

# Getting Started with HTML Client SDK (version 6.5)

## Table of content

<b>Introduction</b>	<b>4</b>
<b>1) HTML SDK Setup</b>	<b>5</b>
Requirements	5
Quick setup steps	5
vCenter Server registration	6
Build and run the samples	7
<b>2) Quick Startup Guide</b>	<b>8</b>
Documentation references	8
Starting from a new HTML plug-in project	9
Starting from an existing HTML plug-in	9
Starting from a web application	9
Testing your plug-in with the local HTML client	10
Plug-in packaging	10
Hot deployment	11
Testing with a local Flex client	11
Testing with a remote HTML or Flex client	12
Checking your plug-in for OSGi problems	12
Location of Virgo server logs	13
<b>3) HTML plug-in development</b>	<b>14</b>
Versions compatibility	14
Plug-in architecture diagram	14
Javascript API	15
Javascript framework and components	16
Java API and Java services	16
Summary portlets support	16
Hybrid plug-ins	16
HTML plug-ins compatibility guidelines	17
Note on plug-in context path	18
Note on external icons support	18
Note on menus and sub-menus	19
Packaging and registering a plug-in	21
Quick test instructions	21
vCenter server plug-in registration tool	21
<b>4) Upgrading from SDK 6.0.2</b>	<b>23</b>
Changes to web-platform.js	23
Changes to *.html files	23
Changes to *.js files	24
Changes related to “external icons”	24
Changes related to menu extensions	25
Testing your changes	26
Resolved issues in 6.0.2 samples	26
<b>5) Upgrading from SDK 5.5.x, 6.0, 6.0.1</b>	<b>26</b>
Upgrading from SDK 6.0.1	26

Upgrading from SDK 6.0	27
Upgrading from SDK 5.5.x	27
<b>6) Known issues</b>	<b>28</b>
API differences with SDK 6.0 HTML Bridge	28
General issues	28
HTML plug-ins issues when running in the Flex client	29
<b>7) How to send feedback or report problems</b>	<b>29</b>

In this document **vSphere Client** refers indifferently to **vSphere Web Client**, the current Flex-based client, or to **vSphere Client (HTML)**, the new HTML-based client. **Flex client** and **HTML client** are used as shortcuts for each type of client. Also, **vCSA** is used as a shortcut for vCenter Server Appliance.

# Introduction

Thank you for using the new HTML SDK for vSphere Client! This SDK allows to build vSphere Client plug-ins that are 100% compatible between the vSphere Client (HTML) 6.5 release and vSphere Web Client 6.0 or 6.5.

You can start creating new plug-ins or upgrade your existing plug-ins from an older SDK version. Please test your plug-ins with both the HTML client and the Flex client, we welcome [your feedback](#) and are committed to making your plug-in development successful!

## What is the big picture for people new to vSphere client plug-in development?

See the architecture diagram in [3\) HTML plug-in development](#).

## What is the difference with the “HTML Bridge” found in Web Client SDK 5.5.x - 6.0.x?

If you are familiar with the HTML Bridge from older SDKs you won't see any difference in terms of APIs but some differences in terms of packaging. The HTML SDK reuses the same APIs on purpose, it is the only way to provide compatibility with vSphere Web Client 6.0 and 6.5!

If you have only used the Web Client SDK in the past, or are new to plug-in development, the HTML SDK is the way forward for any solution that extends the vSphere client! HTML Bridge was released in March 2014 as part of the Web Client SDK 5.5.1. It's goal was to support HTML plug-ins while still running inside a Flex client. Soon the HTML client will become the primary vSphere UI and the HTML SDK guarantees compatibility.

## What happens to old plug-ins using Flex?

Since the HTML client cannot display any Flex content old plug-ins need to migrate their Flex content to HTML/Javascript. A [hybrid plug-in](#) with both Flex and HTML UI will still work on the HTML client but all Flex views will be ignored. Other extensions are not affected and work with no change (menus, custom objects, data extensions). Note that existing Flex plug-ins in the HTML client won't be deployed because their `plugin-package.xml` don't include the attribute `type="html"`.

## Are there new APIs in the HTML SDK?

No, this SDK contains only APIs that are compatible with both clients. A future version may add new APIs only for the HTML client but the goal of this first release is keep plug-ins compatible with both clients.

## Is it possible to build plug-ins that can be deployed only for the HTML client?

Not at this time. End-users will deploy your plug-in solution the same way regardless of the type of vSphere client used. In the near future we will provide a way for HTML plugin to be deployed only with the HTML client.

---

# 1) HTML SDK Setup

## Requirements

To get started with your HTML plug-in development you need the following:

- A Windows or Mac computer with a minimum of 8GB of RAM (16GB recommended) and 20GB of free disk space.
- The latest HTML Client SDK release
- Access to a vCenter Server for Windows or vCenter Server Appliance 6.0
- A good understanding of web application development using HTML/Javascript.
- A basic understanding of Java development.

Download `vsphere-client-sdk-6.5.0-<build number>.zip` . When unzipping the build file on your development machine we recommend that you pick a top directory without spaces because long paths and paths with spaces may cause server deployment errors or script errors.

The HTML Client SDK content looks like this below (this is subject to change):

```
vsphere-client-sdk/html-client-sdk
  README
  html-client-sdk-version.xml
  /docs                Java API doc
  /libs                Common libraries used to build plug-ins
  /resources           Scripts, configuration files, minimal Flex sdk
  /samples             Various HTML plug-in samples
  /tools               Additional tools
  /vsphere-ui
    /plugin-packages   Core plug-in packages of the HTML client
    /server            Runtime server (local HTML client)
```

## Quick setup steps

Install [Java JDK](#): version 1.7.0\_21 or above for vCenter 6.0, version 1.8.0 for vCenter 6.5.

- Set the `JAVA_HOME` env variable to the local JDK location.
- Java version 1.8 is required for your local Virgo runtime to work with the vCenter 6.5 release. From a plug-in compilation standpoint it doesn't make any difference because the build script must have Java target version set to 1.7 for the plug-in Java bundle to be compatible with vCenter 6.0.

Install [Apache Ant](#) (version 1.8.0 or above) as it is used by the build scripts.

- Set the env variable `ANT_HOME` to your local Ant folder on your disk.
- Set the env variable `VSPHERE_SDK_HOME` env to your `/html-client-sdk` directory.

Use the IDE of your choice.

- You can pick any IDE or just use a code editor. Since the UI code is in Javascript your UI debugging will be done directly in the browser. A Java debugger will be useful when developing a service that runs in the Java layer (see [Plug-in architecture diagram](#)).
- Popular IDEs are [IntelliJ IDEA](#), [Eclipse](#) or [Spring Tool Suite](#) (based on Eclipse). If you pick Eclipse or Spring Tool Suite complete your setup with `docs/Eclipse-Setup.html`.

**Adobe Flex SDK note:** A minimal version of the Adobe Flex SDK is included in this SDK at `/resources/flex_sdk_4.6.0.23201_vmw` because the Flex compiler is still needed to build plug-in string resources as `.swf` files and keep the plug-in compatible with the vSphere Flex client. You do not need to install the full Adobe Flex SDK as long as you are only building HTML plug-ins or updating hybrid plug-ins.

## vCenter Server registration

*This setup section is required each time you pick a different vCenter Server to test your plug-in locally.*

The vCenter Server registration consists of generating special files that will connect the local client in your dev environment (the Virgo server) to the remote vCenter instance. The registration files are created with the scripts **dev-setup.sh** or **dev-setup.bat** located in `tools/vCenter registration scripts`. These scripts must be copied to the vCenter host machine to which you will register, using for instance WinSCP on Windows and Cyberduck on Mac OS (or the ssh command line).

1. For vCSA: copy `dev-setup.sh` to your vCSA root directory and make it executable.  
For vCenter Server for Windows: copy `dev-setup.bat` to the administrator directory.
2. Run the dev-setup script.  
Three files get generated: `webclient.properties`, `store.jks` and `ds.properties`.
3. Copy these files to your dev machine at the following locations:

`webclient.properties`

Windows: `C:\ProgramData\VMware\vCenterServer\cfg \vsphere-client\`

Mac OS: `/var/lib/vmware/vsphere-client/vsphere-client/`

`store.jks`

Windows: `C:\ProgramData\VMware\vCenterServer\cfg \`

Mac OS: `/var/lib/vmware/vsphere-client/`

ds.properties

Windows: C:\ProgramData\VMware\vCenterServer\cfg\vsphere-client\config\

Mac OS: /var/lib/vmware/vsphere-client/vsphere-client/config/

4. Set the environment variable `VMWARE_CFG_DIR` to the following location:

Windows: C:\ProgramData\VMware\vCenterServer\cfg\

Mac OS: /var/lib/vmware/vsphere-client/

5. **Mac OS only:** Change `keystore.jks.path` in `webclient.properties` and `solutionUser.keyStorePath` in `ds.properties` to use the correct path for Mac OS, i.e.

/var/lib/vmware/vsphere-client/store.jks

6. **Mac OS first time only:** Edit `vsphere-ui/server/configuration/tomcat-server.xml`

And replace the Windows path for `keystoreFile` with

`keystoreFile="/var/lib/vmware/vsphere-client/store.jks"`

## Verification of your vCenter Server registration

Start the local Virgo server from a Mac OS or Windows terminal using the startup script:

```
vsphere-ui/server/bin/startup.[sh,bat] -debug
```

Check the server log at `vsphere-ui/server/serviceability/logs/vsphere_client_virgo.log`. The server should take a couple of minutes to start. When the log stops scrolling (i.e. all bundles have been deployed) open a browser, go to <https://localhost:9443/ui> and login into your local HTML client.

Note that at <https://localhost:9443/ui> you should see the same vSphere inventory as the inventory you see when you login into the “remote” Web Client hosted with vCenter at `https://<VCENTER_IP>/vsphere-client` or the HTML client at `https://<VCENTER_IP>/ui`

## Registering another vCenter Server

Repeat the steps above for another vCSA or vCenter Server for Windows setup. It is recommended to make a copy of the registration directory (\*) beforehand and empty the current one, since other files get generated at runtime in this local cache. This way you can easily switch between different vCenter setups, by renaming the directory and restarting the Virgo server.

(\*) on Mac OS: `/var/lib/vmware/vsphere-client`

on Windows: `C:\ProgramData\VMware\vCenterServer\cfg\vsphere-client\`

## Build and run the samples

The best way to validate your development environment is by rebuilding the samples. The current samples are `globalview`, `chassisA`, `chassisB`, `vsphere-wssdk`. They are already included by default with other core plug-ins in `html-client-sdk/vsphere-ui/plugin-packages` so that you can review them. Here are the steps to rebuild them and override the pre-built versions:

- Build from the command line:

- Go to `samples/`

- Run `./build-all.sh` on Mac or `build-all.bat` on Windows.
- The build output is in `samples/build/plugin-packages`
- Copy the folders from `samples/build/plugin-packages` to `vsphere-ui/plugin-packages` (confirm that you want to replace existing files)  
Note: `vsphere-ui/plugin-packages` is where local plug-ins are picked up automatically when the server starts.
- Start or restart the Virgo server from the command line:
  - Run `vsphere-ui/server/bin/startup.[sh,bat] -debug`
  - Check that there are no errors in the console or in the [Virgo logs](#).
- Login to your local HTML client at <https://localhost:9443/ui/> to see the samples in action.
  - See the Troubleshooting section in `docs/FAQ.html` in case of problems.

To stop running samples in your local client, delete the sample folders from `vsphere-ui/plugin-packages` and restart the server, the ones with the name ending with `-sample`. **Do not delete** other plugin-packages!

---

## 2) Quick Startup Guide

Now that you have verified your HTML SDK setup by rebuilding the samples and running your local client connected to a vCenter Server instance the next step depends on your SDK experience and whether or not you already built an HTML plug-in with a prior SDK (and the HTML Bridge API):

- If you already have some HTML plug-in code go to [Starting from an existing HTML plug-in](#) below, update your code and test your new version so that you can [report issues](#) and share your feedback.
- If you have built a Flex plug-in before you should be familiar with the SDK concepts and documentation. Spend time discovering the HTML APIs and review how to convert your Flex UI to an HTML version. You will also be interested in the concept of [hybrid plug-ins](#) discussed later in this document.
- And if you are brand new to plug-in development, start a brand new project, get familiar with the API and post questions if you need help. We realize that your biggest challenge will be to navigate the documentation while we can only provide temporary draft versions.

## Documentation references

The HTML Client SDK release includes multiple doc files:

- ❑ `docs/Eclipse-Setup.html`: additional setup instructions for Eclipse
- ❑ `docs/Extension-Points.html`: public extension points (same as SDK 6.0.2)
- ❑ `docs/FAQ.html`: frequently asked questions
- ❑ `docs/javadoc/index.html`: Java API documentation (same as SDK 6.0.2)
- ❑ `docs/Javascript-API.html`: Javascript API documentation (same as HTML bridge in SDK 6.0.2)



❏ docs/Tutorial.html: getting started doc when using the Eclipse IDE

See also the online [Web Client Programming Guide](#) or generate a PDF.

## Starting from a new HTML plug-in project

The easiest way to get started is to create a plug-in template in order to understand the major components, the build environment and the deployment process.

Plug-ins have a UI component running in the browser and a Java component running on the server. The HTML SDK provides tools to create a project template for each of those two components. If you use Eclipse follow docs/Tutorial.html and launch the built-in wizard to create the HTML and Java projects. Otherwise use the `create-html-plugin` scripts in `tools/Plugin generation scripts/`, launch `create-html-plugin.[sh,bat]` on the command line and follow the prompts. This will create a `<pluginName-ui>` project and a `<pluginName-service>` project.

Then run the build scripts to verify that this simple plug-in works:

- Run `./build-war.[sh, bat]` in the UI project directory. This creates a plug-in .war bundle.
- Run `./build-java.[sh, bat]` in the Java project directory. This creates a plug-in .jar bundle
- Continue with the steps below in section [Testing your plug-in with the local HTML client](#).

The plug-in template comes with two simple views, a *global view* containing a button that calls a simple *EchoService* on the java side, and a *Host monitor view* extension which display some host properties.

From here you can start modifying the plug-in code for your own purposes. Go to [HTML plug-in development](#) below which covers some general information. Then review the [Web Client Programming Guide](#) to learn about view, menu and data extensibility in details.

## Starting from an existing HTML plug-in

Old HTML plug-ins require changes because the web context-path is different between the HTML client and the Flex Client. Although you need to rebuild your plug-in no API was changed and you gain full compatibility with both clients.

- For plug-ins created with SDK **5.0.1** to **6.0.1** please see [5\) Upgrading from SDK 5.5.x, 6.0, 6.0.1](#) first because of important changes documented between versions 5.0.1 and 6.0.2.
- For plug-ins created with SDK 6.0.2 (i.e. you either copied an existing 6.0.2 sample or you used the Eclipse wizard to generate it) please see [4\) Upgrading from SDK 6.0.2](#).

As you make changes to your existing plug-in be sure to compare with the SDK samples or to a new plug-in template to verify that you follow the same rules.

## Starting from a web application

If you have a web application from which you would like to re-use UI code here are the basic steps to transform it into an HTML plug-in:

1. You can keep your existing project generating one war file for both UI and Java code. There is no need to have two separate bundles.
2. Decide which view or action extensions your plug-in should be using to display the UI. Add `plugin.xml` containing those extensions and pointing to your existing UI code.
3. The .war bundle must be OSGI-ready. Edit your bundle's `MANIFEST.MF` to include `Bundle-Name`, `Bundle-SymbolicName`, `Web-ContextPath` (`vsphere-client/yourpluginname`), and the correct list in `Import-Package`.
4. Add the required `sessionManagementFilter` in `web.xml` as shown in the samples.
5. Update the code as necessary to take advantage of Javascript APIs, or Java APIs.

## Testing your plug-in with the local HTML client

The first way to test your new plug-in is with the local HTML client that comes with HTML SDK. Once it is working locally you should test it also on the Flex client (local or remote) and on a remote HTML client.

### *Plug-in packaging*

The UI and Java code that makes up each plug-in must be put together into a plug-in *package* folder before it can be deployed and tested. See the directory `html-client-sdk/vsphere-ui/plugin-packages` for examples of such folders with each plug-in name (`chassisA-sample`, `chassisB-sample`, etc.)

A plug-in package folder has one `plugin-package.xml` at the root and a `plugins` directory containing the jar and war bundles.

If you created a HTML plug-in using the script mentioned [above](#) open the generated `plugin-package.xml` in your `<pluginName-ui>` directory, it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<pluginPackage id="com.mycompany.mypluginui" version="1.0.0" type="html" name="mypluginui"
  description="Add plug-in description" vendor="Add vendor">
  <dependencies>
    <!-- Minimum vSphere Client version compatible with this plug-in -->
    <pluginPackage id="com.vmware.vsphere.client" version="6.0.0" />
  </dependencies>
  <bundlesOrder>
    <bundle id="com.google.gson" />
    <bundle id="com.mycompany.mypluginui.myplugin-service">
  </bundlesOrder>
</pluginPackage>
```

### **Important:**

- Pick a unique id for the `pluginPackage`
- `type="html"` is required for the plug-in to be deployed by the HTML client.
- Keep a dependency on `pluginPackage id="com.vmware.vsphere.client" version="6.0.0"` to ensure that the plug-in will be deployed in Flex client 6.0.


In the same `<pluginName-ui>` directory you can use the script `build-plugin-package.[sh, bat]` to create the plugin package folder.

Install your plug-in locally by copying its package folder into `html-client-sdk/vsphere-ui/plugin-packages` and restarting the server. The virgo server picks up all plug-ins located in `plugin-packages` by default.

## Hot deployment

Copying a plug-in folder to `html-client-sdk/plugin-packages` is the standard way to install a plug-in *locally* but it requires to restart the Virgo server for each update. Here are other ways to speed up your development cycle:

- The plug-in's `.war` and `.jar` bundles can be hot-deployed by copying the files directly in `html-client-sdk/vsphere-ui/server/pickup`
  - See the server console being updated by the bundle deployment.
  - You can repeat that for each new version of the `.war` and `.jar` files
  - Refresh your browser so that the new plug-in code is loaded
  - In case of errors it is best to revert to a clean state and restart the server
- If you use Eclipse, follow the instructions in `docs/Tutorial.html` to deploy UI and Java bundles using the Virgo tooling.

 Hot deployment has one known bug at the moment: the bundle's string resources are not reloaded. So any change of string resources won't be visible until you restart the server.

## Testing with a local Flex client

Even if the HTML client is becoming the primary vSphere client you cannot prevent end users from deploying your plug-in in the Flex client, so it is a good idea to verify that it works properly in both!

It is easier to test your plug-in with a local Flex client before you go through the registration steps and test it with remote clients (see next section).

If you don't have the Web Client SDK 6.0.2 installed locally download build 3633101 at <https://developercenter.vmware.com/web/sdk/60/web-client>. Unzip the files in a directory called `flex-client-sdk`. Copy your plug-in package folder into `flex-client-sdk/plugin-packages` and restart the local server with the command line `flex-client-sdk/server/bin/startup` (stop the other server running from `html-client-sdk/vsphere-ui/server` first).

Go to <https://localhost:9443/vsphere-client> and login into your local Flex client. Check the [Virgo logs](#) first in case of problems.

## Testing with a remote HTML or Flex client

In order to test your plug-in with the Flex client hosted on a vCSA 6.0 setup, or 6.5 (RC or above), or with your HTML client setup it needs to be registered with vCenter Server. The registration process for HTML plug-ins is the same as the one for Flex plug-ins.

See [Packaging and registering a plug-in](#) below for more details about plug-in registration with vCenter and for “quick test” instructions.

After the registration is done, for vCSA 6.0 or 6.5 you can login at `https://<VCENTER_IP>/vsphere-client` for the Flex client. Keep the browser’s Network console opened to check for errors in your plug-in’s requests.

Check the [Virgo logs](#) to make sure your plug-in was deployed correctly. Search your *plugin-package id* and your plugin bundles’ *Bundle-SymbolicName* (from MANIFEST.MF).

## Checking your plug-in for OSGi problems

The [Best practices guide](#) is part of the vSphere Web Client documentation which is intended to provide guidelines on how to develop safe and robust Web Client plug-ins. In addition to this the HTML Client comes with an automatic “OSGi usage validation” feature that helps you detect dependency problems and possible interoperability issues of your plug-in with other plug-ins and the HTML Client itself.

The OSGi usage validation runs immediately after the deployment of all plug-ins is completed and validates each plug-in. If your plug-in is registered as a vCenter extension its deployment and validation will start after you login.

The OSGi usage validation output is logged into a dedicated `plugin-medic.log`. It is located in the same folder as `vsphere_client_virgo.log` (see [Location of Virgo server logs](#)).

The following categories of common OSGi issues are covered:

- Clashing plugins: when two or more plug-ins contain bundles that export the same package with different versions or with no version. This could lead to race conditions causing `NoClassDefFoundError` during deployment or at runtime. The OSGi usage validation may report the following warnings:
  - Deployed clashing plug-ins found in the currently running HTML Client  
Conflicting package exports [Package: <package-id>;version="<version>"; Clashing bundles: <bundle-id>:<version>(<plugin-id>:<version>), <bundle-id>:<version>(<plugin-id>:<version>)]
  - Exported packages with no version  
Export of packages with no version specified [Bundle: <bundle-id>; Unversioned packages: <package-id>, <package-id>]
  - Imported packages with no version  
Import of packages with no version specified [Bundle: <bundle-id>; Unversioned packages: <package-id>, <package-id>]
- Unsafe dependency handling: when using OSGi features not recommended for HTML Client plug-ins. This could result in unexpected runtime behavior. The OSGi usage validation may report the following warnings:

- **Unnecessary usage of `DynamicImport-Package` in your bundle manifest**  
`DynamicImport-Package` should be avoided [Bundle: <bundle-id>; Dynamic imports: <package-id>]
- **Unnecessary usage of `Require-Bundle` in your bundle manifest**  
`Require-Bundle` usage in place of `Import-Package` is discouraged [Bundle: <bundle-id>; Required bundles: <bundle-id>, <bundle-id>]
- **Exporting a missing package or package that belongs to another library your bundle uses**  
Exporting nonexistent packages or packages from nested or imported JARs [Bundle: <bundle-id>; Exported foreign packages: <package-id>]
- **Nesting the same library within multiple bundles (instead of reusing it as a separate bundle)**  
JARs with one and the same name are found in multiple bundles [JAR: <library-name>.jar; Nested in bundles: <bundle-id>, <bundle-id>]
- **Naming inconsistencies: when there is a mismatch between `Bundle-SymbolicName`, `Bundle-Vendor` and/or the reverse company name of the bundle's exported packages. This could interfere with the namespace of other plug-ins and cause unexpected runtime behavior. The OSGi usage validation may report the following warnings:**
  - **Non Matching usages of your company name**  
Packages exported with reverse company name other than the one specified in the bundle symbolic name [Bundle: <bundle-id>; Packages: <package-id>]
  - **Misuse of VMware's namespace for bundles and packages (`com.vmware`)**  
Don't use 'com.vmware' prefix for bundle symbolic names and packages [Bundle: <bundle-id>; Bundle vendor: <your-company-name>]  
**or**  
Don't use 'com.vmware' prefix for bundle symbolic names and packages [Bundle: <bundle-id>; Misnamed exported packages: <package-id>]

**Note:** OSGi usage validation warnings do not necessarily indicate problems with your plug-in. Nevertheless, they should be considered to improve your plug-in in the specified areas. This could also help in the certification process of your plug-in.

For hints about fixing warnings discovered by the OSGi usage validation review the current HTML Client SDK [samples](#) as well as the OSGi-Specific Recommendations section of the [Best practices guide](#).

## Location of Virgo server logs

The Virgo server logs are the first place to check for problems because they will indicate if your plug-in bundles were deployed correctly or if an error happened on the Java side.

Environment	Virgo logs location
HTML SDK dev setup (Mac or Windows)	html-client-sdk/ vsphere-ui/server/serviceability/logs/vsphere_client_virgo.log
Flex SDK dev setup (Mac or Windows)	flex-client-sdk/ server/serviceability/logs/vsphere_client_virgo.log

HTML client on vCenter	/usr/lib/vmware-vmware-vmware-client/server/serviceability/logs/
vCSA 6.5 (HTML client)	/var/log/vmware/vsphere-ui/logs/
vCSA 6.0 and 6.5 ( <b>Flex</b> client)	/var/log/vmware/vsphere-client/logs/
vCenter for Windows 6.0 and 6.5 ( <b>Flex</b> client)	C:\ProgramData\VMware\vCenterServer\logs\vsphere-client\logs

For general troubleshooting tips see <docs/FAQ.html#trouble-shooting> .

## 3) HTML plug-in development

### Versions compatibility

Plug-ins built with SDK version N are compatible with vSphere client version M  $\geq$  N, including the new HTML client.

	in Flex client 5.5.x	in Flex client 6.0.x	in Flex or HTML client 6.5
HTML plug-in w/ SDK 5.5.x is	compatible	compatible (1)	compatible (1)
HTML plug-in w/ SDK 6.0.x is	not compatible	compatible	compatible (2)
HTML plug-in w/ HTML SDK is	not compatible	compatible	compatible

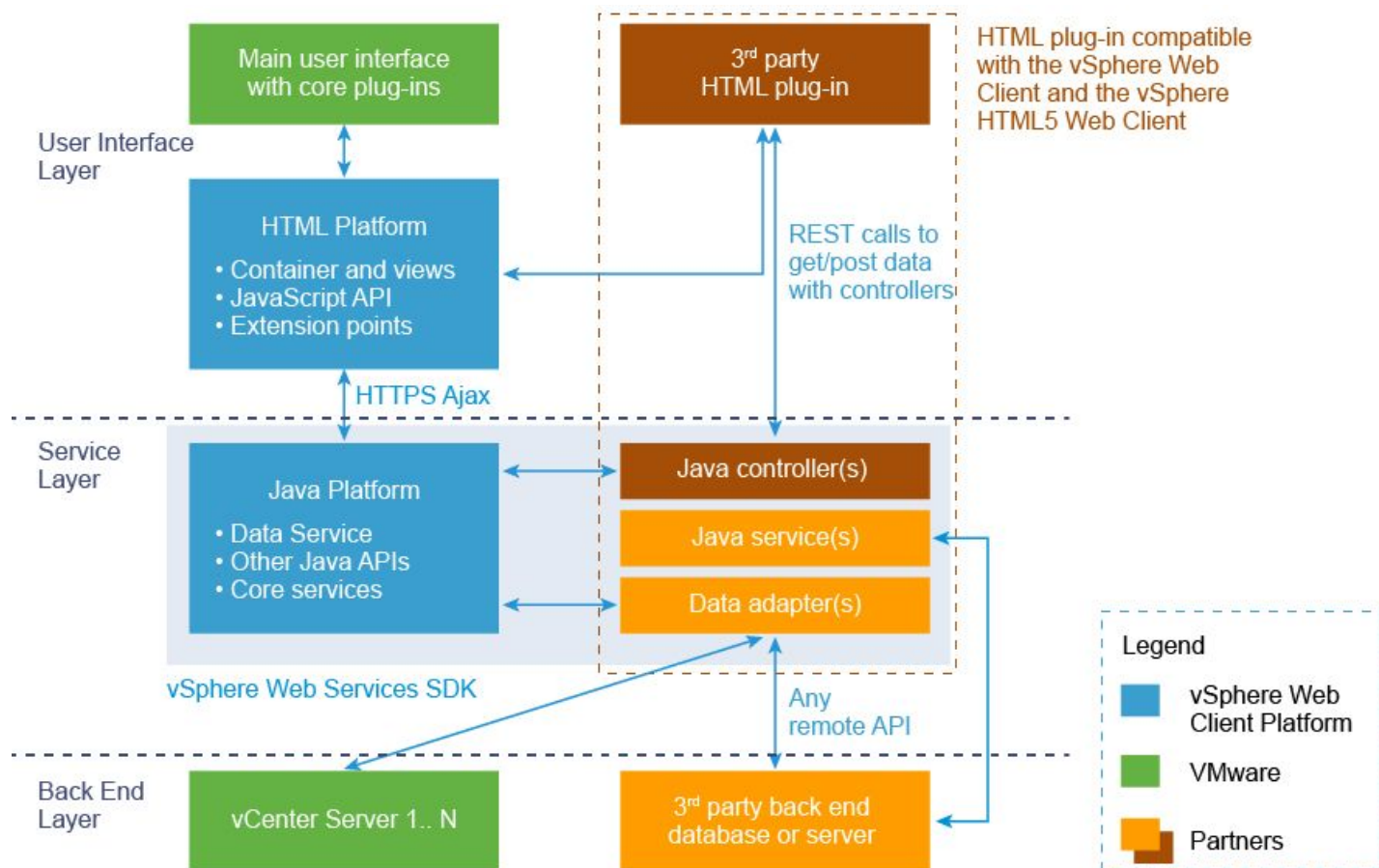
- (1) HTML plug-ins built with SDK 5.5.x should work in 6.0 or the upcoming 2016 release if you followed the 6.0.x SDK release notes or the [checklist](#) below.
- (2) HTML plug-ins built with SDK 6.0.x should work with the 2016 Flex client and 2016 HTML client if you follow the [checklist](#) below.

### Plug-in architecture diagram

The following diagram shows the 3 main layers: the browser layer running UI code, the middle-tier server where multiple browser sessions can connect, and the back-end servers, vCenter and others.

For a 3rd party plug-in's point of view there is no real difference between running inside the new HTML client or the Flex client. By using the Javascript API supported by both vSphere clients your plug-in is *de-facto compatible* with both of them! In the current release all HTML APIs are supported by both clients.

The other important point is that the Java layer is the same between the two clients, the API hasn't changed. So when migrating an existing Flex plug-in and moving the UI component to HTML/Javascript you can re-use all your java services and data adapters. You will only need to add thin java controllers that take care of routing REST calls into those services.



## Javascript API

The Javascript API called *HTML Bridge* before was first introduced with vSphere Web Client SDK 5.5.1 in March 2014. It provides the same level of functionality as the Flex API and is the only thing needed to build HTML plug-ins compatible with the new HTML client.

Two APIs were added since version 6.5 and kept backward compatible with 6.0.

- `WEB_PLATFORM.getClientType()` returns "flex" when the plugin runs in the Flex Client and "html" when it runs in the HTML Client.
- `WEB_PLATFORM.getClientVersion()` returns a version in format "6.0", "6.5", "6.5.1" etc.

See the complete API documentation at [docs/Javascript-API.html](https://docs.vmware.com/en/VMware-vSphere/6.5/HTML-Client-SDK/JavaScript-API.html)

## Javascript framework and components

A key aspect of view extensions is the fact that they are “sandboxed” inside iFrames. This sandboxing was necessary initially to display HTML content “over” the Flex client. With the HTML client the same sandboxing ensures that plug-in views are isolated from the rest of the client web application. The advantage is that each view can use the UI technology of its choice without interfering with the client’s main HTML document.

The current samples are based on jQuery but you can pick any modern framework to implement your content.



You are also free to use any library to implement UI components within your views. The HTML SDK doesn't provide any specific component.

## Java API and Java services

The Java layer, i.e. the services running in the Virgo server such as data adapters, remains the same between the Flex client and the HTML client. See `docs/Tutorial.html` for an introduction on the Java layer of a plug-in project.

## Summary portlets support

HTML content is not supported in portlet views when running in the Flex client. HTML-based portlets are displayed inside a separate popup dialog by default as described in `docs/Javascript-API.html`.

The HTML SDK introduces a new work-around to solve that problem. Your plug-in can display a real portlet within a vSphere object summary view in both the Flex or HTML client, by following these steps:

- The HTML plug-in must contain two implementations of the portlet: one in HTML and one in Flex.
- The Flex implementation must use the regular `<extended point>`, like `vsphere.core.host.summarySectionViews` for Host portlets.
- The HTML implementation must add the suffix `.html` to that extended point, i.e. `vsphere.core.host.summarySectionViews.html` in the case of a host summary view.
- The extension id of both portlet views in `plugin.xml` must be different (this is valid for all extensions, but it's important to avoid duplicating the same extension id in this case!)

Of course there is the added complexity of providing two UI implementations of the same portlet, and making your HTML plug-in “hybrid” by including some Flex code just for that use case. But portlets are by nature small UI components so the code should remain simple, and the Java services supporting both UI implementation are the same.

See the sample `samples/vsphere-wssdk-html` for an example of plug-in with two implementations of the same portlet.

## Hybrid plug-ins

*Hybrid plug-ins* are plug-ins which include both Flex and HTML code. There are 2 use cases for hybrid plug-ins:

1. A HTML plug-in which includes a Flex portlet implementation in order to display the portlet properly in the Flex client. See [Summary portlets support](#) above.
2. An existing Flex plug-in being converted or upgraded to HTML, but where some Flex views remain.

In the first case the plug-in works completely in the HTML client and can remain hybrid for as long as you need compatibility with the Flex client.



In the second case the plug-in is in some “transition mode”: it will work *completely in the Flex client* but *not completely in the HTML client* since all Flex views will be ignored.

## How to combine HTML and Flex views in the sample plug-in

There are two ways to do that:

- Combining two UI plug-ins in the same plug-in package, one using Flex and one using HTML
- Or including both Flex and HTML views in the same plug-in project.

The first solution is easier because the project wizard lets you create separate Flex and HTML projects. You will end up with two different war files that you can reference in the same `plugin-package.xml` and include in the same `plugin-package.zip`. Note that you must use different `Web-ContextPath` values in each plug-in's `MANIFEST.MF`!

The second solution assumes that you already have a Flex plug-in project, and you also created an HTML plug-in project to get started with some HTML views. The basic steps for combining everything in the Flex project are the following:

- In the Flex project, replace `war/src/main/webapp/WEB-INF/web.xml` by the `web.xml` configuration from the HTML project.
- Edit `webapp/META-INF/MANIFEST.MF` to include the `Import-Package` list used in the HTML project (use correct package name for java services and mvc controllers)
- Add `webapp/WEB-INF/spring/bundle-context.xml` as defined in the HTML plug-in if you are using Java controllers.
- Copy CSS, images and Javascript assets directly under `webapp/assets`
- Copy the HTML and javascript code under `webapp/resources` (including `web-platform.js`)
- Edit `webapp/plugin.xml` to add your HTML view extensions
- Modify your project build script to take into account the new files.

On the java side you need to incorporate the controller classes if you were using any.

## HTML plug-ins compatibility guidelines

The plug-in generation script included with the SDK creates a HTML plug-in project compatible with both the HTML and Flex clients but it is important to review *what to do and not to do* when updating that code. Follow the guidelines below and your plug-in should remain compatible.

1. Always use relative URLs to reference your own resources or end-points in your HTML and JS code (i.e. do not add a `/vsphere-client` root path)
2. Use the variable `[myplugin_namespace].webContextPath` defined in `web-platform.js`
3. Keep the `vsphere-client` root path only in `MANIFEST.MF` and `plugin.xml`

4. Add css classes for external icons in `plugin-icons.css`. See the note on external icons below.
5. When adding your own object menu or extending another object menu (VM, Host, etc.) you need to define a custom menu with a `vsphere.core.menus.solutionMenus` extension in addition to the actions defined in `wise.actions.sets`. See the note on menu extensions below.

If you are upgrading your plug-in project from an older SDK start with [4\) Upgrading from SDK 6.0.2](#) below.

## Note on plug-in context path

Each plug-in view is a separate web app whose context path is declared in the bundle's MANIFEST.MF. See for instance for ChassisA:

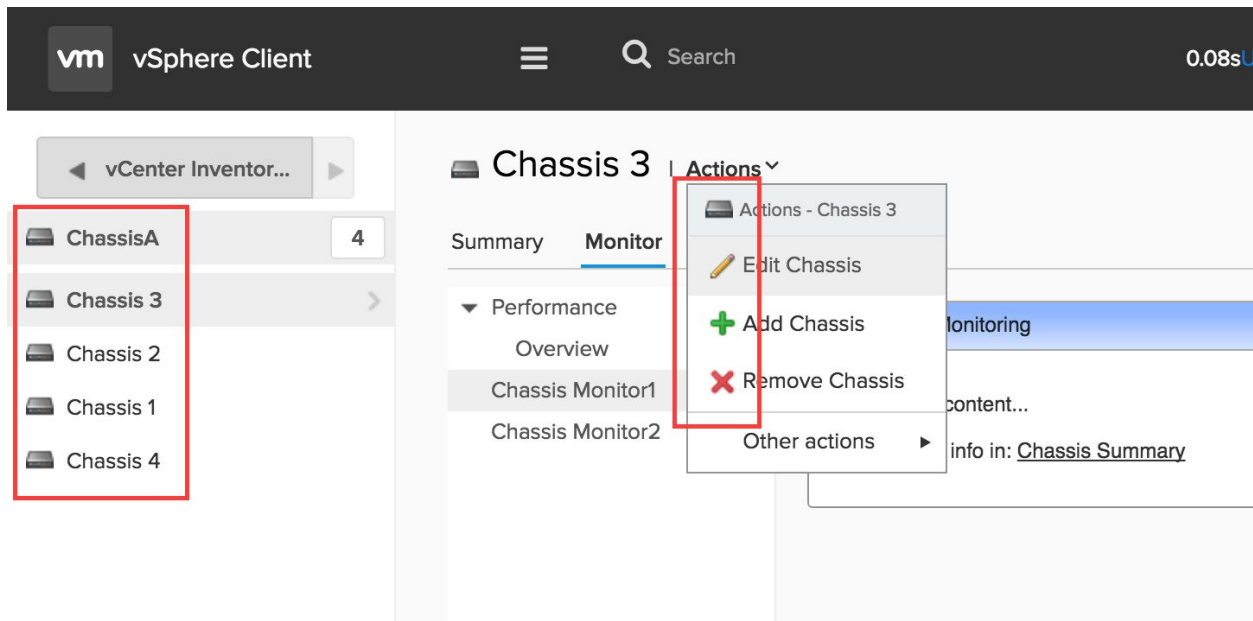
```
Web-ContextPath: vsphere-client/chassisa
```

When using HTML client you will notice that all resources and requests start with the `ui/` root path instead of `vsphere-client/` for the Flex client. The same plug-in can run in both clients even though the root path remains `vsphere-client/` in MANIFEST.MF and in URLs declared in `plugin.xml`. The transformation is done at runtime.

## Note on external icons support

*External icons* are icons displayed outside the plug-in views and handled by the HTML client itself:

- Home view shortcut icons
- Menu icons
- Objects list icons



If you generated your plug-in with the Eclipse wizard or the `create-html-plugin.xml` script you can see two menu icons defined in `plugin-icons.css` and the dependency is correctly defined in `plugin.xml`:

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

#### Additional notes:

- It's important to keep fully qualified names for resource bundles to avoid conflict with other plugins
- If the key contains "." like `<icon>#{chassis.icon}</icon>` it must be replaced by "-" in the CSS class like `.com_vmware_samples_chassisa-chassis-icon`
- You may use other bundles other than the default one, like this:

```
<icon>#{my_other_bundle:chassis}</icon>
```

In which case the CSS class name is `.my_other_bundle-chassis`

## Note on menus and sub-menus

When adding your own object menu or extending an object menu (VM, Host, etc.) it is not enough to define `vise.actions.sets` extensions for the individual menu actions. You also need to define a `vsphere.core.menus.solutionMenus` extension as shown below in the `vsphere-wssdk` sample. The `solutionMenus` extension defines the menu layout and uses references to each actions `<uid>` such as `myVmAction1` below.

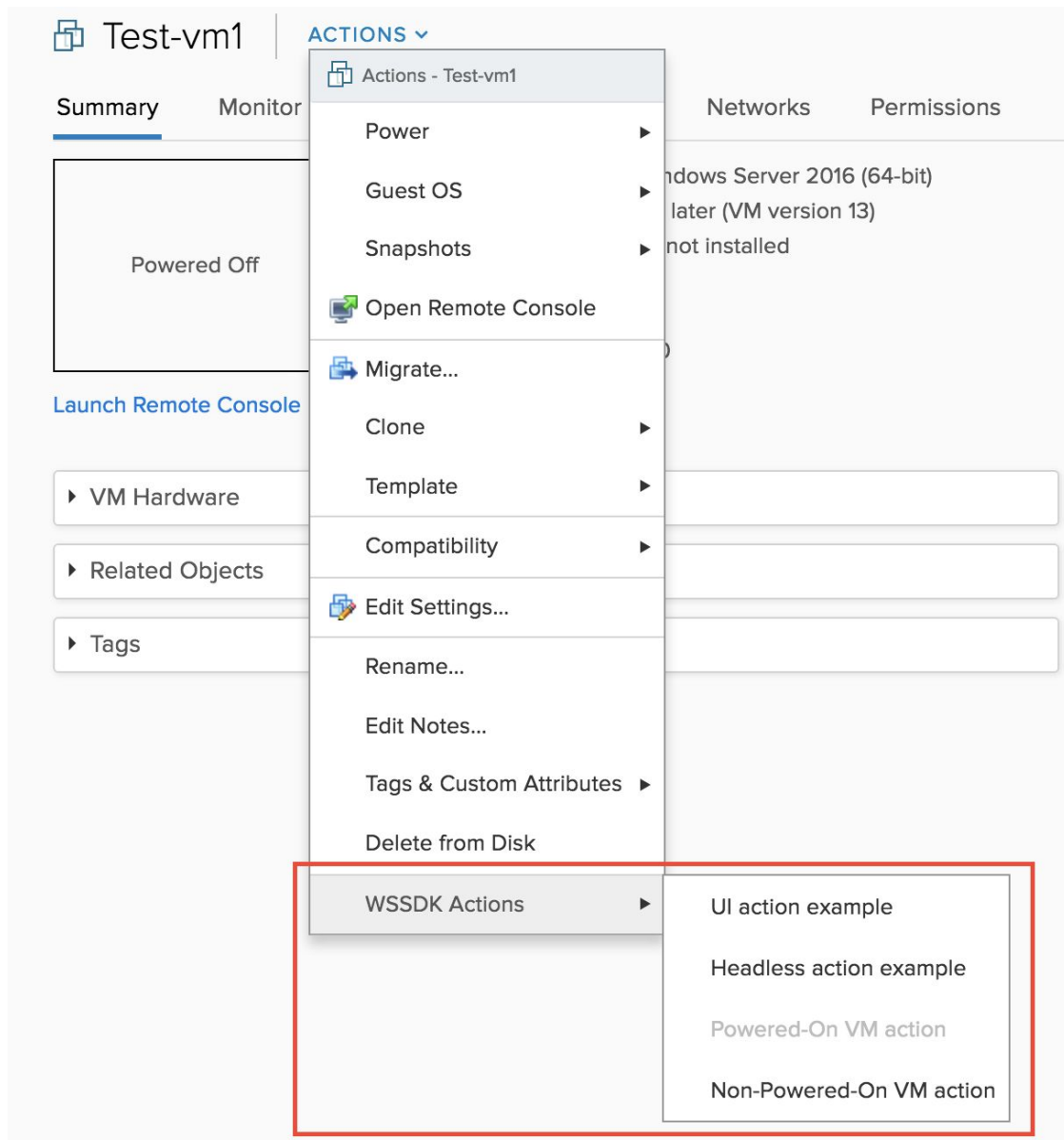
```
<extension id="com.vmware.samples.vspherewssdk.vmActionSet">
  <extendedPoint>vise.actions.sets</extendedPoint>
  <object>
    <actions>
      <com.vmware.actionsfw.ActionSpec>
        <uid>com.vmware.samples.vspherewssdk.myVmAction1</uid>
        <label>#{action1.label}</label>
        ...
      </com.vmware.actionsfw.ActionSpec>
    </actions>
  </object>
  <metadata>
    <objectType>VirtualMachine</objectType>
  </metadata>
</extension>
```

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

```

</object>
<metadata>
  <objectType>VirtualMachine</objectType>
</metadata>
</extension>

```



## Packaging and registering a plug-in

HTML plug-ins are packaged and delivered to end-users the same way Flex plug-ins were if you are already familiar with the SDK. The first step is to create a plugin-package folder as mentioned in [Plug-in packaging](#) above. Then you need to registering this plug-in package as a vCenter Server Extension.

See also [Creating and Deploying Plug-In Packages](#) in the SDK 6.0.2 Programming Guide.

### Quick test instructions

If you just want to test your plug-in package with a remote Flex client or HTML client setup without going first through the normal vCenter registration follow these steps:

- Copy your plug-in package directory to the Flex client or HTML client host, under the plugin-packages directory. For instance for Flex client with vCSA 6.0 and 6.5 it will be in  
`/usr/lib/vmware-vmware-vmware-client/plugin-packages`
- Change the owner of your plug-in directory to vsphere-client, for all files like this:  
`chmod -R vsphere-client /usr/lib/vmware-vmware-vmware-client/plugin-packages/myplugin`
- Restart the server:  
(vCSA 6.5) `service-control --start vsphere-client`  
(vCSA 6.0) `service vsphere-client restart`

### vCenter server plug-in registration tool

The SDK includes a tool to help you register your plug-in as a vCenter server extension. It is available in `html-client-sdk\tools\vCenter plugin registration`.

The **prebuilt** directory contains the main script `extension-registration` which lets you register or unregister an extension with the vCenter server of your choice. In addition the tool allows you to update the registration of an existing extension. It uses the default implementation from the local `.jar` file.

The **project** directory contains the source code and a build script to recompile `extension-registration.jar`. You can use this code to extend or customize the existing logic for your own business purpose.

Run `extension-registration.[sh,bat]` from the command line with the following parameters:

- `-action` - the action to complete: `registerPlugin`, `unregisterPlugin`, `isPluginRegistered`
- `-k <key>` - unique extension key which should match your plug-in package id
- `-url <vc url>` - the URL of vCenter server (ending with `/sdk`) where your plug-in will be registered.
- `-p <vc password>` and `-u <vc user>` - the vCenter server credentials

And also add these parameters for `registerPlugin`

- `-v <version>` - the plugin extension version, which should match the version in `plugin-package.xml`
- `-pu <plugin url>` - the URL from which the plug-in package `.zip` will be downloaded.
- `-st <thumbprint>` - the thumbprint of the server hosting the plug-in package (required when the URL is `https`)

```
./extension-registration.sh -url https://<vCenter Server IP>/sdk -username <Username> -password <Password> -action registerPlugin -key <Plug-in Key> -version <Plug-in Version> -pluginUrl https://<Host Location>/<Plug-in Package>.zip -serverThumbprint <Thumbprint Data>
```

**Note:** special characters require quoting the word or escaping the character. For instance instead of `foo!23` you need to use `'foo!23'` or `foo\!23`

#### Example:

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

For help with the full list of parameters run the script without any arguments.

Once the plug-in is registered test that the deployment works correctly from that URL by doing a new login to the HTML client. If the plug-in is not visible check for errors in the Virgo log.

#### Notes:

- Using an **http** plug-in URL instead of **https** is ok for development but not recommended for production. It is not secure and requires including the flag `allowHttp=true` in vSphere Client's `webclient.properties`.
- You can review all vCenter extensions with the MOB interface at `https://VC_IP/mob/?moid=ExtensionManager`
- To update your plug-in extension info you must first unregister it. If you try to register the same plugin twice you will get the error: `A specified parameter was not correct: extension.key`

#### Installation without a web server

The normal installation requires your plugin-package .zip file to be accessible from a web server. If this is not possible here is a work-around:

- Register your plug-in with a dummy value for the URL (it cannot be empty).
- Have your installer program copy the plugin-package folder directly in the local cache `$VMWARE_CFG_DIR/vsphere-client/vc-packages/vsphere-client-serenity/`
- That folder name must be `[extension-key]-[extension-version]`, for instance `com.vmware.samples.globalview-6.0.0.543`, and it must contain the plugin-package.xml file and the plugins subfolder with .war and .jar files.
- The dummy URL won't be used because the plugin-package files already exist in the local cache.

---

## 4) Upgrading from SDK 6.0.2

Read this section carefully if you are upgrading from the latest Web Client SDK, i.e. version 6.0.2 (it means that you used the Eclipse wizard 6.0.2 to generate your plug-in or copied a 6.0.2 sample). There are several types of changes: *URLs changes* to make the plug-in compatible with either client, *css additions* if your plug-in is using external icons, and *bug fixes* that may affect you if you had copied some sample code. See also the [API differences with SDK 6.0](#) listed in the Known issues section at the end.

The URL changes affect `web-platform.js`, `.html`, and `.js` files where `/vsphere-client` should not be hard-coded anymore. On the other hand `MANIFEST.MF` and `plugin.xml` must keep the `/vsphere-client` root path. Also, no changes to the plug-in's Java layer, i.e. services and data adapters.

When in doubt, be sure to compare your code with the samples or with a newly generated plug-in template!

Note: On the Java side the required Java *target version* was increased from 1.6 to 1.7 for Java services build scripts. This preserves the compatibility of your plug-in with vSphere Web Client 6.0. Note that java version 1.8 is required for your local Virgo runtime to work with the vCenter 6.5 release.

### Changes to web-platform.js

Each plug-in UI code contains a copy of `web-platform.js` used to bootstrap the Javascript API based on which client the plug-in runs into. A new function `WEB_PLATFORM.getRootPath` was introduced to set the correct path in both clients. So you need to add this line in the `if (!WEB_PLATFORM)` block at the top:

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

And then use this function to define the `webContextPath` as

```
myplugin_namespace.webContextPath = WEB_PLATFORM.getRootPath() + "/myplugin";
```

For instance the chassisA sample's `web-platform.js` code was changed from:

```
com_vmware_samples_chassisa.webContextPath = "/vsphere-client/chassisa";
```

to:

```
com_vmware_samples_chassisa.webContextPath = WEB_PLATFORM.getRootPath() + "/chassisa";
```

### Changes to \*.html files

All URLs starting with `/vsphere-client/` must now use relative URLs so that the root context path can be adjusted at runtime depending on where the plug-in runs (see [Note on plug-in context path](#) above). The path to each resource must be relative to the location of the html document loading that resource. For instance the chassisA sample's `chassis-manage.html` was changed from:

```
<link rel="stylesheet"
      href="/vsphere-client/chassisa/assets/css/jquery-ui-1.10.3.marge.css" />
```

```

<script src="/vsphere-client/chassisa/assets/jquery-1.10.2.min.js"></script>
<script src="/vsphere-client/chassisa/assets/jquery-ui-1.10.3.custom.min.js"></script>
<script src="/vsphere-client/chassisa/resources/js/web-platform.js"></script>
<script src="/vsphere-client/chassisa/resources/js/jquery-util.js"></script>

```

to:

```

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

```

The need to change URL format is triggered by a new deployment model which supports the two clients together. Following these instructions will enable your plug-ins to work on both vSphere Web Client and vSphere HTML5 Client.

## Changes to \*.js files

A plug-in's javascript code should use fewer URLs than HTML files and thus it should be less affected. For instance, as long as you use the helper function `buildDataUrl` to make data requests (as shown in the samples) no changes are necessary: the helper already uses the correct context path.

The easiest is to search for all instance of `/vsphere-client/` in your JS code and make the appropriate change. For example in `chassisA/chassis-summary.js`, the modal dialog code was changed from:

```

var url = "/vsphere-client/chassisa/resources/editChassisDialog.html";
WEB_PLATFORM.openModalDialog("Edit Chassis", url, 500, 300, objectId);

```

To:

```

var url = "chassisa/resources/editChassisDialog.html";
WEB_PLATFORM.openModalDialog("Edit Chassis", url, 500, 300, objectId);

```

Note that in that instance `url = "../../resources/editChassisDialog.html"` would not work!

## Changes related to “external icons”

Additional changes are required if your plug-in uses *external icons*, i.e. icons displayed outside your plug-in views and handled by the HTML client itself:

- Home view shortcut icons
- Menu icons
- Dialog title icons
- Objects list icons.

Those icons are the ones defined in `plugin.xml` with the `{key}` format like `<icon>#{chassis}</icon>`. The key value is provided in `com_vmware_samples_chassisa.properties` using the Flex syntax to embed to the `chassis.png` image asset:

```
chassis = Embed("../../webapp/assets/images/chassis.png")
```



The Flex client code uses the default resource bundle `com_vmware_samples_chassisa` declared at the top of `plugin.xml`:

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

With the HTML client you must also defined an icon class in a separate `plugin-icons.css`, and `plugin-icons.css` must be declared as a dependency in `plugin.xml` like this:

In `webapp/plugin.xml`:

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

In `webapp/assets/css/plugin-icons.css`:

```
.com_vmware_samples_chassisa-chassis {
    background: url("../images/chassis.png");
    width: 16px;
    height: 16px;
    display: inline-block;
    vertical-align: text-bottom;
    margin: 1px 4px 0;
}
```

So if your plug-in includes external icons follow these steps:

1. Create a new separate `plugin-icons.css` file.
2. Add the dependency to this file in `plugin.xml`
3. Include CSS rules only for external icons in `plugin-icons.css` (the CSS for your plug-in views should be in different files)
4. The CSS class names must be `.bundleName-iconName`
5. Set `width`, `height` and `display` attributes.
6. For objects list and menu icons use:  
`vertical-align: text-bottom;`  
`margin: 1px 4px 0;`
7. For home shortcut and dialog icons use:  
`vertical-align: top;`

See also [Note on external icons support](#) above for additional information.

## Changes related to menu extensions

If your plug-in is using `wise.actions.sets` extensions to define menu actions it must also define the menu with a `vsphere.core.menus.solutionMenus`. Note that this was not required by the Flex client, the menu was created as a “flat list” of actions. Using `vsphere.core.menus.solutionMenus` is now required to be compatible with both Flex and HTML clients. See [Note on menus and sub-menus](#) above for details.

## Testing your changes

Once you have applied those changes to your code and redeployed your plug-in locally, test it with the browser's Network console opened. Failing request(s) should be easy to detect if you missed some URL changes. External icons should be displayed correctly.

## Resolved issues in 6.0.2 samples

Please review the bug fixes below in case you used similar code samples in your plug-in:

The **globalview**'s `GlobalServiceImpl.java` source was updated to fix `getGlobalViewDataFolder` which was no longer returning the correct data directory. The name of the process running Virgo is now required. Follow that code sample if your plug-in needs to persist a small data file: on vCSA the directory will be `/storage/vsphere-client/your-plugin` for the Flex client and `/storage/vsphere-ui/your-plugin` for the HTML client.

The **globalview**'s `ServiceController.java` source was updated to add the proper `@RequestParam` in 2 methods:

```
public String echo(@RequestParam(value = "message", required = true) String message)
public void setSettings(@RequestParam(value = "json", required = true) String json)
```

The **chassisA** and **chassisB** samples were updated in `editChassisAction.html` to make the *Cancel* button work correctly. *Cancel* was working in the Flex client but it would submit the form in the HTML client instead of cancelling it. The fix is to give *Cancel* the type "button" explicitly as shown below:

```
<button type="submit" id="submit"
        class="pure-button pure-button-primary"></button>
<button type="button" id="cancel" class="pure-button pure-button-secondary"
        onclick="WEB_PLATFORM.closeDialog()">Cancel</button>
```

---

## 5) Upgrading from SDK 5.5.x, 6.0, 6.0.1

If you are upgrading directly from an older SDK this section covers the additional steps required to ensure compatibility of your plug-in with the HTML client. (For reference this is the same material as the SDK online release notes: [Web Client SDK 6.0.2](#), [Web Client SDK 6.0.1](#), [Web Client SDK 6.0](#))

### Upgrading from SDK 6.0.1

Three changes are required:

1. Names of properties files must be suffixed with `_en_US` instead of `-en_US`
2. Property `viewResolvers` should be removed from `bundle-context.xml`
3. The initialization code `web-platform.js` should be updated

First the names of properties files in `webapp/locales` must be suffixed with `_en_US` instead of `-en_US`, `_fr_FR` instead of `-fr_FR`, etc. For instance see the chassisA-HTML sample `src/main/webapp/locales/com_vmware_samples_chassis_a_en_US.properties`

The value of the `REGEXP2` property in `build-war.xml` was changed from `\2-\1.properties` to `\2_\1.properties`. You should update it in your existing build script and regenerate your war bundle. Keeping incorrect `.properties` file names will result in missing resources (i.e. keys are displayed instead of string values)

Regarding `viewResolvers`, just remove them from your plug-in's `bundle-context.xml` in case they are still there to avoid potential conflicts.

Open your plug-in's `web-platform.js` file and make sure it starts like this:

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

Remove the test `if (!WEB_PLATFORM)` further below where APIs are defined. You can compare to an existing sample's `web-platform.js` to make sure you have the correct code.

## Upgrading from SDK 6.0

Follow all the steps above (SDK 6.0.1) in addition to the following three changes:

1. Error handling code updated in java controller classes.
2. `WEB_PLATFORM.getUserSession()` no longer contains a `samlTokenXml`
3. The API `onGlobalRefreshRequest` is deprecated and replaced with `setGlobalRefreshHandler`. See the chassisA sample's `chassis-summary.js`

Regarding error handling in java controller classes the code using `@ExceptionHandler` was updated in order to return the correct error map in case of exceptions. See the example of `ServicesController.java` in `samples/globalview-html-service` and you should use a similar pattern.

## Upgrading from SDK 5.5.x

Follow all the steps above (SDK 6.0 and 6.0.1) and note the following additions since 5.5.1

- New JavaScript APIs `setDialogSize()` and `setDialogTitle()` to modify size and title at runtime.
- New XML attributes `<dialogIcon>` and `<showCloseButton>` for dialogs and summary portlets
- New JavaScript API `openModalDialog()` to open a modal dialog outside the current view.

---

## 6) Known issues

This section contains all known issues HTML Client SDK issues for the current release. Please check <docs/FAQ.html> for answers to common questions.

If you are not targeting vSphere Client 6.5 in particular and can afford to wait for the next upgrade check out the latest version of the SDK available on the [vSphere Client Fling](#) page. We update the SDK on a regular basis between each official release. This allows you to try new features, get bug fixes and send us feedback!

### API differences with SDK 6.0 HTML Bridge

- Extension point `vmware.prioritization.action` is not supported.
- Several extension points under the Monitor and Configure tabs are being deprecated in HTML client 6.5 because the layout changed, see <docs/Extension-Points.html> for details.
- Parameter `scrollPolicy` is not supported for views and action dialogs defined in `plugin.xml`, and by `openModalDialog`. The browser adds scrollbars automatically whenever the view content is larger than its enclosing `iFrame`.
- API `openModalDialog` doesn't support the parameter `showCloseButton`, the close button is always visible in the upper right of such modal dialog for consistency with other modal dialogs in the app.
- API `setDialogSize` is not supported (it is only useful for changing the dialog size once it is opened)

### General issues

Object lists headers do not include action bars and action menus like they do in the Web Client (except for global actions such as the *Add Chassis* action in the ChassisA or ChassisB samples). Simply right-click on a list item to get the action menu for that object.

The Administration tab is not visible yet so if a plug-in (such as the Globalview sample) uses the `vsphere.core.navigator.administration` category to add menu items under the Administration tab they won't be accessible. The Administration tab will be added to the next official upgrade (post 6.5). You can already turn on this feature locally to try it out: add a file named `uiFeatures.cfg` inside `/var/lib/vmware/vsphere-client/vsphere-ui/` (for MacOS) or `C:\ProgramData\VMware\vCenterServer\cfg\vsphere-ui\` (for Windows) and add this line to the file:

```
h5uiAdminPlugin = enabled
```

API `setGlobalRefreshHandler` doesn't support multiple portlets. Until this is fixed we recommend not to use `setGlobalRefreshHandler` in portlet code.

## HTML plug-ins issues when running in the Flex client

HTML plug-ins running in the Flex client have the following problems:

### **The Flex client doesn't support summary portlets in HTML**

This is a limitation due to the fact that only one HTML view per page is supported within the Flex client container. The work-around is to combine both a Flex version and an HTML version of the portlet in the same portlet as described in [Summary portlets support](#).

### **The plug-in view content may change slightly under a Flex menu or dialog**

If a Flex menu or dialog overlaps an HTML view that view is replaced by a generated image of the HTML content. That image is not always 100% accurate so users may see some small differences, but this is temporary and doesn't affect the plug-in. This is done automatically, otherwise the iFrame would cut off the Flex component under it. The HTML view is restored correctly as soon as the menu is released or the dialog closed.

The only other alternative is to show an empty view. You can force this behavior by adding the flag `<useHtmlContentImage>false</useHtmlContentImage>` in the view definition in plugin.xml.

### **The plug-in view may blank out after changing the browser size repeatedly**

For instance if you maximize and minimize the browser window repeatedly it may be possible to reach a state where the HTML view remains empty. This is relatively hard to reproduce. The only work-around is to refresh the browser.

---

## 7) How to send feedback or report problems

Your feedback is very important to us! You can report problems or ask questions in the [vSphere Web Client SDK forum](#).