA-256 or the SHA-1 fingerprint.

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

- The SHA-1 fingerprint is a string of 40 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 either 64 or 40 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. You can start the server to determine its certificate thumbprint.

Find the SSL Thumbprint of the Remote Plug-in Server

The remote plug-in sample has an embedded application server with a self-signed certificate that is used in 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's identity certificate. You use this thumbprint in the arguments to the registration command.

Prerequisites

- Before you can find the thumbprint of the application server, you must install the SDK and build the sample code.

- Before you start the application server to find its thumbprint, you need access to its port in the firewall. The default port number is 8443. You can configure a different port number in the `application.properties` file.

Procedure

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

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

2   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`.

3   Examine the certificate presented by the application server.

    The way to examine the certificate depends on the browser. For example, you can view a
    server certificate in Firefox by clicking the padlock icon next to the URL, then selecting **More
    Information > View Certificate**. The thumbprint is the field labelled `SHA-256 Fingerprint`.

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

4   Save the certificate thumbprint to a text file.

    If the thumbprint contains colon separators, do not remove them. If the thumbprint contains
    spaces or other separators, replace them with colons. If the thumbprint has no separators,
    insert a colon after every two digits. This is the format accepted by vCenter Server when you
    register the plug-in server.

What to do next

Use the application server thumbprint 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 and you are ready to run the remote plug-in sample.

Prerequisites

Before you register the remote plug-in sample, you must do the following:

- Build the remote plug-in sample.

- If needed, start vCenter Server.

- If needed, change permissions on the plug-in registration tool to allow execute access.

In addition, you need to know the following parameters:

- 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 that 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 of the plug-in server, so that vCenter Server can retrieve the plug-in manifest file. A SHA-1 thumbprint is also supported, but SHA-1 is deprecated in favor of SHA-256.

  Note The thumbprint must contain colon separators, unlike the thumbprint format used when starting the plug-in server. vCenter Server accepts only thumbprints that have a colon between each pair of characters.

- The version number of the remote plug-in sample.

- The key of the remote plug-in sample, defined in the plug-in manifest. By default, this is `plugin.name`.

- The path from the plug-in web server root to the plug-in manifest file. By default, this is `/sample-ui/plugin.json`.

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

Procedure

1 In a command shell, change to the `tools/vCenter plugin registration/prebuilt` directory.

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

   The directory contains both an `extension-registration.bat` script for Windows DOS shells, and an `extension-registration.sh` script for Unix or Linux shells.

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

For a Unix or Linux shell, use this syntax:

```
./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 \
-c 'Example, Inc.'  -n 'Remote Plug-in'  -s 'This is a sample plug-in'
```

For a DOS command shell, use this syntax:

```
./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 ^
-c 'Example, Inc.'  -n 'Remote Plug-in'  -s 'This is a sample plug-in'
```

For a PowerShell prompt, use this syntax:

```
./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 `
-c 'Example, Inc.'  -n 'Remote Plug-in'  -s 'This is a sample plug-in'
```

Note If the password contains special characters, use the appropriate escaping sequences for your shell.

The registration script displays a message that the plug-in `has been successfully registered in vCenter.`

Results

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. If the server is not running yet, change to the root directory of the remote plug-in sample and 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
```

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 2-2. vCenter Server Welcome Screen
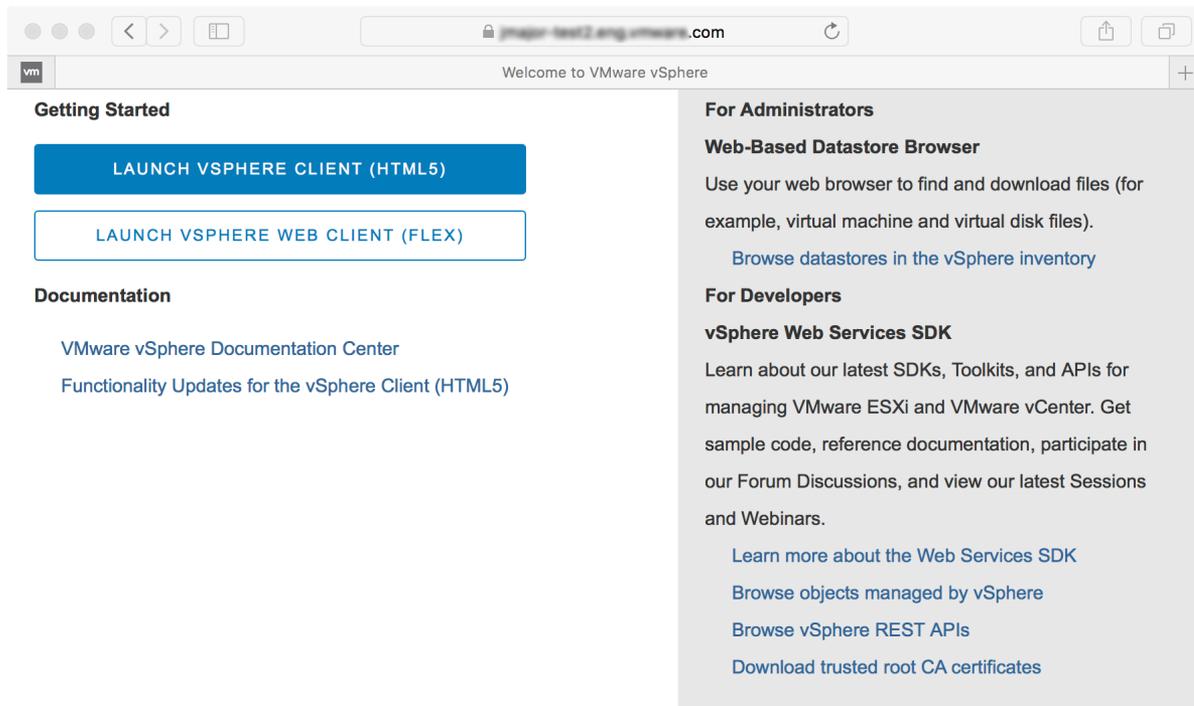
After you select **LAUNCH VSPHERE CLIENT (HTML5)** and authenticate with vCenter Server, the vSphere Client displays the home page with a **SDK Remote Plugin** link in the object navigator pane on the left. The link leads to the remote plug-in global view.

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 2-3. Selecting the Sample Plug-in Global View in the Navigation Pane
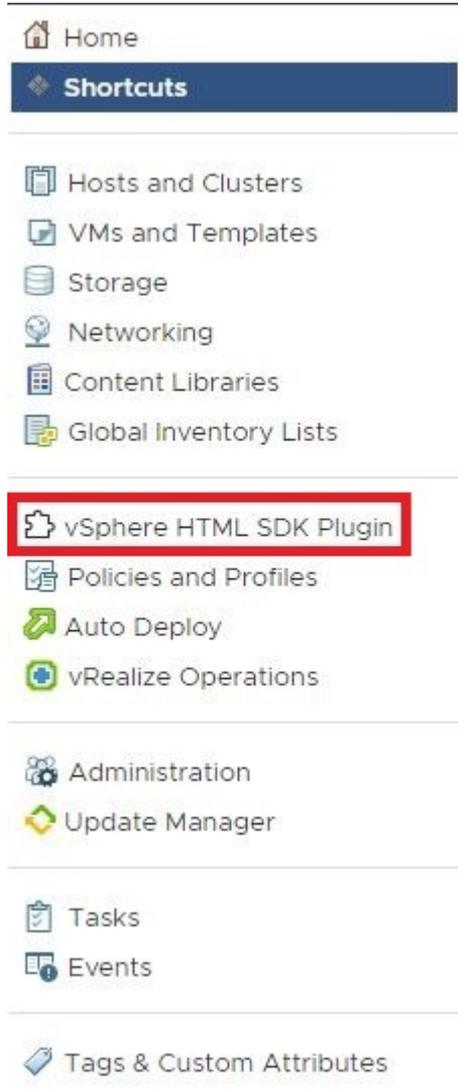
The Global View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"global": {
    "view": {
        "navigationId": "com.vmware.remote.sample.entrypoint",
        "uri": "index.html?view=entry-point",
        "navigationVisible": false
    }
}
```

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

Figure 2-4. Sample Plug-in Global View Welcome Page

Figure 2-5. Sample Plug-in Global View Chassis List Page
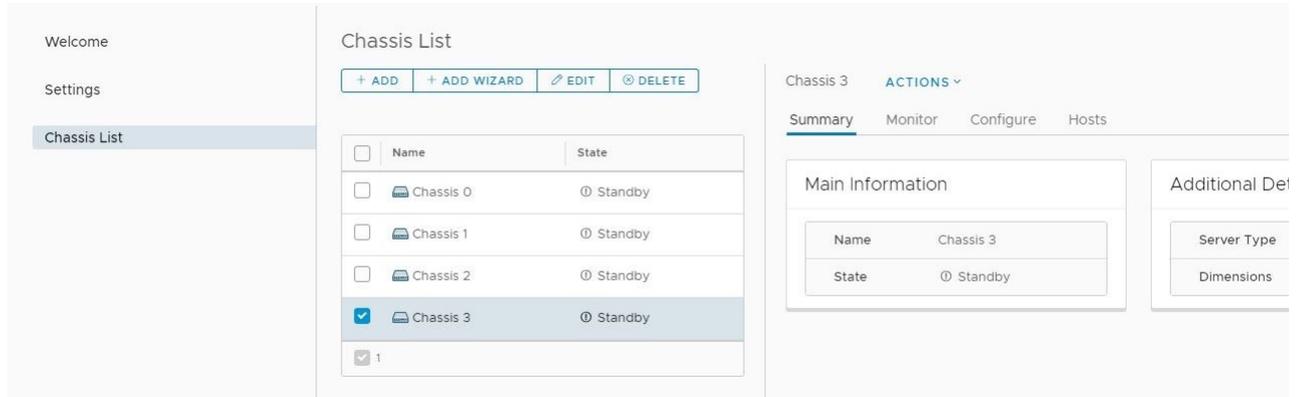
Figure 2-6. Sample Plug-in Edit Chassis Dialog
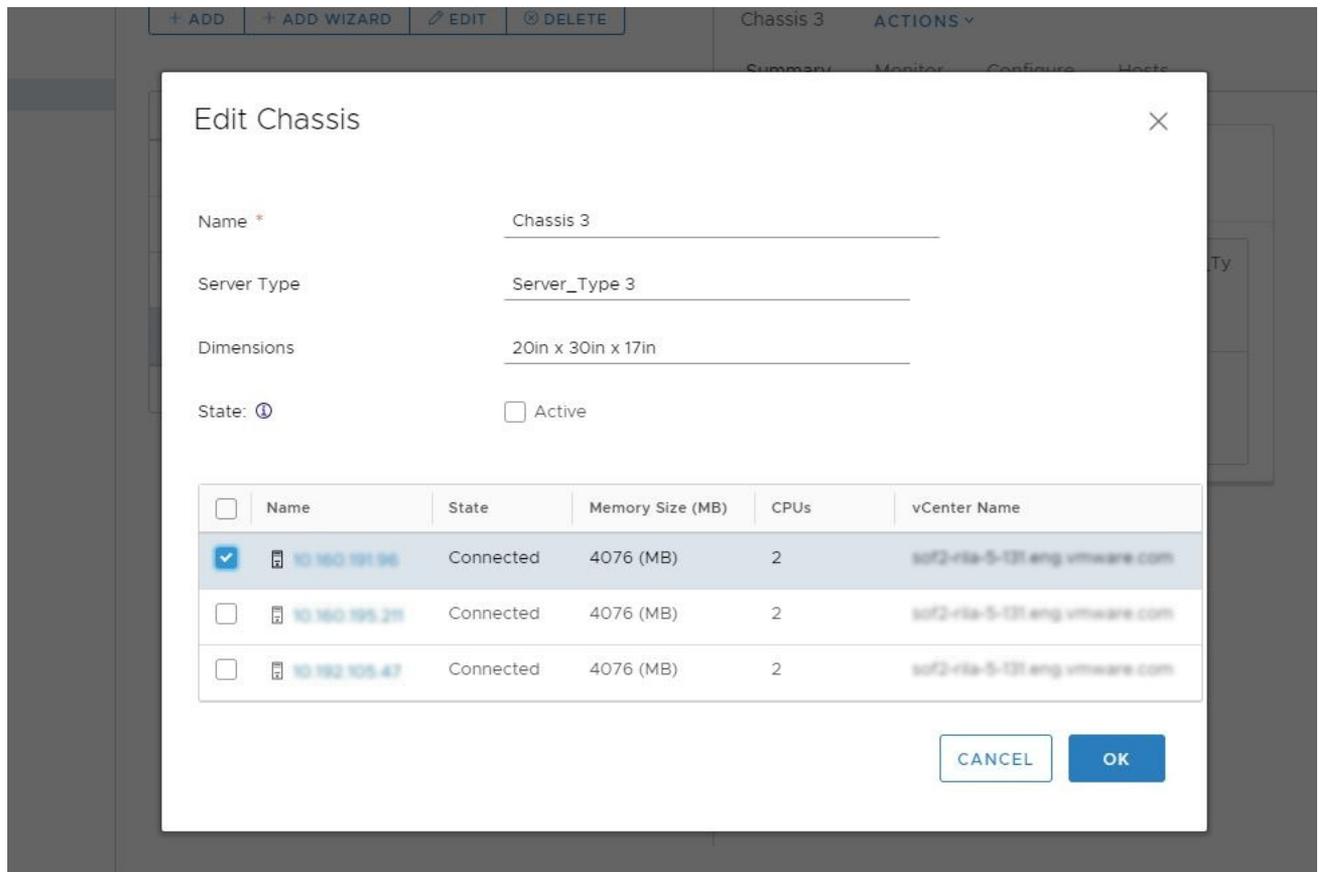
Figure 2-7. Sample Plug-in Chassis with Related Hosts
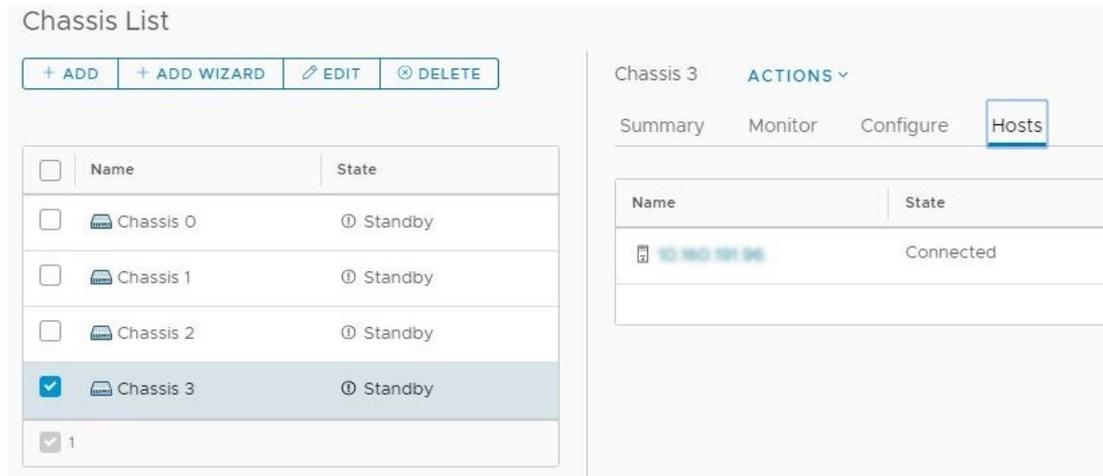
In addition to the global view and its subviews, the remote plug-in sample provides portlets that show how to extend the VirtualMachine vSphere object. You can see the portlets in context in the following illustrations.

Figure 2-8. Portlet Extending the VirtualMachine Summary View
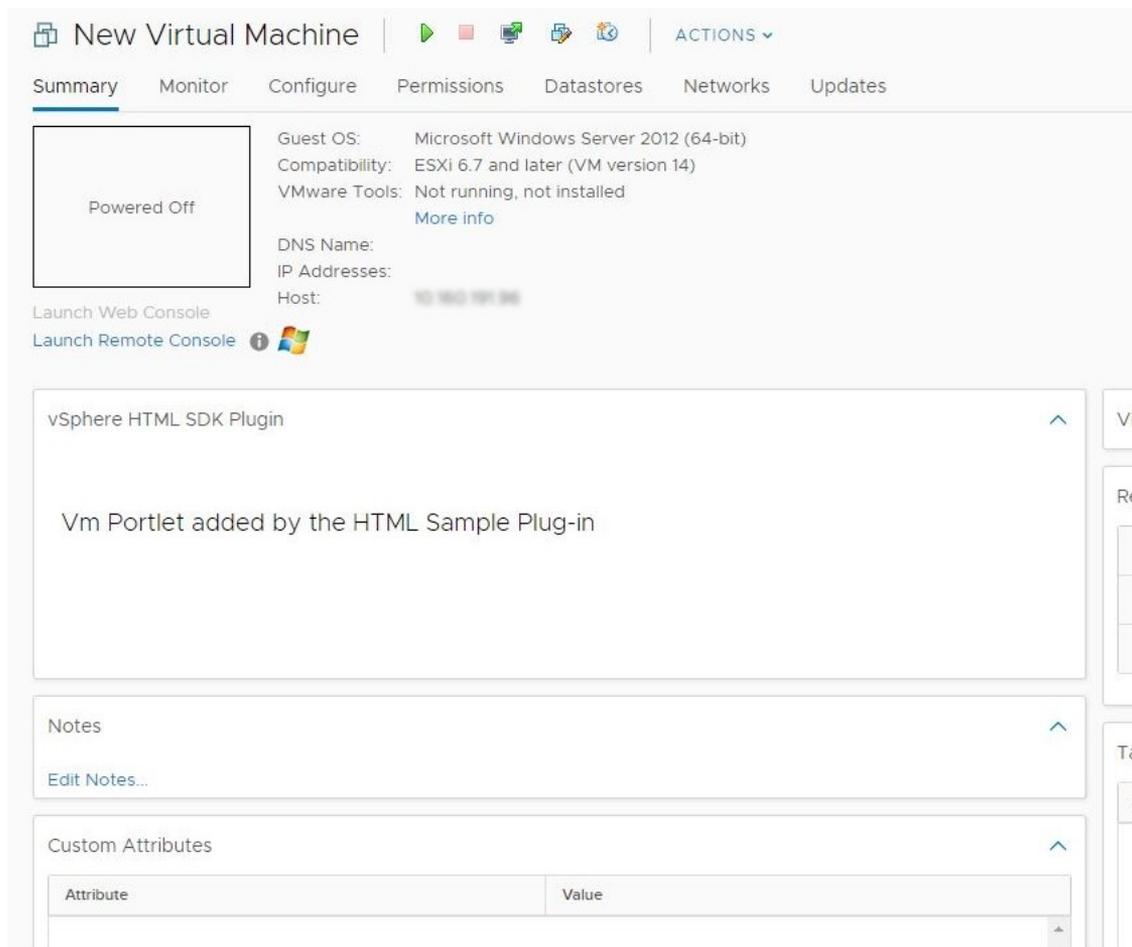
The VirtualMachine Summary View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
    "VirtualMachine": {
        "summary": {
            "view": {
                "uri": "index.html?view=vm-portlet",
            }
        },
        ...
    }
}
```

Figure 2-9. Portlet Extending the VirtualMachine Monitor View
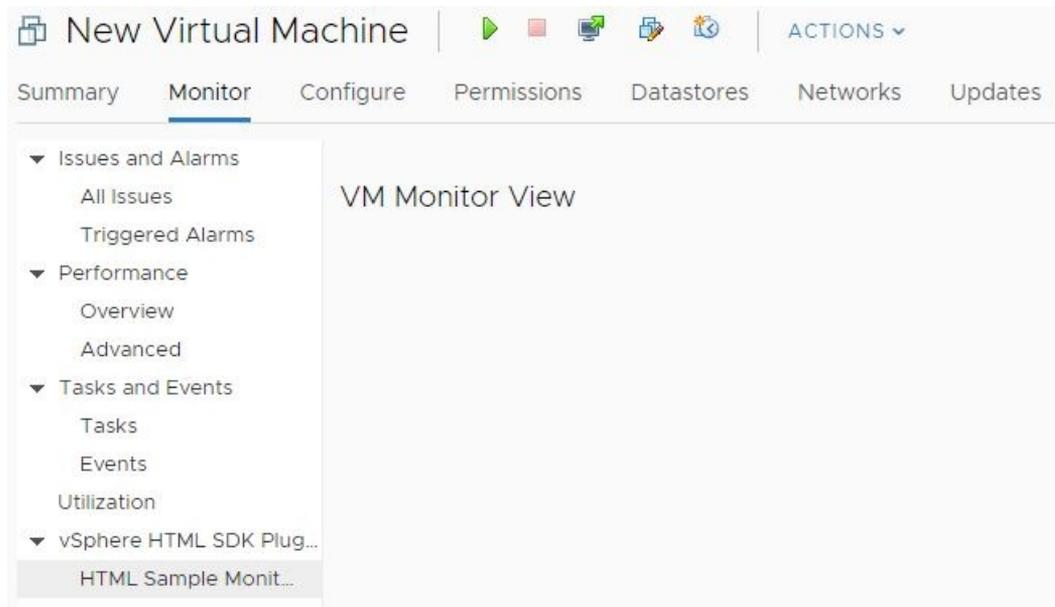
The VirtualMachine Monitor View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
    "VirtualMachine": {
        ...
        "monitor": {
            "views": {
                "navigationId": "com.vmware.remote.sample.monitor",
                "labelKey": "RemoteSample:vm.monitor.view.title",
                "uri": "index.html?view=vm-monitor",
            }
        }
    }
}
```

Figure 2-10. Portlet Extending the VirtualMachine Configure View
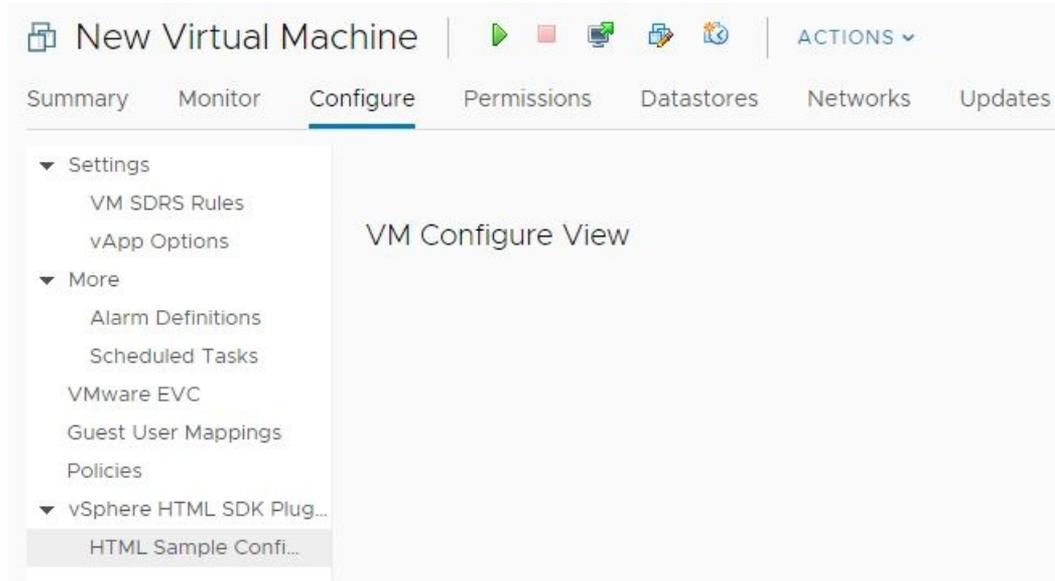
The VirtualMachine Configure View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
    "VirtualMachine": {
        "configure": {
            "views": {
                "navigationId": "com.vmware.remote.sample.configure",
                "labelKey": "RemoteSample:vm.configure.view.title",
                "uri": "index.html?view=vm-congfigure",
            }
        }
    }
}
```

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 2-11. Action Added to VirtualMachine Context Menu
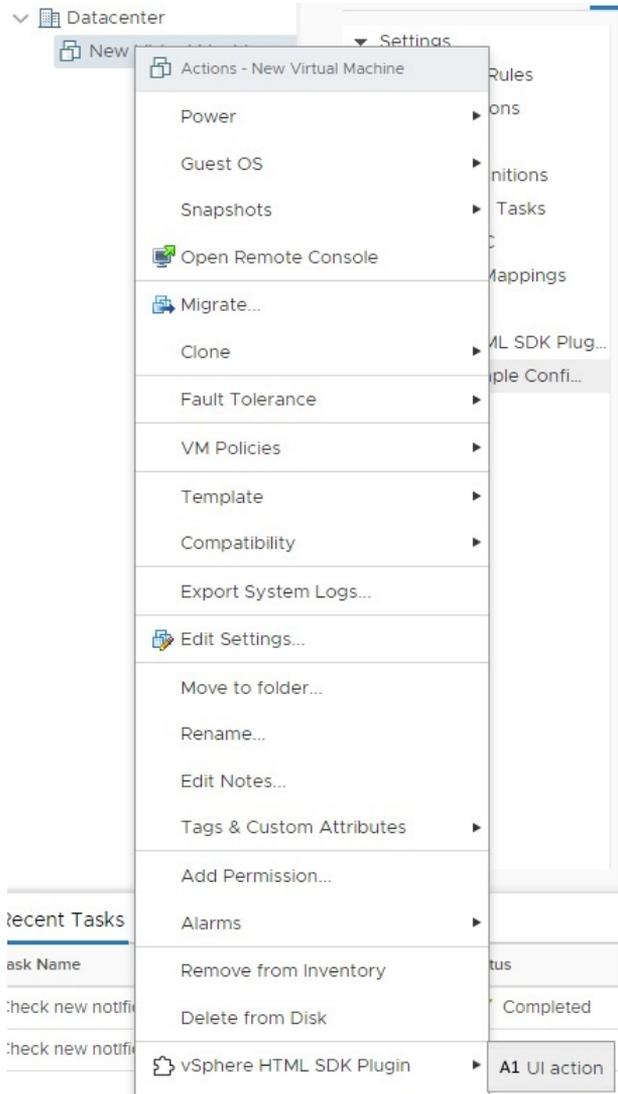
The VirtualMachine context menu extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
    "VirtualMachine": {
        "menu": {
            "actions": {
                "labelKey": "RemoteSample:vm.action.label",
                "icon": {
                    "name": "action-1"
                },
                "trigger": {
                    "type"; "modal",
                    "uri": "index.html?view=vm-action-modal",
                    "titleKey": "RemoteSample:vm.action.model.title",
                    "size": {
                        "width": 600,
```

```
            "height": 250
        }
      }
    }
  }
}
```

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 checkbox beside the Chassis object and click **Update**.

Figure 2-12. Portlet Extending the HostSystem Summary View
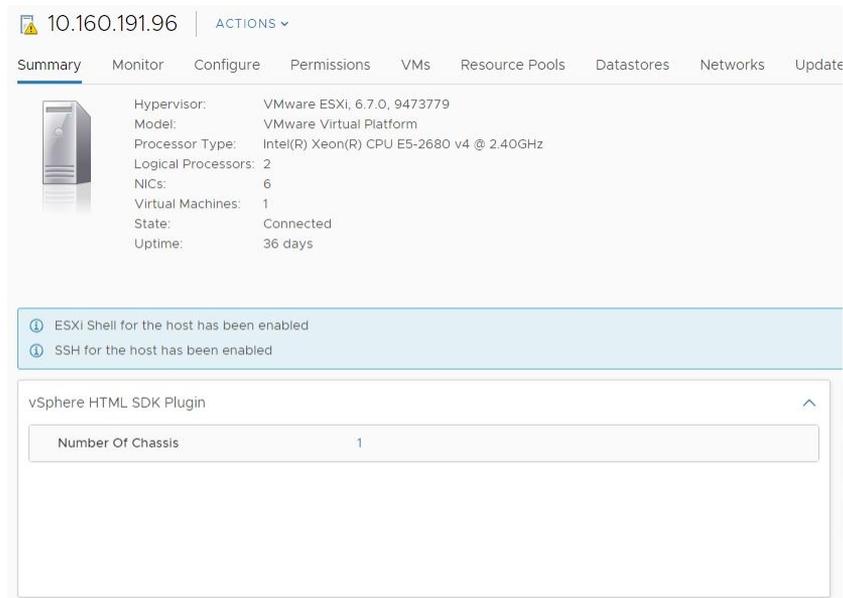
The HostSystem Summary View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
    "HostSystem": {
        "summary": {
            "view": {
                "uri": "index.html?view=host-summary"
            }
        }
    }
}
```

Figure 2-13. Portlet Extending the HostSystem Monitor View
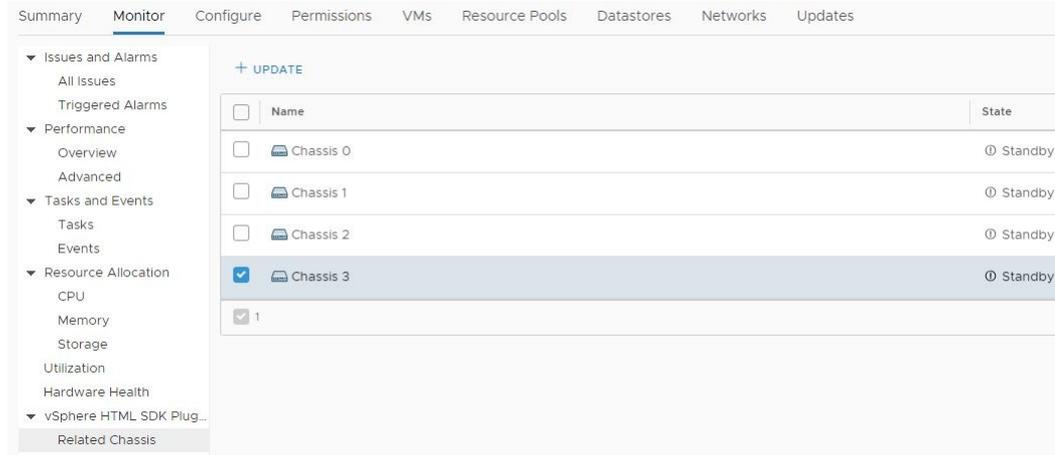
The HostSystem Monitor View extension point is specified by the following lines from the manifest file, `plugin.json`:

```
"objects": {
    "HostSystem": {
        "monitor": {
            "views": [
                {
                    "navigationId": "hostMonitor",
                    "labelKey": "host.monitor.view.title",
                    "uri": "index.html?view=host-monitor"
                }
            ]
        }
    }
}
```

Remote Plug-in Sample Directory Structure

The following illustrations show the directory structure of the remote plug-in sample code.

Top Level Directory

Figure 2-14. Remote Plug-in Sample Top Level Directory
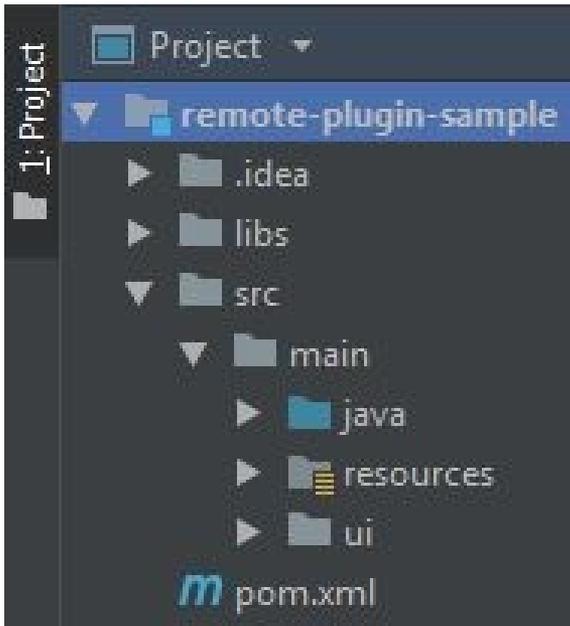
Table 2-1. Contents of Top Level Directory

Subdirectory	Description
remote-plugin-sample/libs	The external library (vim25.jar) that needs to be installed as a local dependency during the project build and deploy process.
remote-plugin-sample/src	The root of the plug-in source code. It contains three main subdirectories: ■ `java` ■ `resources` ■ `ui` Descriptions of these subdirectories follow.

`java` Directory for Server-Side Code

The server-side code for the remote plug-in sample is written in Java. It contains several packages.

Figure 2-15. Remote Plug-in Sample Server-Side Directory
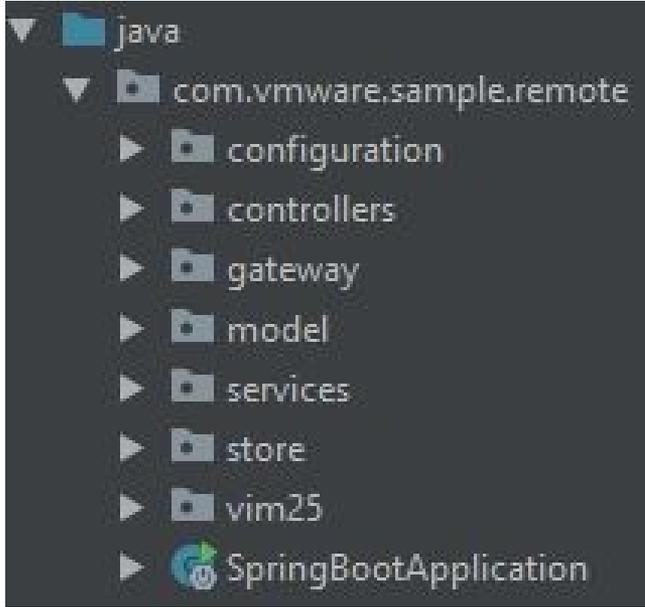
Table 2-2. Contents of Server-Side Directory

Subdirectory	Description
configuration	The Configuration service reads connection properties of vCenter Server, such as FQDN, thumbprint, port, and GUID, and exposes them to Spring Boot components through public accessors.
controllers	The Controller components have the logic to keep open user sessions to the vCenter Server.
gateway	The Session Service interacts with vCenter Server on behalf of the plug-in server. The Session service authenticates the plug-in server with a clone ticket.
model	Object definitions for plug-in server use.
services	Services that implement the plug-in server-side business logic.
store	In-memory database to store custom data for the plug-in.
vim25	Components that communicate with vCenter Server: ■ `VimObjectService` retrieves data from the vCenter Server, using the Web Services API. ■ `ThumbprintTrustManager` helps to authenticate the remote side of a secure socket, using the public `interface javax.net.ssl.X509TrustManager` interface.
SpringBootApplication.java	The Spring Boot Application bootstraps, deploys, and runs the plug-in on an embedded Tomcat application server.

`resources` Directory

The `resources` directory of the remote plug-in sample contains application configuration files and a certificate file for encrypted communications.

Figure 2-16. Remote Plug-in Sample Resources Directory
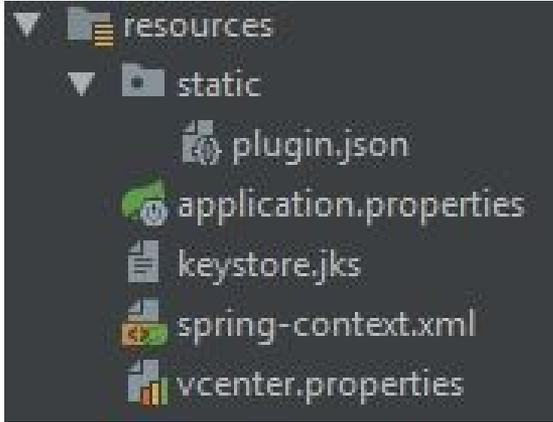
Table 2-3. Contents of Resources Directory

Subdirectory	Description
application.properties	Settings specific to the plug-in server, including port number for incoming HTTPS connections, context root of the web application, log level setting, and settings pertaining to the keystore.
keystore.jks	Java keystore for self-signed certificate of vCenter Server.
spring-context.xml	Context configuration for plug-in server Spring application.
static/plugin.json	Descriptors for UI components the plug-in adds to the vSphere Client.

`ui/src` Directory

The `ui/src` directory of the remote plug-in sample contains the client-side code for the plug-in. The main source files are in the `app` subdirectory, which contains an Angular application, including the AppModule, services, and component files.

Figure 2-17. Remote Plug-in Sample ui/src Directory

.
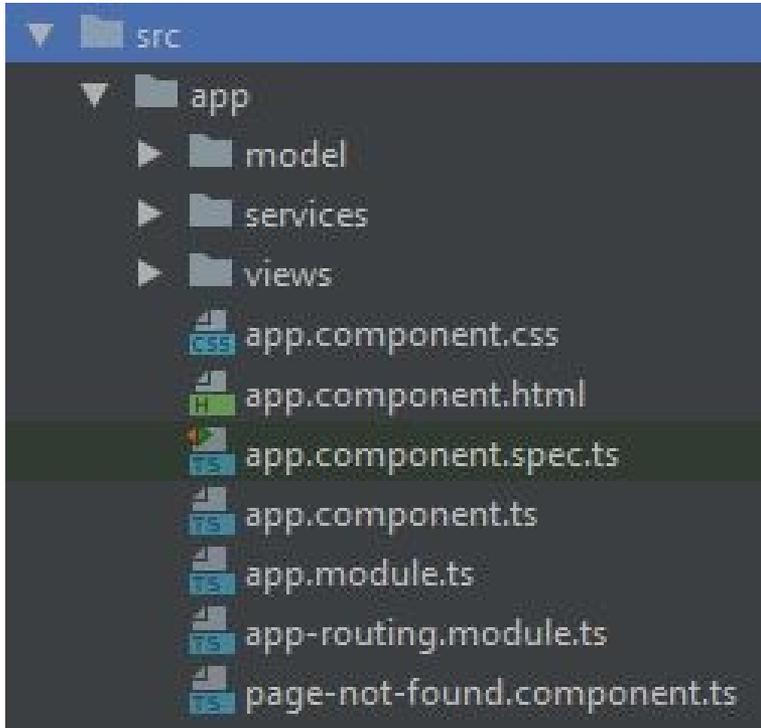
Table 2-4. Contents of ui/src Directory

Subdirectory	Description
app/model	Object definitions for plug-in client-side use.
app/services	Services that implement the plug-in client-side business logic
app/views	UI views (Angular components) that implement the visible functionality of the plug-in. Each view has its own subdirectory that contains HTML, CSS, and Angular code in separate files.
app/app.component	The main component that bootstraps the Angular Application and initializes the `htmlClientSdk` service.
app/app-routing.module	The routing module of the plug-in client-side application.
index.html	Static HTML page that loads by default when the plug-in server context path is loaded. The `index.html` page loads `htmlClientSkd.js` and `app.component`, which is the entry point of the Angular application.

Remote Plug-in Architecture in the vSphere Client

3

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 to plug-in developers:

- Your plug-in is protected from interference by unstable or compromised plug-ins loaded in the same vSphere Client.

- Plug-in compatibility is robust across vCenter Server upgrades.

- An incompatible plug-in does not interfere with vCenter Server operation.

- 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 understand. This aids in the process of troubleshooting a problem with your plug-in.

This chapter includes the following topics:

- Components of the vSphere Client Architecture

- vCenter Server Configurations

- Communication Paths in Remote Plug-in Architecture

- Communications Among UI Components in the vSphere Client

- Client-Server Communications with Remote Plug-ins

- Security Concepts for Remote Plug-ins

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 with an embedded Platform Services Controller to a number of vCenter Server instances in Enhanced Linked Mode or Hybrid Linked Mode.

The user interface component runs in a browser and manages the HTML5 views presented to the user. The user interface communicates with the vsphere-ui service, requesting HTML and Javascript files and vSphere inventory data, and 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 and the Platform Services Controller, 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 running in the Platform Services Controller.

vCenter Server Configurations

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

The following illustration shows a single vCenter Server with an embedded Platform Services Controller.

Figure 3-1. vCenter Server with Embedded PSC
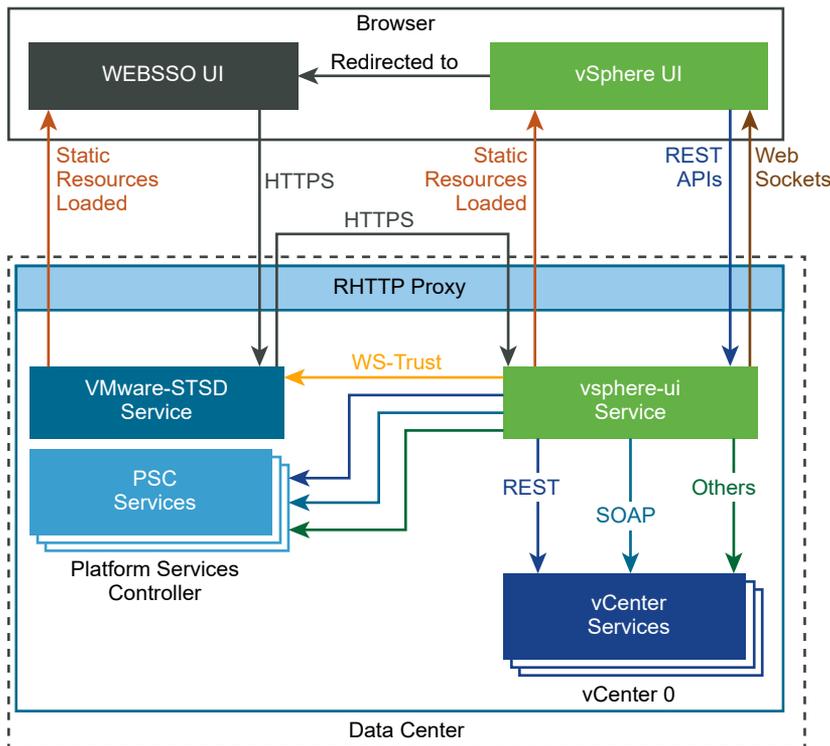
The following diagram shows three vCenter Server instances configured in Enhanced Linked Mode. The Platform Services Controller for this group is external to the vCenter Server instances.

Figure 3-2. vCenter Server Instances Configured in Extended Linked Mode
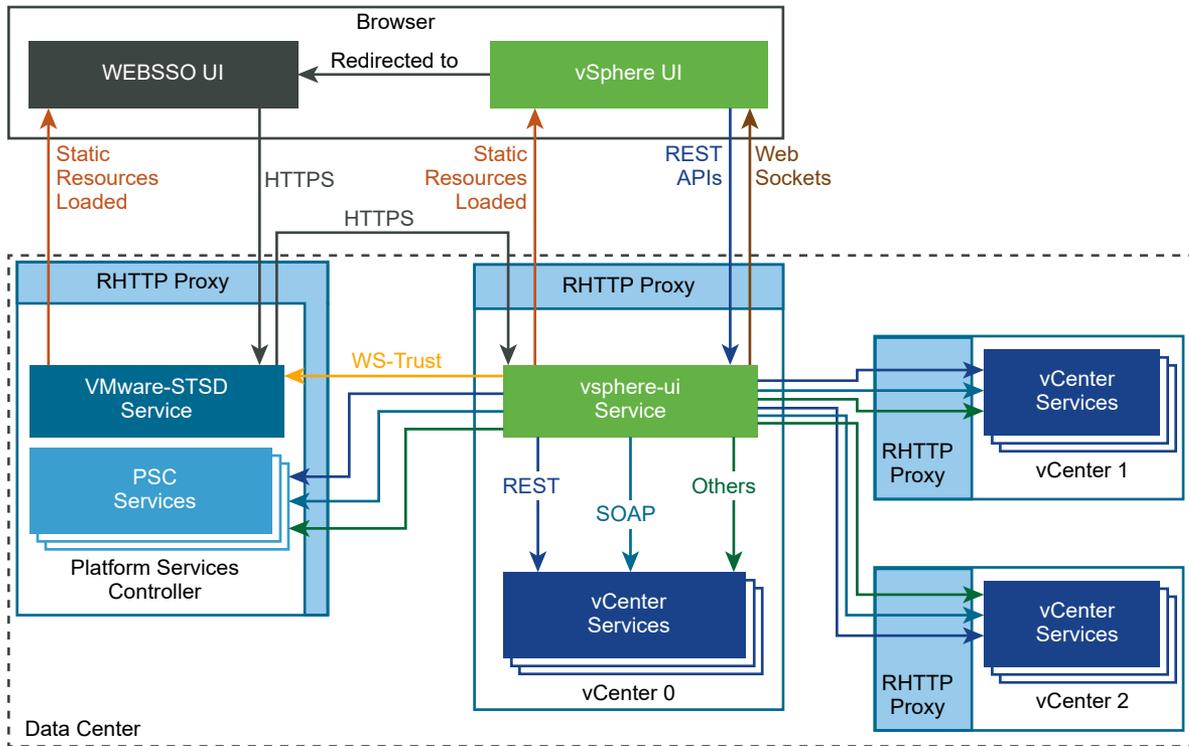
In a linked mode configuration, the vsphere-ui service handles delegation of requests to the linked vCenter Server instances, using the same protocols that it uses when communicating with its own services. From the point of view of the user interface, communications are as simple as in the embedded configuration.

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 3-3. A Simplified View of Remote Plug-in Architecture
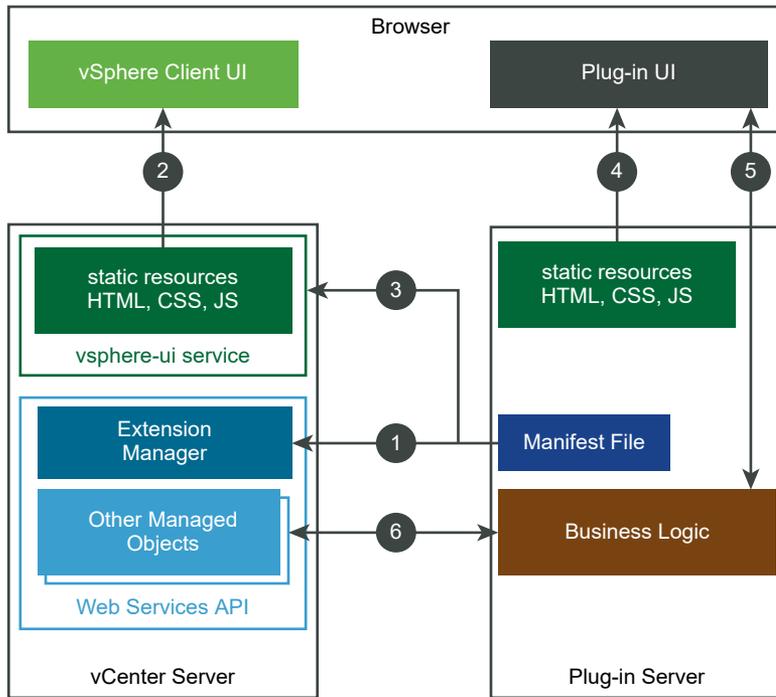
The circled numbers identify the following data paths:

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 3-4. User Interface with a Sandboxed Plug-in
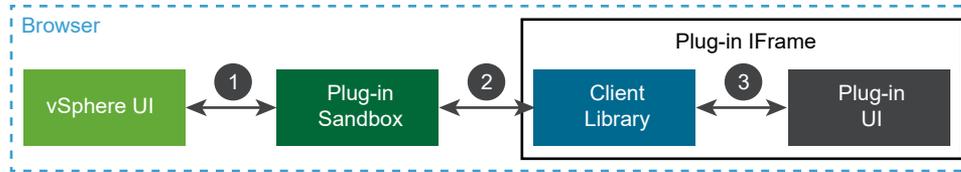
Paths:

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.

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 3-5. User Interface with Two Plug-ins in Separate Sandboxes
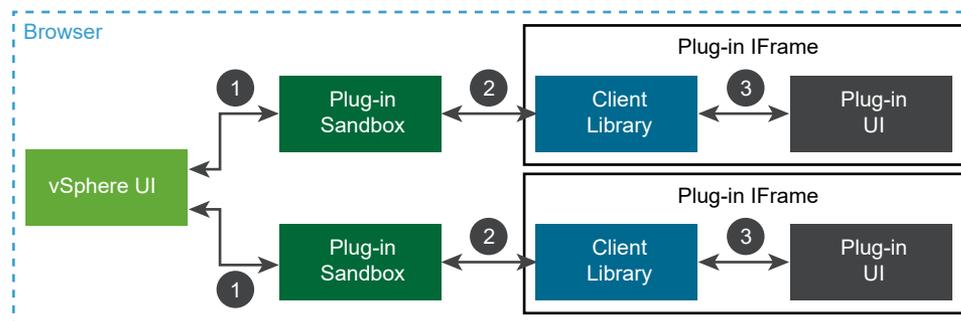
Paths:

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 Remote Plug-in Server Considerations for the vSphere Client.

Creating a Remote Plug-in for the vSphere Client

4

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.

This chapter includes the following topics:

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

Code Components To Create a Remote Plug-in for the vSphere Client

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.

Deployment Requirements for a Remote Plug-in for the vSphere Client

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 on the same machine as the web server.

- Prepare a plug-in manifest file, `plugin.json`, that specifies the plug-in components. The manifest file must be 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 certificate thumbprint of your plug-in manifest server and the URL of your plug-in manifest file.

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

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 \
-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 need to 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.

- The `server` array must have a `serverThumbprint` property in the first element, to support HTTPS connections. The `serverThumbprint` has the value of the SHA-256 hash of the server certificate.

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

For more information about the plug-in manifest, see Sample Manifest File for a Remote Plug-in. For more information about auxiliary plug-in processes, see Using Auxiliary Plug-in Servers. For more 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.

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

- 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

Each auxiliary server must be added to the `server` array of the plug-in registration record in the `ExtensionManager` object.

Registering Auxiliary Plug-in Servers

To register auxiliary servers with a vCenter Server instance, use the `server` property of the extension record. This is an array of `ServerInfo` objects, where each `ServerInfo` represents one of the plug-in's servers.

The first element of the array represents the manifest server. All other array elements represent auxiliary servers, in any order. The `type` field for each element contains a service identifier, and the `url` field contains the server endpoint address. The reverse HTTP proxy translates the server endpoints into proxy endpoints, which the JavaScript API makes available to front-end code.

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.

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, or updates the registration record of a plug-in.

Plug-in Registration Script

To register a plug-in, use the script in the SDK: `tools/vCenter plugin registration/prebuilt/ extension-registration.sh`, which is a wrapper for a Java tool. The registration tool opens a session with the vCenter Server instance using the VMware Web Services API.

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 URL must specify HTTP or HTTPS as the protocol. Optional properties that can also be present in the object are:

    - `type` is a service registration identifier string, by which the plug-in front end can discover a needed endpoint.

    - `ServerThumbprint` is the thumbprint of the auxiliary server's SSL certificate. See the constraints for the `-st` argument.

    - `label`

    - `Summary` is a brief description of the auiliary 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 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** (required to register a plug-in) is the SHA-256 signature of the plug-in back-end server certificate. Pairs of digits must be separated by colon separators.

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

- **-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.

Sample Manifest File for a Remote Plug-in

The following sample manifest file contains the minimum elements required for a very 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

5

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.

This chapter includes the following topics:

- Remote Plug-in Life Cycle
- Remote Plug-in Deployment
- Plug-In Caching
- Remote Plugin Uninstallation
- Redeploying Plug-ins During Development
- Specifying Remote Plug-in Compatibility
- Remote Plug-in Topologies

Remote Plug-in Life Cycle

Remote plug-in content becomes visible in the vSphere Client user interface 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 the 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 stages in its life cycle:

Build

The plug-in developer assembles the resources and builds the package to implement the 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 instance 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 5-1. Redeploy feature in Plugin Developer Mode
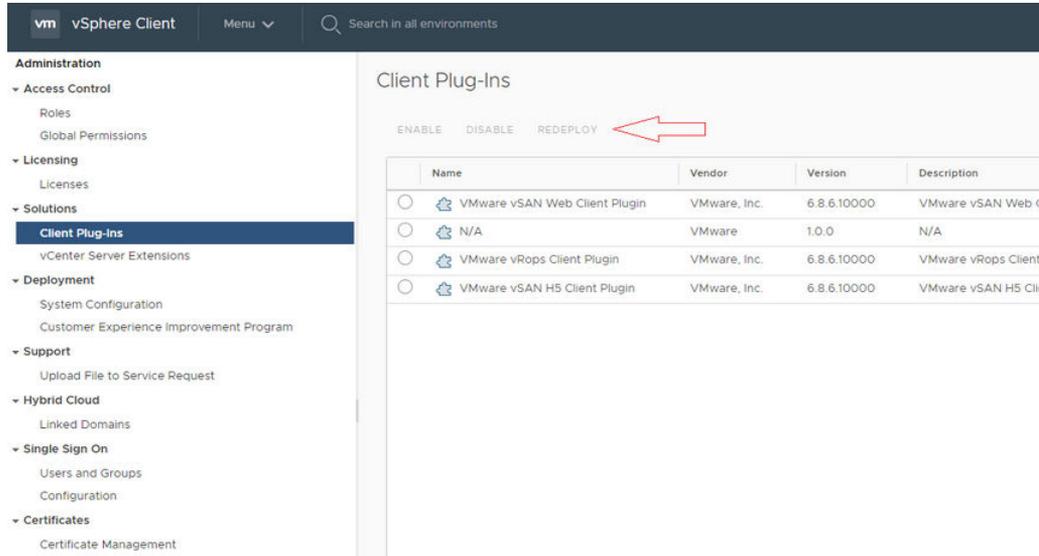
To enable the **Redeploy** button, set `pluginDeveloperMode` by appending the query parameter `pluginDeveloperMode=true` to any URL loaded into the browser window. For example, the URL could be the following:

```
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`.

Specifying Remote Plug-in Compatibility

The vSphere Client provides a way to specify compatibility between remote plug-ins and vCenter Server versions. You can also specify 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 path applies to views that access vSphere resources managed by the registration instance of vCenter Server.

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 suppose your plug-in is incompatible with vSphere Client versions before 7.0. In that case, you should specify a minimum version of 7.0 in the `vsphere.client` object in the manifest. If a user connects to a vCenter Server instance with an earlier version number, it will refuse 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 obects 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`

        An array of specifiers for vCenter Server environments. If the `environments` property is present in the `vcenter.server` object, the values specified limit the environments from which the plug-in can be deployed. If the `environments` property is absent, the plug-in can be deployed from any vCenter Server with which it was registered. Legal values are:

        - `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 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[.n]]]`

> **Note** 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`

    An array of specifiers for vSphere environments. If the `environments` property is present in the `vsphere.client` object, the values specified limit the environments that can deploy the plug-in to access resources on a vCenter Server instance where the plug-in is registered. If the `environments` property is absent, any environment can deploy the plug-in to access resources on a vCenter Server instance where the plug-in is registered. Legal values are:

    - `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[.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 the 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 the following examples, 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 instances that are completely equivalent in function, as long as they have the same version.

Figure 5-2. One Plug-in instance per vCenter Server
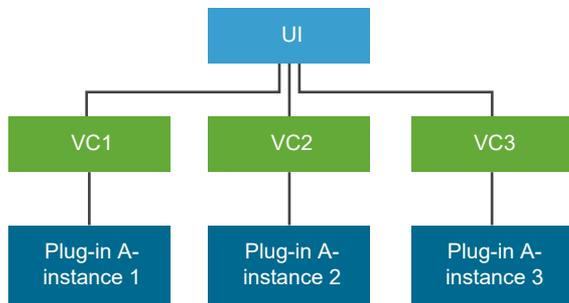
This topology has three instances of PluginA, such that the first plug-in server is connected to VC1, the second is connected to VC2, and the third server is connected to VC3. These plug-in servers are completely equivalent in function: the plug-in manifests that they host and the plug-in specific bits they run are identical.

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 portlet on the VM summary page, then browsing VMs on VC2 will load the portlet from the second plug-in server, and browsing VMs on VC3 will load the portlet 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 5-3. Selector for Single Plug-in Instance per vCenter Server
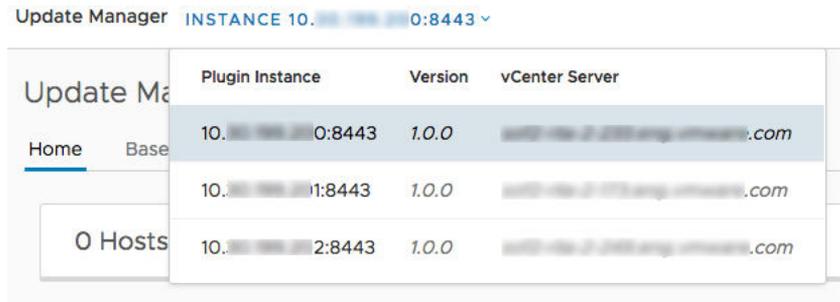
The instance selector shown above displays three back-end servers that represent the three instances of PluginA with version 1.0.0. The **Plugin Instance** column displays the IP address or fully qualified domain name of each plug-in instance, and the **vCenter Server** column displays the vCenter Server 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.

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.

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.

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 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, there will be a single entry point in the object navigator that takes the user to a plug-in instance selector view where the user will be able to switch between the global views of the two plugin 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:
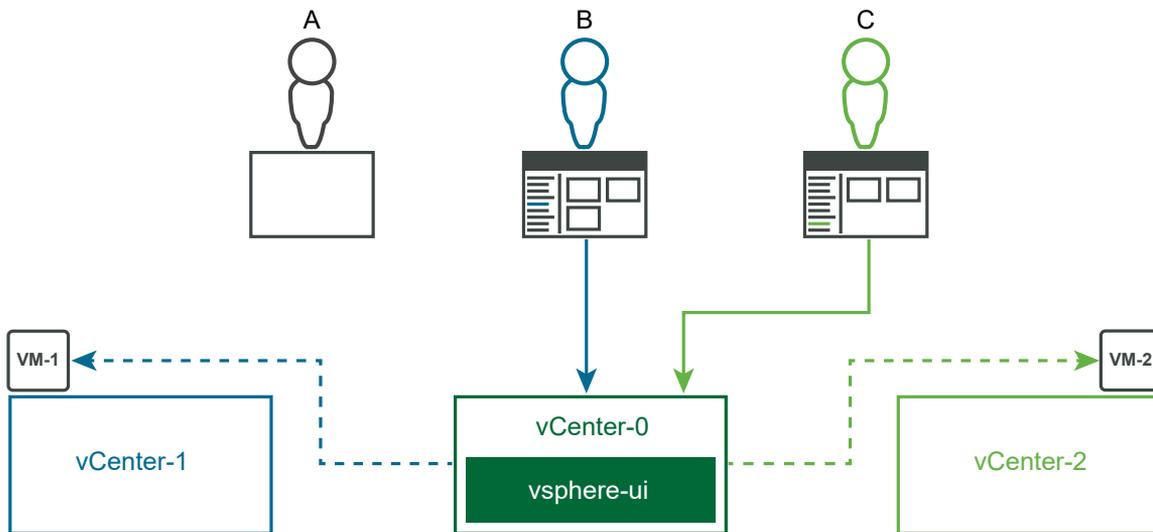
1   Alpha installs and configures the back-end server for the `ExampleCo` plug-in:
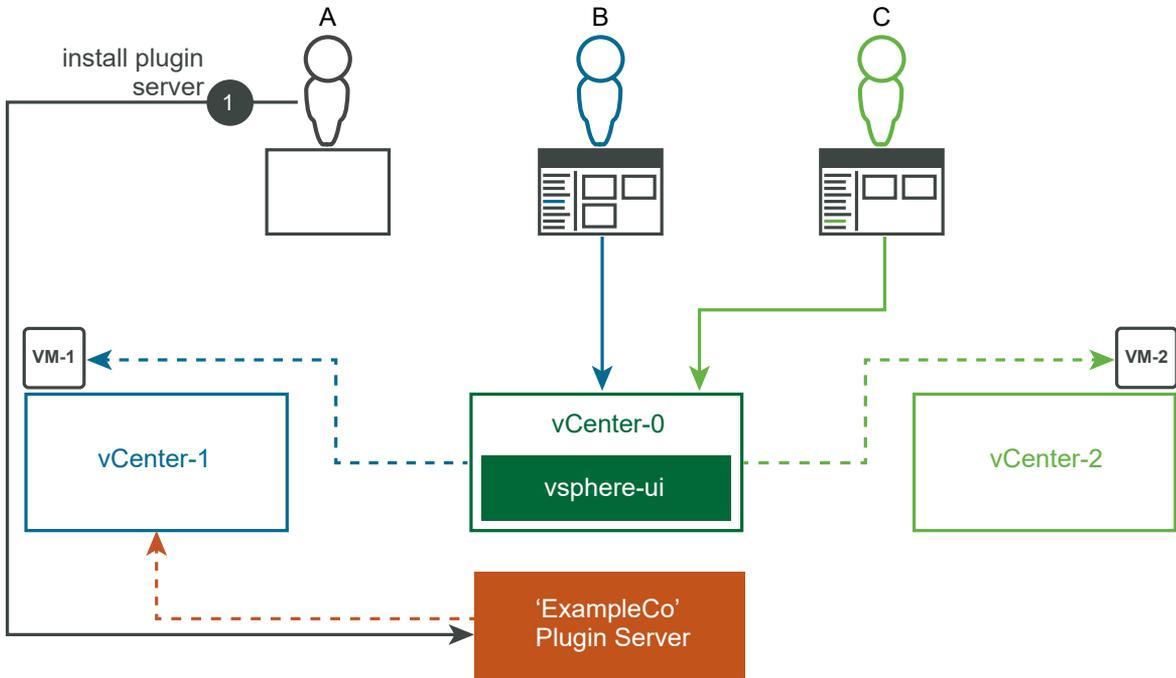
2   Alpha registers the `ExampleCo` plug-in with the `vCENTER-1` ExtensionManager by using Example Company's plug-in installer:
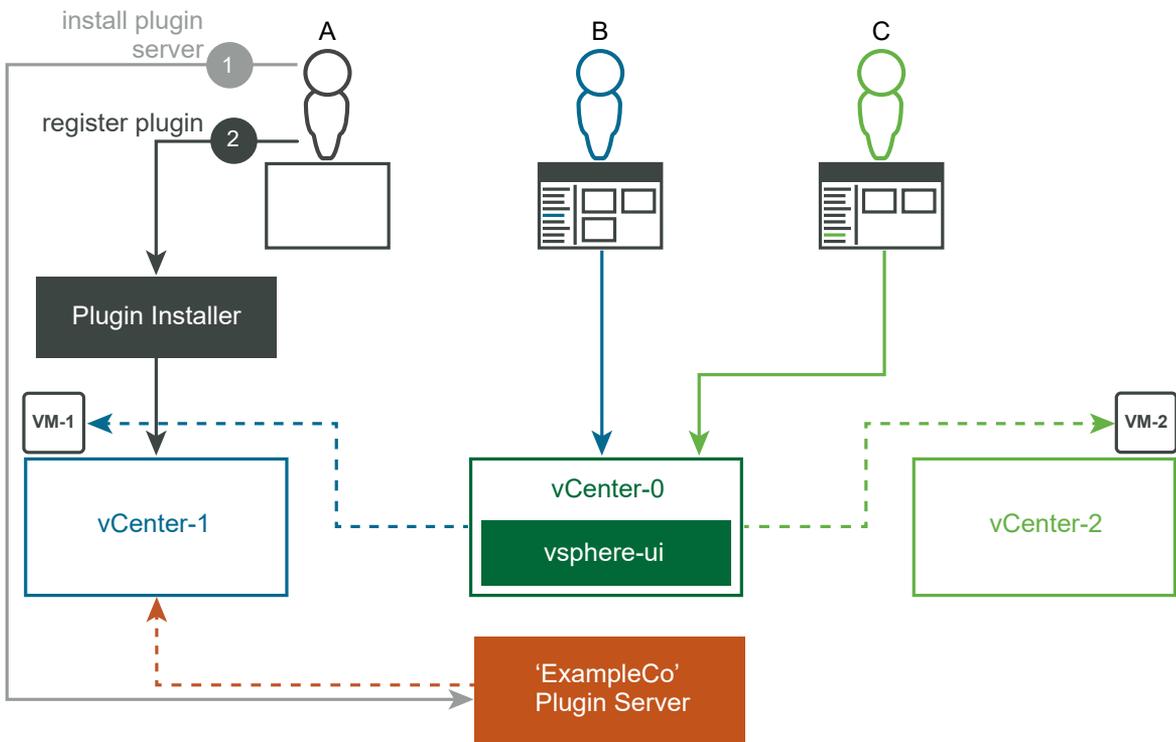
3   The plug-in registration triggers notifications to all linked vCenter Server instances:
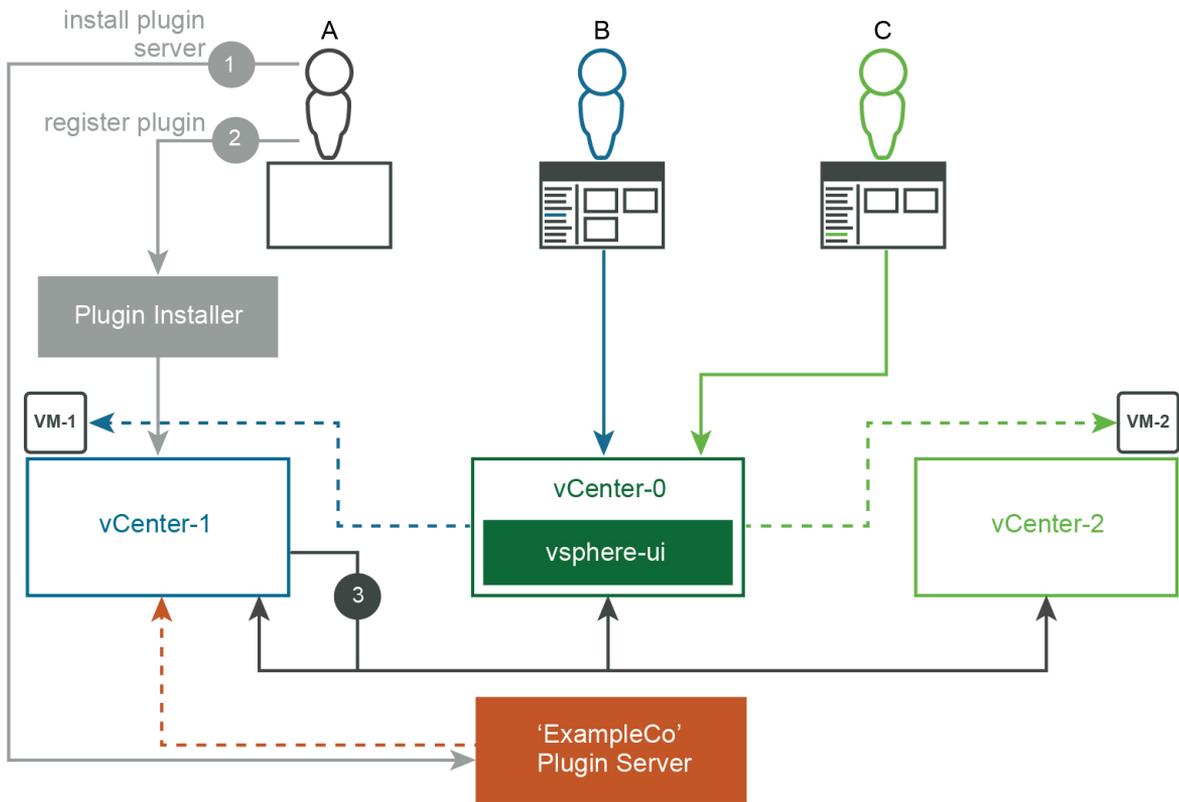
4   The `vsphere-ui` service in each vCenter Server instance downloads the plug-in manifest JSON
    of the `ExampleCo` plug-in from the plug-in manifest URL in the extension registration record:
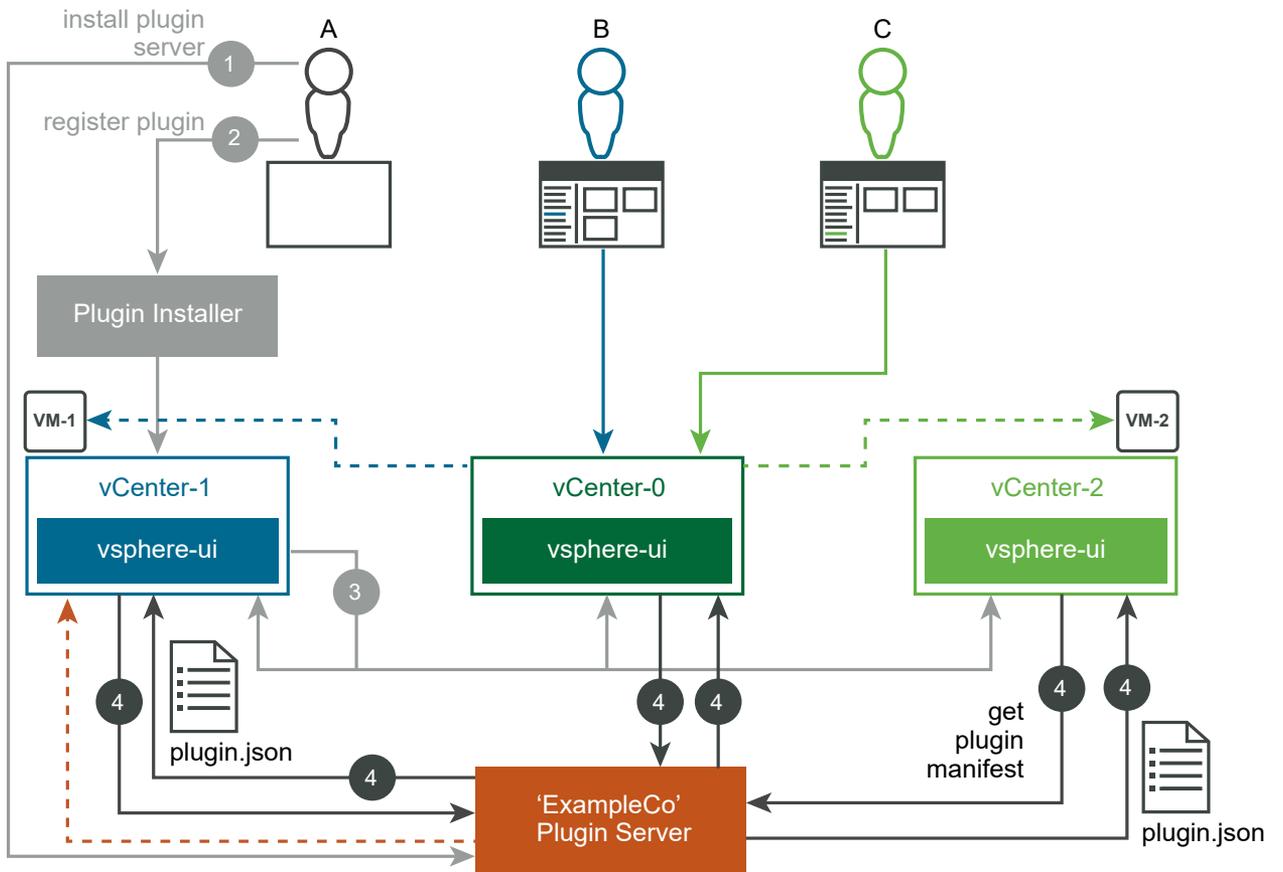
5   The vsphere-ui service in `vCENTER-0` sends notifications to currently logged in vSphere Client users (Blue and Claire). Each user sees a blue notification banner at the top of the screen:
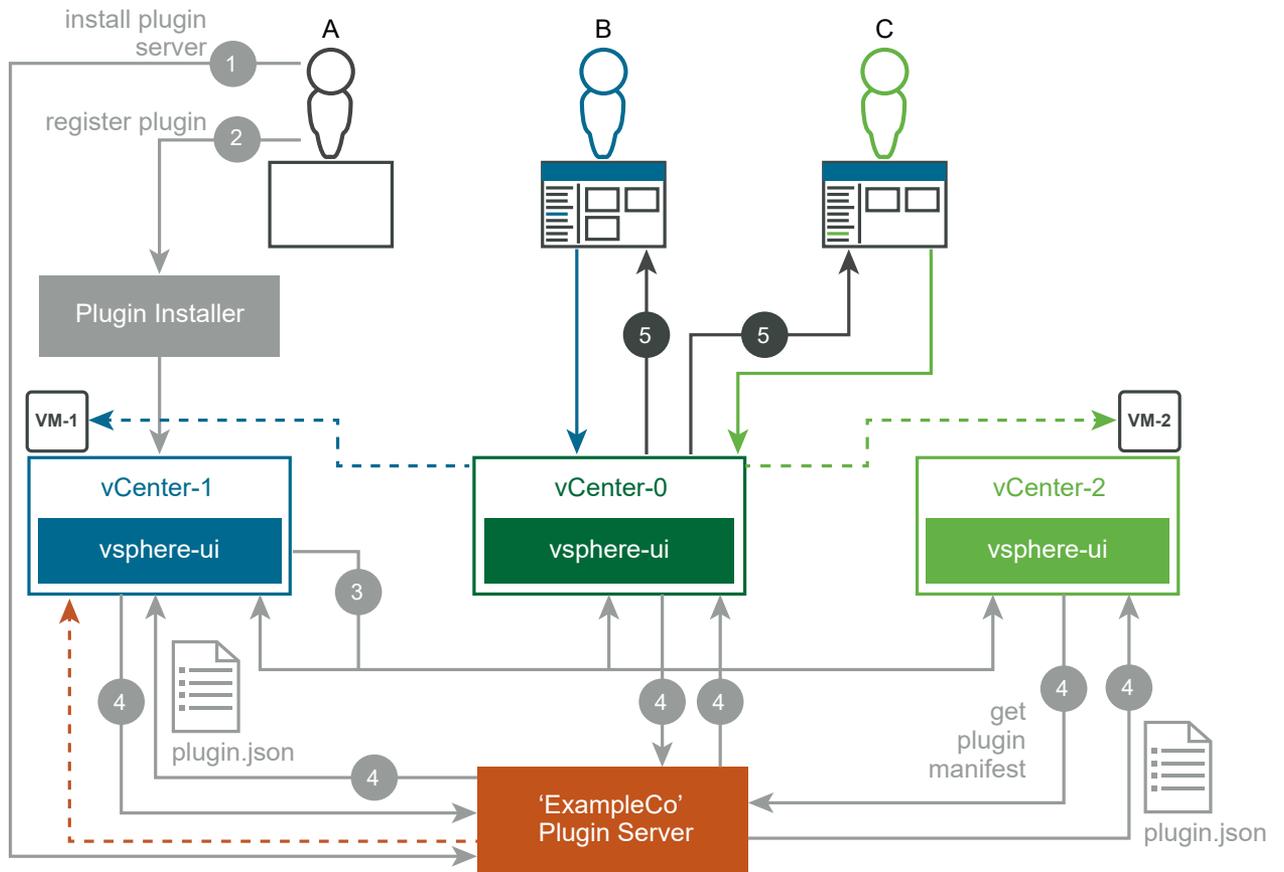
6   When Blue refreshes the vSphere Client UI in the browser, the `ExampleCo` plug-in is loaded for this user. The plug-in adds a portlet that extends the summary page of VMs. Because `VM-1` is managed by `vCENTER-1`, which has the `ExampleCo` plug-in registered, Blue sees the newly added portlet:
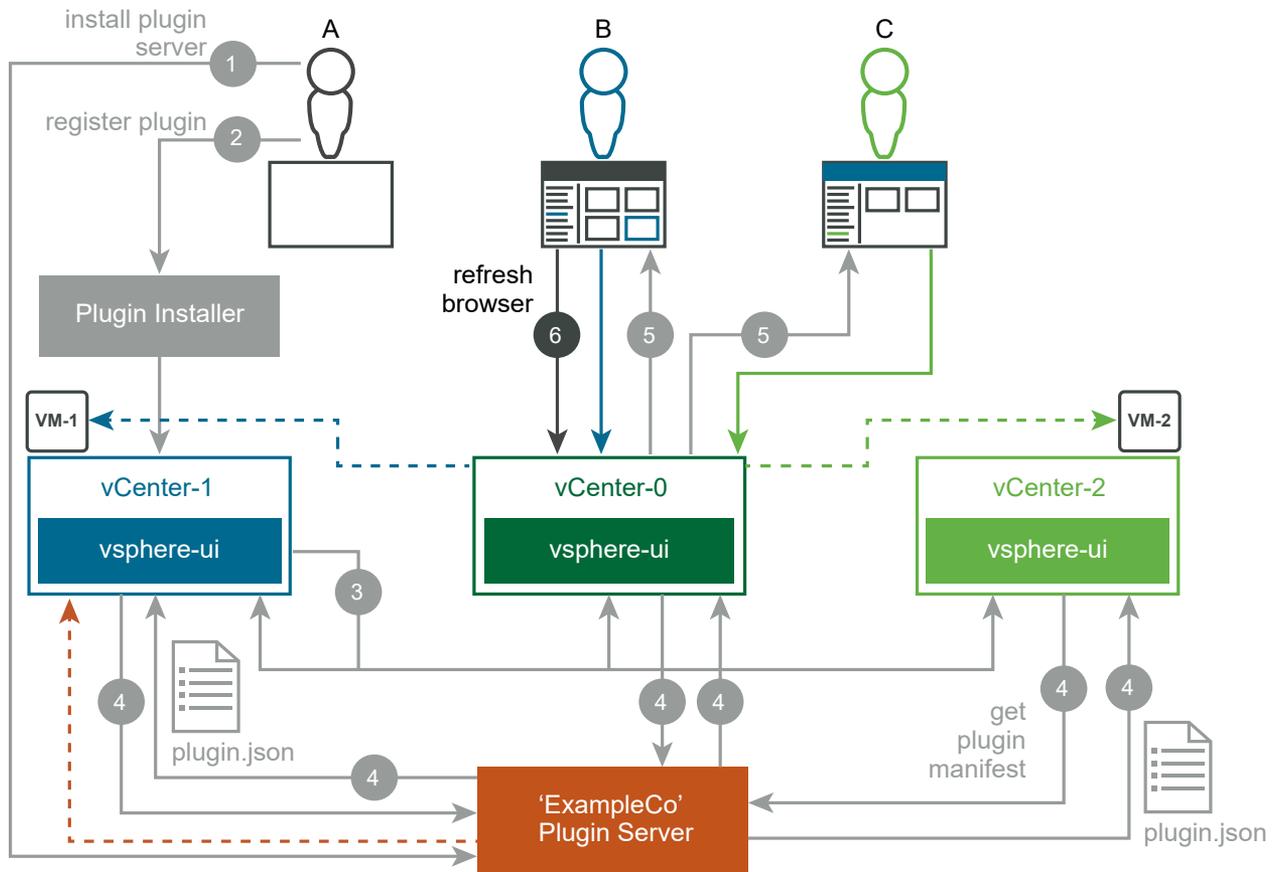
7   When Claire refreshes the vSphere Client UI in the browser, the `ExampleCo` plug-in can now
    be loaded for this user. However, Claire is looking at `VM-2`, which is managed by `vCENTER-2`.
    Because `vCENTER-2` does not have the `ExampleCo` plug-in registered, Claire does not see the
    newly added portlet.

    If Claire later navigates to a VM on `vCENTER-1`, the vSphere Client will display the portlet added
    by the `ExampleCo` plug-in.
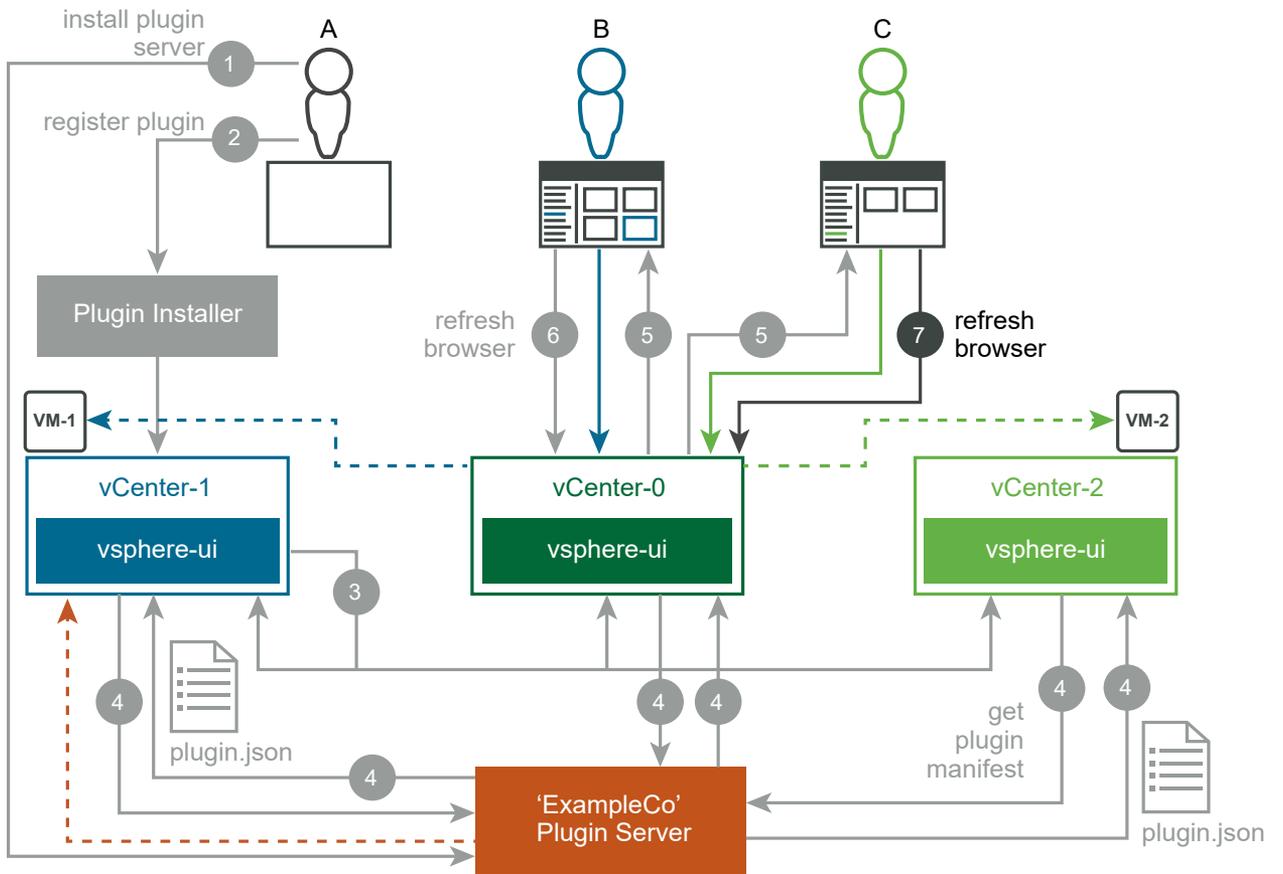
8   Alpha logs in to the vSphere UI connected to `VCENTER-0`:
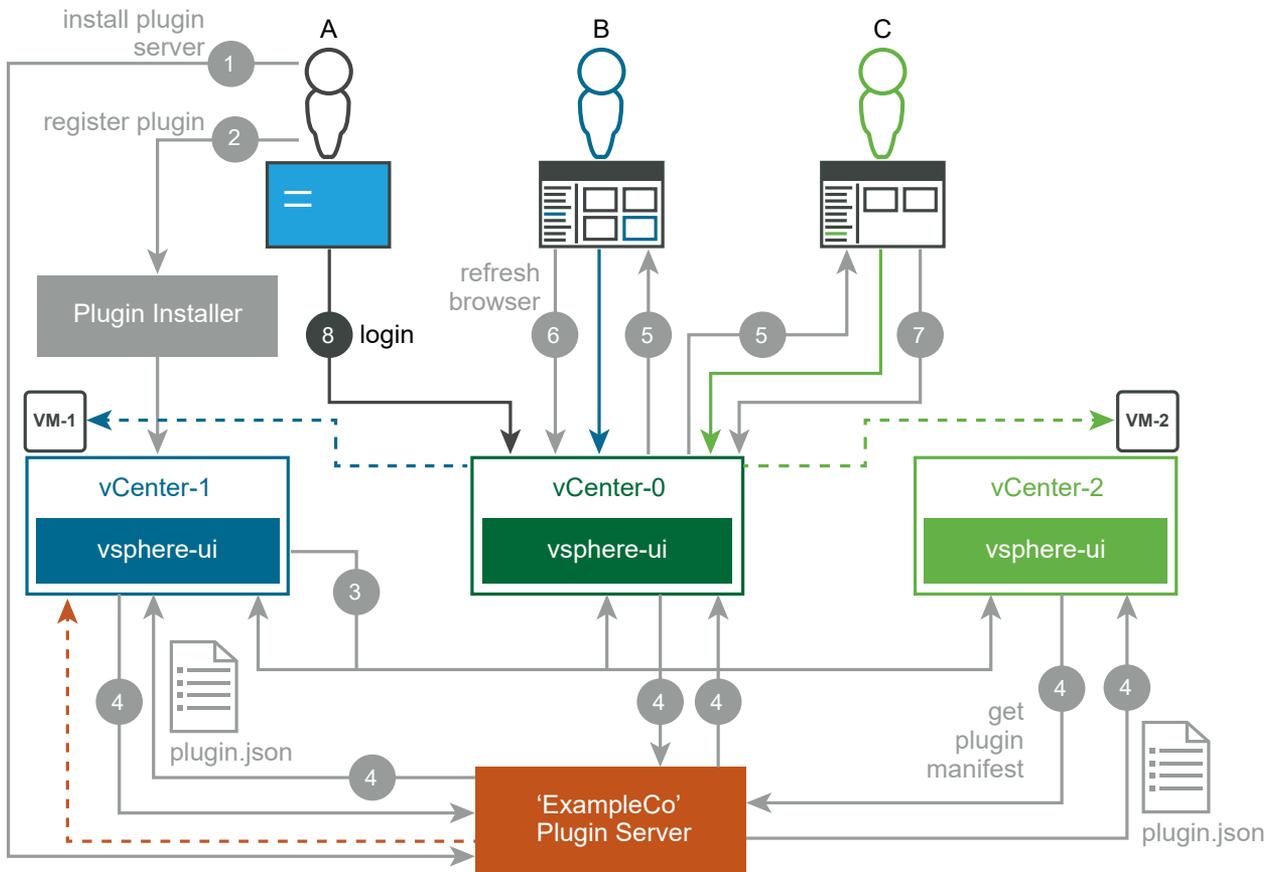
9   Alpha has completed the login and has loaded the vSphere Client UI, which displays the home screen of the vSphere Client. `VCENTER-0` deployed the `ExampleCo` plug-in prior to the login, so Alpha sees the new home page menu item and the shortcut link immediately:
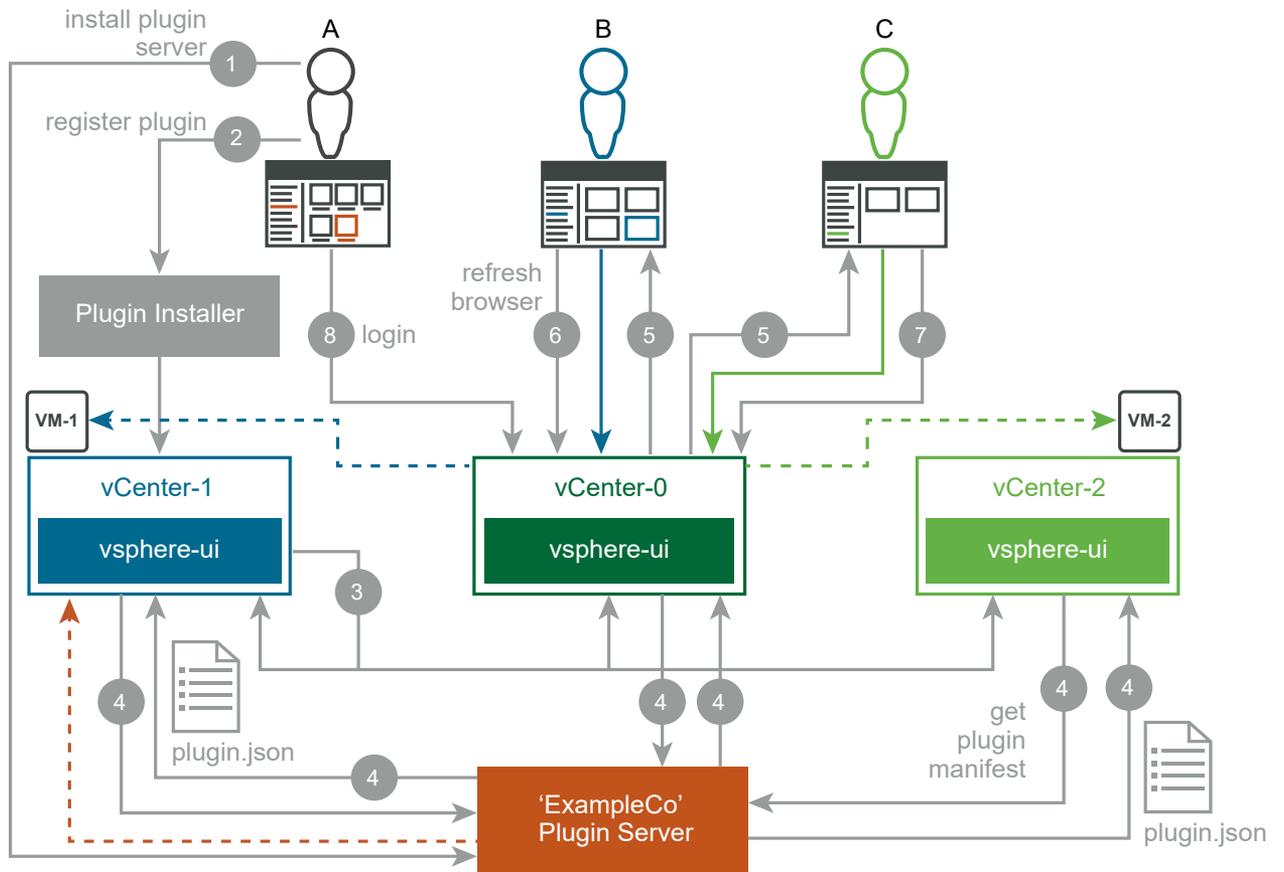
Remote Plug-in Multi-Version Support

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.

Start by creating two or more manifests as separate files, each file conforming to the schema that belongs to a supported feature set. For instance, you could create a `plugin-67.json` file that supports only vSphere Client 6.7 features; a `plugin-70.json` file that excludes a feature deprecated in vSphere Client 7.0, and a `plugin-70u1.json` file that includes a feature added in vSphere Client 7.0 Update 1.

After you test your manifest files separately, create a list of their file names as a JSON object named `manifests`. Store the JSON in a file named `plugin-multi-manifest.json`. See the following example:

```
{
   "manifests" : [
       "plugin-70u1.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.

Choosing Extension Points for vSphere Client Plug-ins

<div style="text-align: right">6</div>

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.

This chapter includes the following topics:

- 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. 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 he 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.

- **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**

    This is a single plug-in view that is displayed as a small box in the object's summary view. 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.

    You have limited control over the size of the Summary Section View. By default, the width/height ratio is 1:1, but you can specify 1:2 by using the <size> element in the plug-in manifest file.

For example, you can specify a double-height Summary Section View as follows:

```
...
  "summary": {
    "view": {
      "uri": "index.html?view-host-summary"
      "size": {
        "width": 1,
        "height": 2
      }
    }
  }
...
```

- **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.

- **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.

- **Menu**

  This is a single plug-in solution menu with one or more actions. It contains actions that apply to the currently selected object, 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.

  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 in remote plug-ins are not supported.

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",
    "icon": {
      "name": "main"
    }
  },
  "global": {
    "view": {
      "navigationId": "myGlobalViewId",
      "uri": "myplugin/globalView.html",
      "navigationVisible": false
    }
  },
  "objects": {
    "Datacenter": {
      "summary": {
        "view": {
          "uri": "myplugin/summary.html",
          "icon": {
            "name": "main"
          },
          "size": {
```

```
          "widthSpan": 1,
          "heightSpan": 2
        }
      }
    },
    "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
            }
          }
        }
      ]
    }
  }
},
"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

7

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

This chapter includes the following topics:

- Dynamic Extension Use Cases
- How the vSphere Client Displays Dynamic Extensions
- Caching Dynamic Extension Visibility
- Configure Dynamic Extensions
- Dynamic Extensions Filter Query
- Example Code for Filtering Dynamic Extensions

Dynamic Extension Use Cases

The vSphere Client supports dynamic extensions for remote plug-ins with an SPI that allows the 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 the 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 implements one or more of 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.

Dynamic Views

While the vSphere Client waits for answers, it might diplay 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 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.

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 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.

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 extensions and Configure extensions are configured in the same way. Dynamic Menu extensions are similar. Where the steps differ, the examples show both ways.

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 Monitor tab.	<pre>...
	  "objects": { ...
	    "VirtualMachine": { ...
	      "monitor": {
	      ...</pre>

For a dynamic extension in the Configure tab.	<pre>...
	  "objects": { ...
	    "VirtualMachine": { ...
	      "configure": {
	      ...</pre>

For a dynamic extension in the Actions menu.	<pre>...
	  "objects": { ...
	    "VirtualMachine": { ...
	      "menu": {
	      ...</pre>

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.

Option	Description

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

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

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

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

Option	Description

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

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

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

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

Option	Description

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

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

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

7 Add a `navigationId` (to a Monitor or Configure view) or an `Id` (to a Menu action) if not already present.

Option	Description

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

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

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

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 Monitor view, a fixed Configure view, and a dynamic action.

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": {
      "monitor": {
```

```
      "dynamicUri": "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": "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 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 the 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.

manifest fragment:
```
{"VirtualMachine":
 {"monitor":
  {"dynamicUri":"rest/items/monitor",
   "views":
   [{"navigationId":"vmMon1",
    "uri":"index.html#/vm-mon1",
    "dynamic":true}
   ]}}}...
```

POST https://*proxy-base*/rest/items/monitor
... objectIds:[
"urn:vmomi:VirtualMachine:vm-1:23ab3..."]

... [{id:"vmMon1", visible:true}]
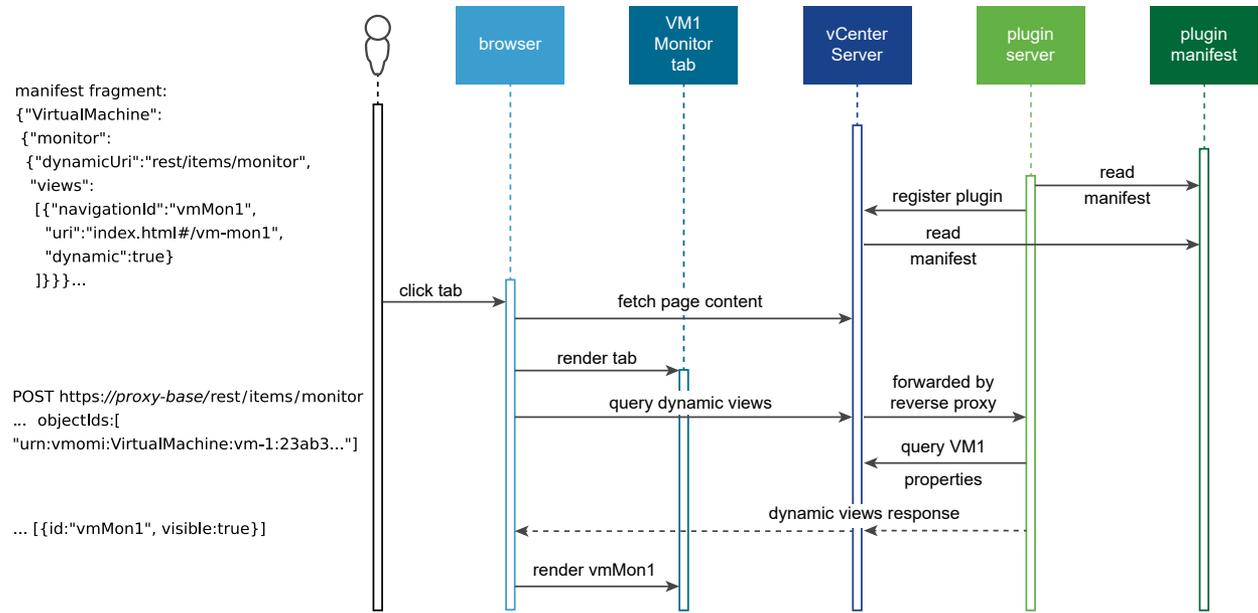
Notes about the sequence diagram:

- 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 (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 **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 `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.

  - A Boolean value that controls whether the dynamic extension will display in the user interface.

For example:

```
{
  apiVersion: "1.0.0",
  dynamicItems:
  [
    {
      id: "MonitorDynView",
      visible: true
    },
    {
      id: "TakeAction",
```

```
        visible: 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 8

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.

This chapter includes the following topics:

- 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.

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. 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.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.

Property	Type	Required?	Notes
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.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.

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. (Only valid value is administration.ceip.)
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.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>
```

```html
<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

```javascript
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

<div style="text-align:right">9</div>

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.

This chapter includes the following topics:

- 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 disable 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 disable 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.

Remote Plug-in Server Considerations for the vSphere Client

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 that serves both the plug-in manifest file and the 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 views, if needed.

- Session cloning service to authenticate with the vCenter Server Web Services API, if needed.

This chapter includes the following topics:

- Communication Paths for Authentication in the Remote Plug-in Server

- vSphere Authentication in the Remote Plug-in Server

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 process of authentication requires several steps.

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 a SOAP client session token to authenticate its operations with the Web Services API. The following diagram shows the communication paths involved in converting the vSphere Client session token to a plug-in server SOAP session token.

Figure 10-1. Plug-in Server Communication Paths for Authentication
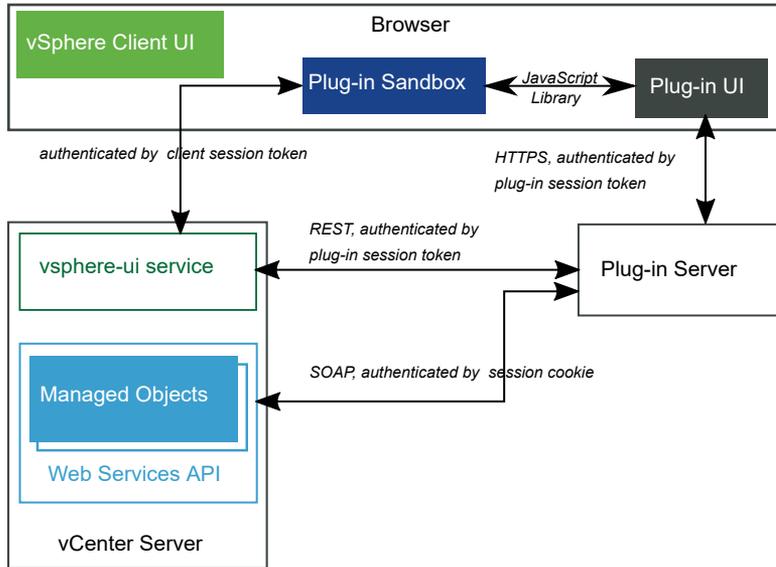
Cloning a session consists of three interactions involving the plug-in server:

1   The plug-in user interface sends its session ID and the vCenter Server endpoint 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.

vSphere Authentication in the Remote Plug-in Server

When a plug-in server accesses vSphere data, it needs to authenticate with vCenter Server. To authenticate by using the Web Services API, the plug-in server clones the user session that is currently in use by the plug-in user interface. This gives the plug-in server the same access rights as the user who is logged in with the vSphere Client.

Following are the detailed steps to clone the user session.

Note   The plug-in server must be registered with the vCenter Server instance before it can clone a session with that instance.

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.

2   The plug-in user interface calls the `app.getApiEndpoints()` method in the client JavaScript library, which returns an object containing a `uiApiEndpoint` property. The value of the `uiApiEndpoint` property is an object containing a `fullUrl` property, which contains the endpoint URL for a plug-in server REST request to the vsphere-ui service.

3   The plug-in user interface removes any query parameters and fragments from the URL, leaving the scheme, host, port, and path segments. The user interface sends both the session token value and the base URL to the plug-in server.

> **Note**   Do not hard-code the URL in the server.

4   The plug-in server builds a REST request to the vsphere-ui service. The request contains the following:

   ■   A `POST` verb.

   ■   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 server sends the REST request to the vsphere-ui service, which returns a clone ticket valid for the Web Services API of the vCenter Server instance. 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 server 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(cloneTicket);
```

7   The plug-in server uses the `key` property of the `UserSession` object to authenticate subsequent requests to the Web Services API. The server places a `vmware_soap_session` cookie in its SOAP request headers, with the session key as the cookie value.

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.

Additional Resources 11

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.

Best Practices for vSphere Client Remote Plug-ins

12

This chapter includes the following topics:

- Best Practices for Implementing Plug-in Workflows
- Best Practices for Deploying and Testing Remote Plug-ins

Best Practices for Implementing Plug-in Workflows

The following practices are recommended to assure that your plug-in provides stability and performance to users.

- If you need to temporarily store files, 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 the security of your extensions, limit the access to your plug-ins to specific users and control the user access to your extensions based on their privileges. For example, you can make your extensions available only to users who have privileges to create or delete Datastore objects. For more information, see Dynamic Extension Use Cases.

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/`

Windows OS local development environment	`%PROGRAMDATA%\VMware\vCenterServer\cfg\vsphere-client\vc-packages\vsphere-client-serenity`
Mac OS local development environment	`/var/lib/vmware/vsphere-client/vsphere-client/vc-packages/vsphere-client-serenity/`

- To avoid performance issues, make sure that your plug-in 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
vSphere Client development environment (Windows or Mac OS)	`html-client-sdk/vsphere-ui/server/logs/vsphere_client_virgo.log`
vCenter Server Appliance 6.5 installation vSphere Client	`/var/log/vmware/vsphere-ui/logs/`

  The `vsphere_client_virgo.log` file contains the log information that the 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.

Troubleshooting Remote Plug-ins for the vSphere Client

13

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.

This chapter includes the following topics:

- Plug-in Does Not Appear in vSphere Client
- Missing Entry in the Instance Selector
- Unable to Change Plug-in Manifest File
- OSGi Deployment Failure
- 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.

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.

   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 should match the characters shown in the plug-in server certificate. To find the plug-in server thumbprint, see Find the SSL Thumbprint of the Remote Plug-in Server.

- 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. However, SHA-256 is more secure, and 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

1   Set the property `remote.plugin.updateOnBrowserRefresh=true` in the `webclient.properties` file.

2   Restart the vSphere Client service on the vCenter Server.

For example,

Solution

Adding`remote.plugin.updateOnBrowserRefresh=true` to the configuration will enable the automatic redeploy of remote plug-ins on each browser refresh.

OSGi Deployment Failure

Your plug-in bundle fails to deploy.

Problem

The vSphere Client was unable to resolve a package dependency.

Cause

Various situations can cause an OSGi bundle deployment failure.

Solution

◆   Check `equinox.log` for error messages related to the OSGi deployment.

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.