

# HTML Client Plugin Seed

The fast and easy path to great vSphere Client plugins!

Version 0.9.2

[Feedback welcome!](#)



## Content

<b>Introduction</b>	<b>3</b>
<b>Getting Started with Plugin Seed</b>	<b>4</b>
Generating your first plugin	4
Running your plugin in standalone dev mode	4
Plugin views and dev UI components	4
Deploying your plugin with vSphere Client	6
<b>Tech Stack</b>	<b>7</b>
HTML Client SDK 6.5	7
Clarity Design System	7
Angular 2+ / Typescript	7
Tools	8
<b>Plugin-seed Overview</b>	<b>8</b>

View Extensions	8
Actions - with modal dialog, wizard or headless	10
Service Calls	12
Data Retrieval	12
Customizing your Plugin	12
<b>Fast Development Cycle</b>	<b>13</b>
Dev Mode	13
Dev mode components	14
View Routing	14
Mock Data	15
Live Data Mode	17
Plugin Mode (i.e. Production)	19
Differences with Dev Mode	20
Modal Dialogs - local vs. global	22
Fast Development in Plugin Mode	24
<b>Testing</b>	<b>25</b>
Debugging	25
Unit Testing with Jasmine	25
Code Coverage	26
End-to-end testing with Protractor	26
End-to-end testing within the vSphere Client	26
<b>Internationalization</b>	<b>27</b>
<b>Production Optimizations</b>	<b>29</b>
Build Options	29
AOT and Lazy loading	30
<b>Plugin-seed in Details</b>	<b>30</b>
UI code organization and syntax	30
Source files	31
<b>Summary of the dev mode advantages</b>	<b>35</b>
<b>Developing without standalone dev mode</b>	<b>35</b>
<b>Known Issues</b>	<b>36</b>
<b>References</b>	<b>36</b>
<b>FAQ</b>	<b>37</b>
<b>Feedback Welcome!</b>	<b>38</b>

---

# Introduction

The HTML Client plugin-seed helps you kickstart vSphere Client plugins based on a modern tech stack and robust patterns. It guides you through a fast development process which will allow you to expand the initial plugin skeleton into a production version while focusing on a good architecture and the proper level of testing.

## Modern tech stack:

- It is based on HTML SDK 6.5, so the generated plugin is compatible with both HTML Client 6.5 and Flex Client 6.0 or 6.5.
- The UI is 100% based on the [Clarity Design System](#), a VMware open-source project.
- It uses the [Angular 2+ framework](#) and the [Typescript language](#) which have become industry standard for modern web applications.
- It comes with utilities and development tools that will speed up your development cycles, help you start with a good architecture and show how to integrated unit tests and end-to-end tests.

See the [Tech Stack](#) section below for more details.

## Fast development process:

The traditional way of building a plugin is to treat each view as a separate UI appearing in the corresponding extension point (an object tab, a menu action, a global view). Each UI must be implemented and tested on its own, it is only visible inside the vSphere Client. The development cycle, code -> build -> re-deploy bundle -> refresh Client -> debug, is not very efficient compared to today's state-of-the-art web development.

The new HTML plugin seed lets you treat a plugin as an *integrated single page web application*, regardless of the number of views it needs to embed into the vSphere Client. In *standalone mode* the plugin is an app made of several views that are connected together with routing code and some extra "dev UI". It can be developed entirely outside the vSphere Client, using mock data or real data, and taking advantage of live browser updates.

When the same app is deployed as a plugin its internal *plugin mode* is turned on automatically, which means that each extension point knows to display the correct view. Data is served from production services, i.e. rest calls to backend services. Any extra dev UI is hidden automatically.

By having a clear separation between dev and plugin mode it is a lot easier to take advantage of modern build systems such as Angular-CLI, and to run end-to-end tests outside the vSphere Client. See [Fast Development Cycle](#) below.

In addition to these two modes the generated plugin lets you connect easily to "live data" in dev mode (i.e. use your production services instead of mock data). It also provides a left-hand side navigator for selecting objects. More details below!

**Note:** Developing a plugin as a standalone app is neither a requirement nor an SDK feature, it is only a process to facilitate your development experience. If this hybrid mode approach looks too elaborate for your use case you can ignore it but still use [the other patterns](#) and best practices offered by this plugin seed. See [Developing without standalone dev mode](#) at the end for the standard “direct development” approach.

On another note, this document focuses only on UI and doesn't cover the Java service development aspects of a plugin. Please refer to the SDK doc and samples.

---

## Getting Started with Plugin Seed

### Generating your first plugin

Generate your first plugin with the command line tool (the script requires `ANT_HOME` to point to a local Apache Ant folder):

```
./plugin-seed/tools/generate-plugin.[bat, sh]
```

If you use the default name `myplugin` the script will create two directories, `myplugin-ui` and `myplugin-service`, containing the UI component and service component.

### Running your plugin in standalone dev mode

The UI component is the web app that you can instantaneously run in dev mode:

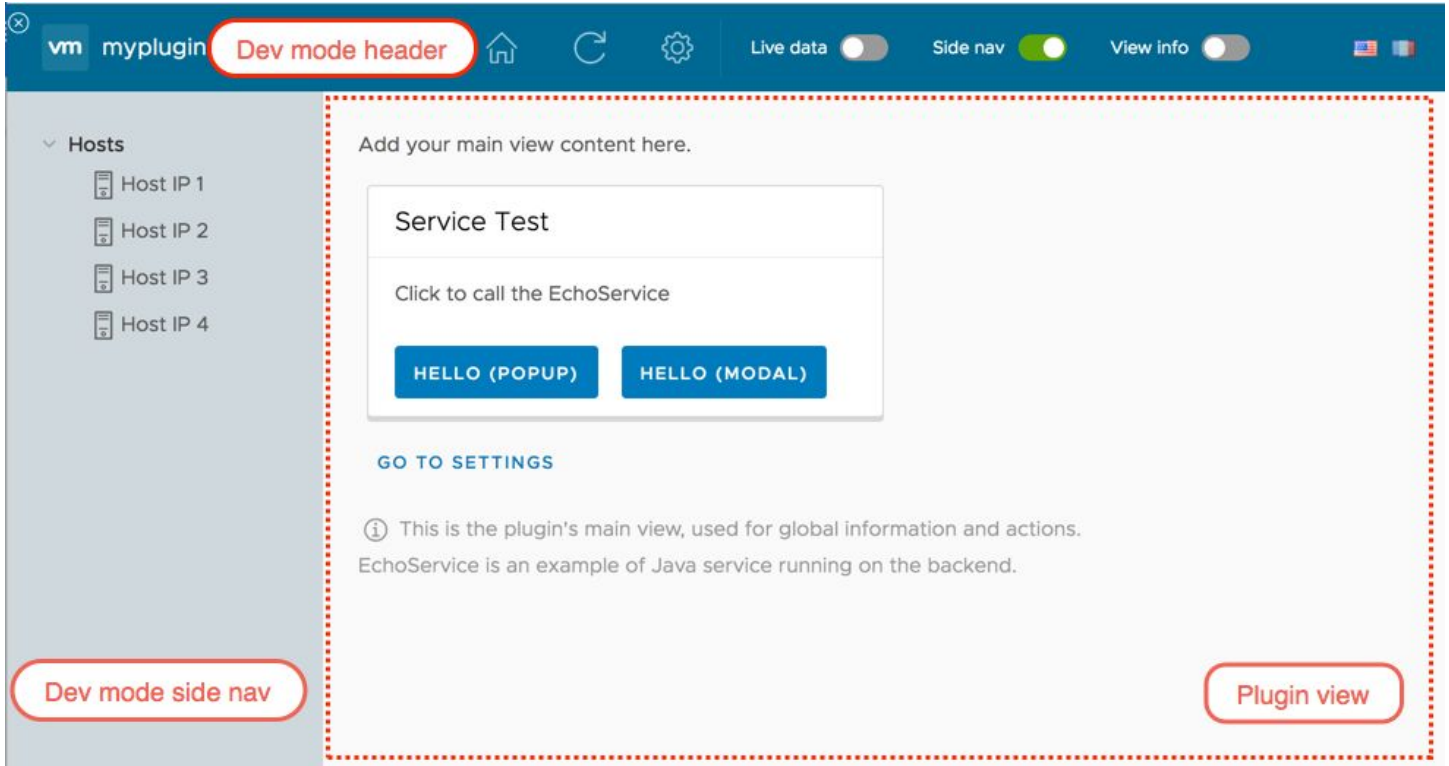
1. As prerequisite you need to install the following tools:
  - o [Node.js](#) version 6.9
  - o [Angular-CLI](#)
  - o [Json-server](#)
  - o [yarn](#) (yarn is a faster alternative to “npm install”, it is optional)
2. `cd myplugin-ui`
3. Install dependencies by typing:  
`yarn or npm install`
4. Start json-server in a separate terminal:  
`json-server --watch db.json --static ./src/webapp`
5. Start the application in standalone dev mode with:  
`npm start`
6. Point your browser to <http://localhost:4201/>

**Note:** if you are already familiar with Angular-CLI you may also use `ng serve` to start the application. This will use default port 4200 instead of 4201. This can be customized in `package.json`.

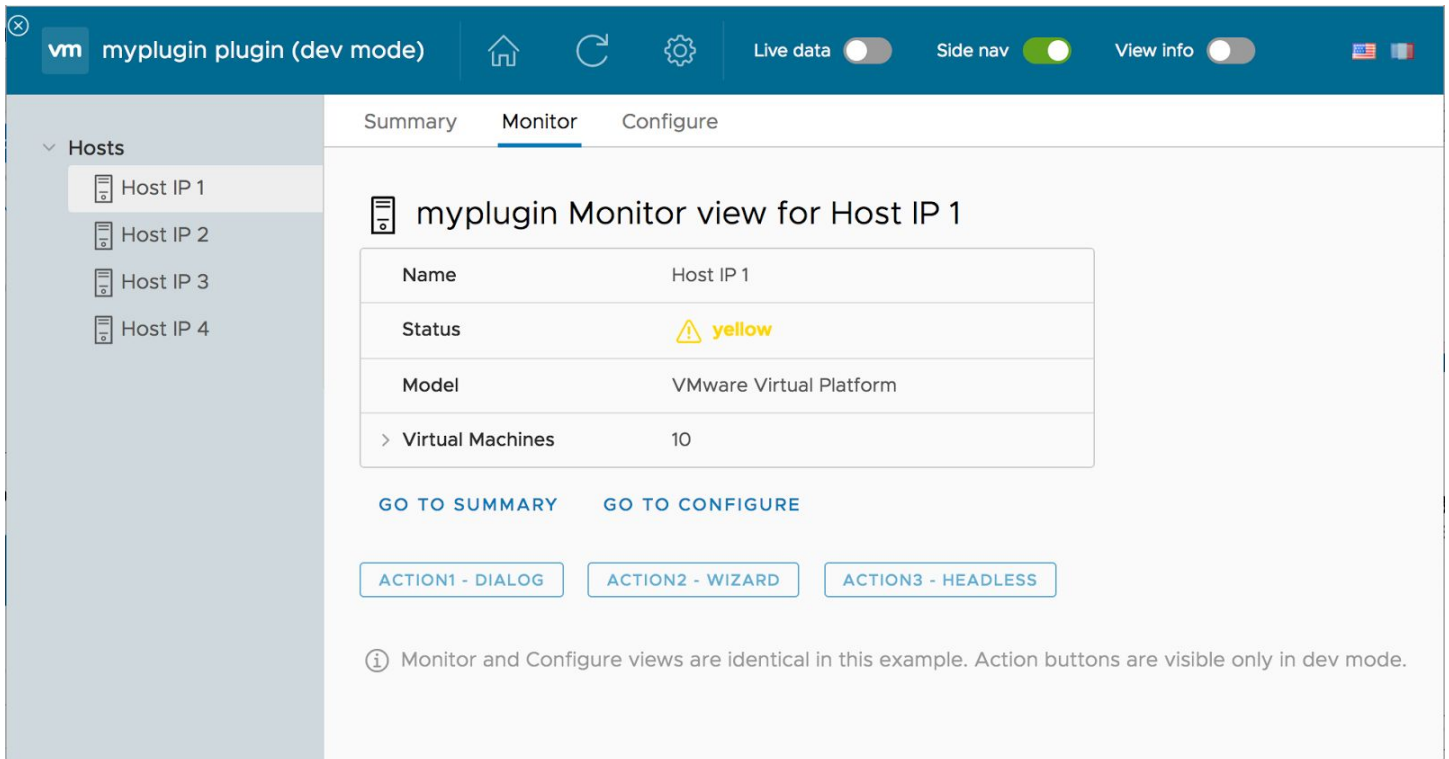
### Plugin views and dev UI components

The application home page is made of a dark blue application header, a light blue object navigator on the left and a main view in the center. Only the central main view is the plugin content, the top and left elements are

the extra *dev UI* components:



Clicking on a Host navigates to the object's Summary, Monitor and Configure views:

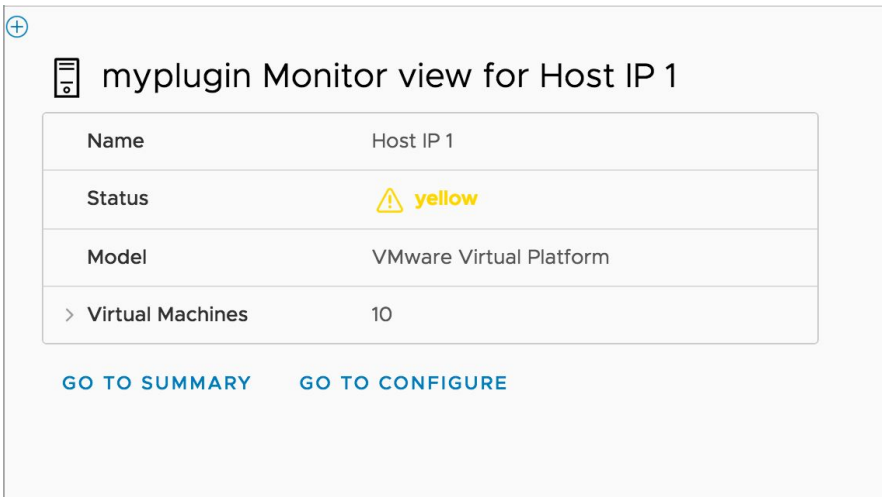


The plugin content is again the central white area, under the header, the tabs, and next to the object navigator (the “dev UI components”). This additional dev UI helps bring everything together into a standalone application for ease of development and testing.

You can hide all dev UI by clicking on the top left (X) icon:



This leaves the monitor view stripped out of the “dev UI”, as it will be displayed inside the vSphere Client:



Click again on (+) to restore the full app.

## Deploying your plugin with vSphere Client

To deploy the plugin with vSphere Client you need to build its UI and service bundles, package them and add the plugin package to your local vSphere Client setup.

Build the UI and service bundles individually with the Ant scripts in the `/tools` directory (you may need to make the `.sh` files executables on Mac OS):

```
cd myplugin-service/tools
./build-java.[bat, sh]
```

```
cd myplugin-ui/tools
./build-war.[bat, sh]
```

The `.jar` and `.war` files are generated in the respective `/target` directories.

To build everything at once, including the plugin package use this command:

```
cd myplugin-ui/tools
./build-plugin-package.[bat, sh]
```

Then copy the generated directory `target/myplugin` to your local vSphere Client setup and restart Virgo:

```
cp target/myplugin $VSPHERE_CLIENT_SDK/vsphere-ui/plugin-packages
$VSPHERE_CLIENT_SDK/vsphere-ui/server/bin/startup.[sh,bat] -debug
```

**Note:** once you have deployed the plugin package and started the server, you can use the following shortcut for updating the UI bundle in `server/pickup` without restarting the server:

```
cd myplugin-ui/tools
./deploy-war.[bat, sh]
```

Or you can use the `gulp watch` command. See [Fast Development in Plugin Mode](#).

---

## Tech Stack

Plugin-seed is based on a state-of-the-art tech stack described below.

### HTML Client SDK 6.5

SDK 6.5 is the minimum version required to build true HTML plugins that can work both in vSphere Web Client 6.0 or 6.5 (Flex) and vSphere HTML Client 6.5 and above.

You can also use the more recent Fling SDK available on the [HTML Client Fling page](#). The advantage is that you will be installing the latest version of the HTML Client for local development, while retaining compatibility with 6.0 and 6.5.

### Clarity Design System

The [Clarity Design System](#) is a VMware open-source project. It consists of UX guidelines, HTML/CSS framework, and Angular 2+ components working together to build great web applications.

Plugin-seed is based on Clarity first because it is becoming quickly a leading UI framework, and second because it will make your plugin UI de-facto consistent with vSphere HTML Client which uses Clarity too!

Clarity is evolving rapidly both inside VMware and throughout the open source community. It is already feature-rich and you can expect more to come on a regular basis.

### Angular 2+ / Typescript

[Angular 2+ framework](#) and the [Typescript language](#) have become industry standard for web applications. The other main reason for picking Angular is to be able to take full advantage of the Clarity components library (instead of being restricted to the Clarity UI styles if we were using another JS framework).

Typescript is not a requirement per se, but Angular 2+ and Typescript go very well together. Typescript is easy to learn, most of the Angular documentation uses examples in Typescript. Whether or not you already know Javascript, Typescript is the right place to start from, for cleaner code and better productivity.

**Note:** version 2+ means Angular versions 2, 4 and above, by opposition to Angular 1.x.

# Tools

The UI build system is using [Angular-CLI](#) which is also becoming a standard tool for building Angular application.

Testing is following [Angular's best practices](#) using the [Jasmine test framework](#) for unit testing and [Protractor](#) for end-to-end tests.

---

## Plugin-seed Overview

Plugin-seed creates a plugin scaffolding to kickstart a new plugin project. It demonstrates many SDK extensions points and features that you can choose to include or not. It also contains utilities that are not part of the SDK per se and provides different options that can be customized.

## View Extensions

The plugin contains the 3 types of views that can extend the vSphere Client UI:

- *Global view*, one that is not attached to a particular object context in the left navigator
- *Object view*, included into a specific object workspace (Summary, Monitor, Configure)
- *Modal view*, used in response to actions, such as wizards, one-page dialogs, etc.

The **Home** and **Settings** view are global views, they both use extension point `vise.global.views` in `plugin.xml`:

```
<extension id="com.mycompany.myplugin.mainView">
  <extendedPoint>vise.global.views</extendedPoint>
  <object>
    <name>#{app.name}</name>
    <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
      <object>
        <root>
          <url>/vsphere-client/myplugin/index.html?view=main</url>
        </root>
      </object>
    </componentClass>
  </object>
</extension>
```

The Host object views are accessed through a host context. Here is the extension definition for **Monitor**:

```
<extension id="com.mycompany.myplugin.host.monitor">
  <extendedPoint>vsphere.core.host.monitorViews</extendedPoint>
  <object>
    <name>#{monitorTab.label}</name>
    <componentClass className="com.vmware.vsphere.client.htmlbridge.HtmlView">
      <object>
        <root>
          <url>/vsphere-client/myplugin/index.html?view=monitor</url>
        </root>
      </object>
    </componentClass>
  </object>
</extension>
```



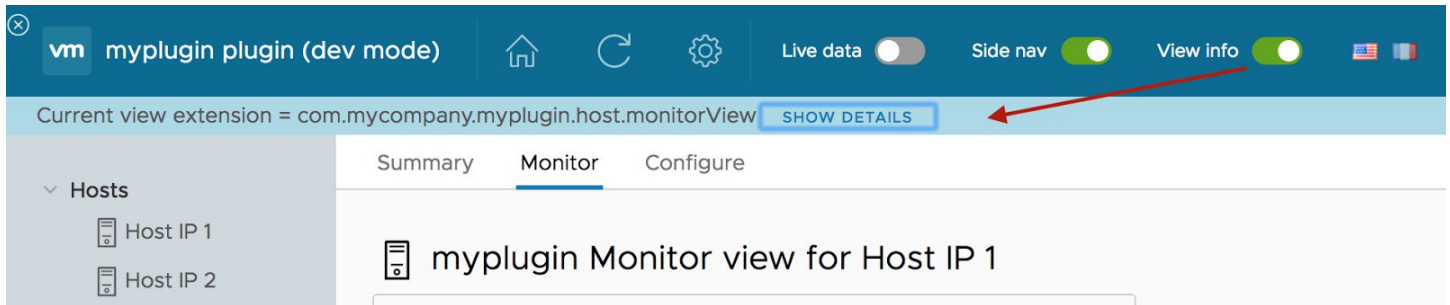
```

        </root>
      </object>
    </componentClass>
  </object>
</extension>

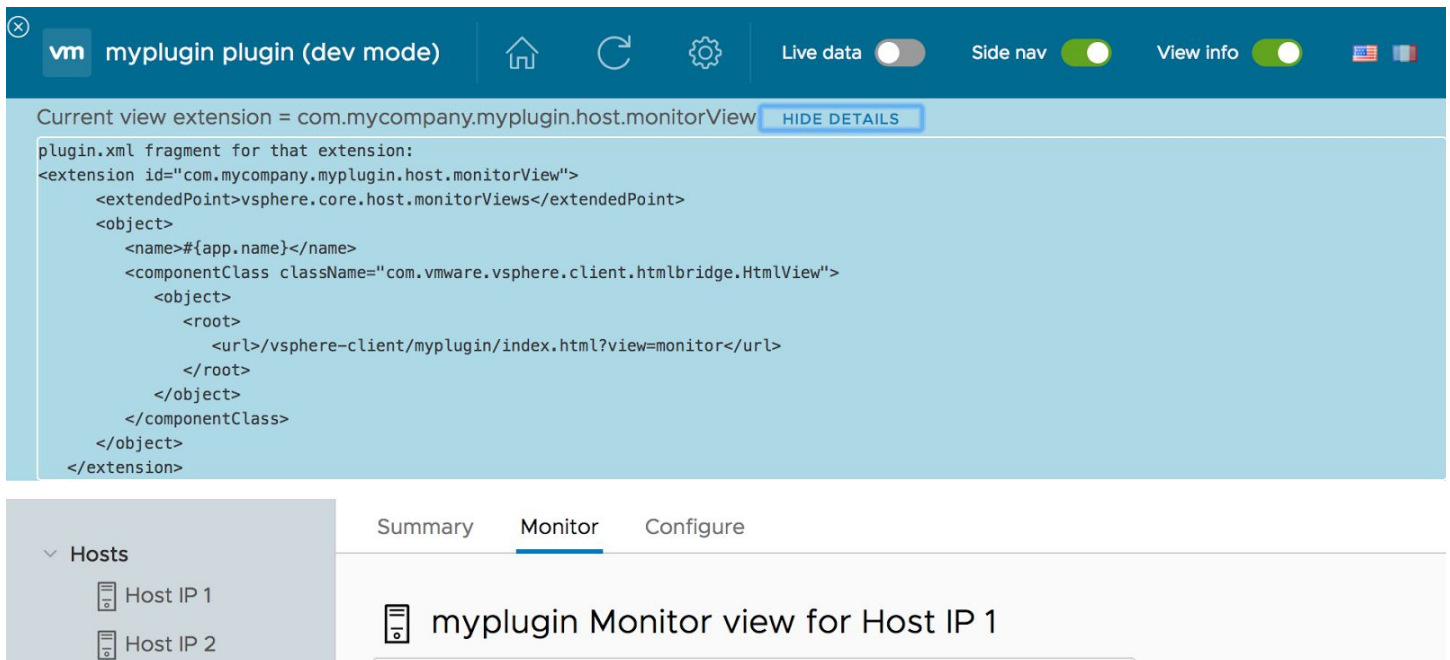
```

Navigation between the different views is handled transparently through the `app-routing` component. See [View Routing](#) below for more details.

The app header includes a *View info* switch to display additional information on the current view extension.

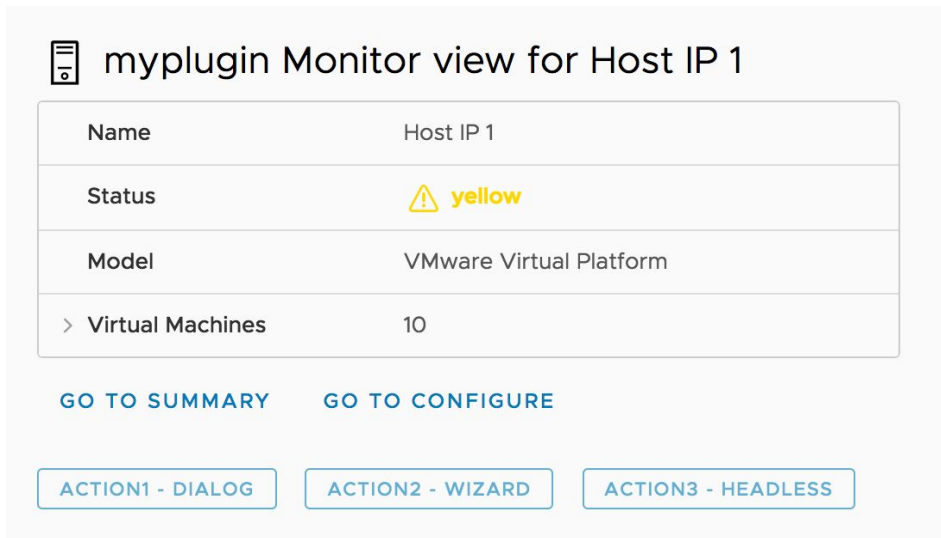



The details mode displays the xml fragment from `src/webapp/plugin.xml` for that particular extension, to be able to match the view with its extension definition.



## Actions - with modal dialog, wizard or headless

The generated plugin comes with three sample actions on Hosts, see `sampleAction1,2,3` in `plugin.xml`. In dev mode it is easier to use buttons to implement and test these actions, they will be hidden in plugin mode.

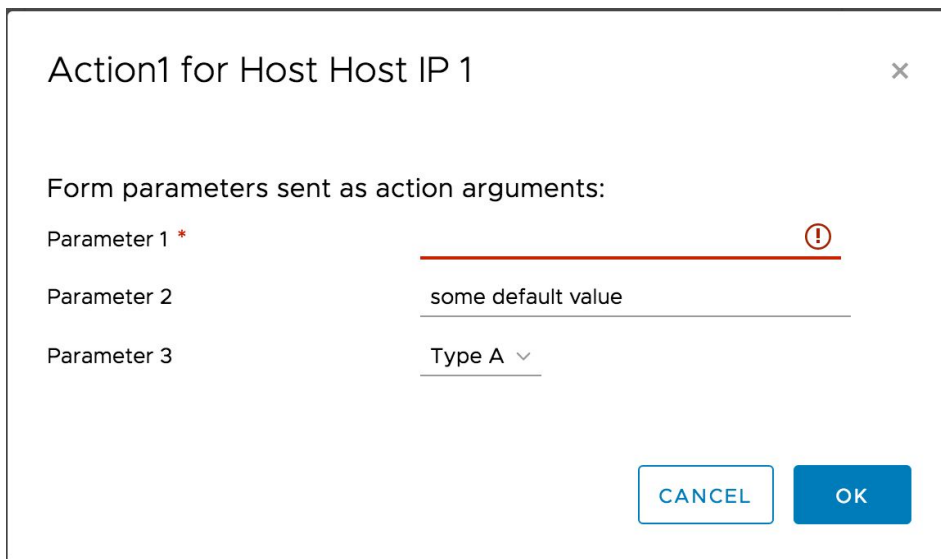


Name	Host IP 1
Status	 yellow
Model	VMware Virtual Platform
> Virtual Machines	10

[GO TO SUMMARY](#)   [GO TO CONFIGURE](#)


[ACTION1 - DIALOG](#)   [ACTION2 - WIZARD](#)   [ACTION3 - HEADLESS](#)

The *Action1 - dialog* button opens a single page modal dialog where one can fill in parameters that will be sent along. This is the general case of actions that need user input.



Action1 for Host Host IP 1

Form parameters sent as action arguments:

Parameter 1 \*  

Parameter 2

Parameter 3

[CANCEL](#) [OK](#)

The corresponding extension definition in `plugin.xml` is the following:

```
<com.vmware.actionswf.ActionSpec>
  <uid>myplugin.sampleAction1</uid>
  <label>#{sampleAction1.name}</label>
  <icon>#{addIcon}</icon>
  <delegate>
    <className>com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate</className>
    <object><root>
      <!-- execute action1 on client-side, i.e. opens a modal dialog -->
      <actionUrl>/vsphere-client/myplugin/index.html?view=action1-modal</actionUrl>
      <dialogTitle>#{sampleAction1.name}</dialogTitle>
    </root>
  </object>
</delegate>
</com.vmware.actionswf.ActionSpec>
```

```

        <dialogSize>576,260</dialogSize>
    </root></object>
</delegate>
</com.vmware.actionsfw.ActionSpec>

```

Note that `<dialogSize>576,260</dialogSize>` must adjusted to match the standard Clarity dialog sizes.

The *Action2 - wizard* button opens a multi-page wizard:

The corresponding extension definition in `plugin.xml` is the following:

```

<com.vmware.actionsfw.ActionSpec>
  <uid>myplugin.sampleAction2</uid>
  <label>#{sampleAction2.name}</label>
  <icon>#{editIcon}</icon>
  <delegate>
    <className>com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate</className>
    <object><root>
      <!-- execute action2 on client-side, i.e. opens a modal wizard -->
      <actionUrl>/vsphere-client/myplugin/index.html?view=action2-wizard</actionUrl>
      <dialogTitle>#{sampleAction2.name}</dialogTitle>
      <dialogSize>861,580</dialogSize>
    </root></object>
  </delegate>
</com.vmware.actionsfw.ActionSpec>

```

Here also `<dialogSize>861,580</dialogSize>` must adjusted to match the standard Clarity wizard sizes.

Finally the *Action3 - headless* button is the less frequent type, it calls the backend directly hence the name “headless”.

The corresponding extension definition in `plugin.xml` is the following:

```
<com.vmware.actionsfw.ActionSpec>
  <uid>myplugin.sampleAction3</uid>
  <label>#{sampleAction3.name}</label>
  <delegate>
    <className>com.vmware.vsphere.client.htmlbridge.HtmlActionDelegate</className>
    <object><root>
      <!-- action3 is headless, it calls the ActionController endpoint -->
      <actionUrl>/vsphere-client/myplugin/rest/actions.html</actionUrl>
    </root></object>
  </delegate>
</com.vmware.actionsfw.ActionSpec>
```

See the HTML SDK documentation for more information on handling action results and errors.

## Service Calls

An example of service call is done in the Main view, using the `EchoService`. You can use the same pattern to send POST requests. See the generated `EchoService.java` in `myplugin-service`.

## Data Retrieval

The hosts list in the side navigator and the host Monitor view show how to retrieve data from your Java layer through standard rest calls. Note that the code in `host.service.ts` is very generic, it handles the case of mock data or live data.

## Customizing your Plugin

Once you have generated your first plugin you can customize it in multiple ways:

- Remove patterns and utilities that you are not interested in
- Use [Angular-CLI commands](#) to generate new components and services
- Carve out code and features and insert them in another project

Note that the `plugin-package.xml` generated for the plugin contains attribute `scope="global"`. This is related to the new OSGI sandboxing feature introduced in SDK Fling 9 on Feb 20, it doesn't affect anything for prior versions of the Flex or HTML client. This attribute ensures that the plugin bundles will still be deployed on the `server/pickup` folder so that it is easy to update them during development without restarting the server.

---

# Fast Development Cycle

In addition to the features described in [Plugin-seed Overview](#) your generated plugin contains the code necessary for fast development cycles. It offers a multimode development approach:

1. A [dev mode](#), where you will spend most of your time to focus on UI.
2. A [live data mode](#), within the dev mode, which allows to connect to your production data services
3. A [plugin mode](#) (or production mode), where you run and test your app in the vSphere Client.

## Dev Mode

“Dev mode” means building and running the plugin UI as a standalone web application, without the vSphere Client in the picture. Dev mode is the default since it is the easiest.

There are many advantages of that approach:

- You can focus on the UI components and treat all views in a single page app without being encumbered with the vSphere Client environment.
- The development cycle is really fast: you build quickly, prototype quickly, test quickly, make errors and fix them quickly!
- You can use mock data easily to separate UI development and testing from backend services.
- Debugging is integrated
- End-to-end testing is also easier than doing everything within the vSphere Client.
- Having a standalone mode makes it easier to convert your plugin into another app if necessary.

Start the plugin UI as a Angular 2 application with:

```
npm start
```

You should see a webpack trace like this after a few seconds, unless there is a compilation error (note: webpack is included in Angular-CLI):

```
chunk    {0} main.bundle.js, main.bundle.map (main) 98 kB {3} [initial] [rendered]
chunk    {1} scripts.bundle.js, scripts.bundle.map (scripts) 263 kB {4} [initial]
chunk    {2} styles.bundle.js, styles.bundle.map (styles) 785 kB {4} [initial]
chunk    {3} vendor.bundle.js, vendor.bundle.map (vendor) 3.79 MB [initial]
chunk    {4} inline.bundle.js, inline.bundle.map (inline) 0 bytes [entry]
webpack: bundle is now VALID.
```

Open the browser of your choice at <http://localhost:4201/> to see the app running. If you forgot to start the *json-server* that provides [mock data](#) you will be reminded with an error message.

The `npm start` script is defined in `package.json` as:

```
ng serve --port 4201 --proxy-config proxy.conf.json
```

- You can change the default port to the value of your choice.  
(Note that the same port is used in `app/shared/dev/webPlatformStub.ts`)
- The Angular-CLI proxy configuration allows to avoid CORS issues when using the [Live Data mode](#).

`ng serve` is the standard way to generate and serve an Angular2 project via a development server. It also provides **live reload**, i.e the app automatically reloads if you change any of the source files! This is the best way to try things out quickly as you make changes.

While making changes you should also keep another terminal window open with unit tests running at all times. This way, in addition to the live reload which shows the effect of your change in the browser, you can verify that tests are passing at all times.

Start tests like this on the command line: `ng test`. See the [Testing](#) section below for more details.

## Dev mode components

Plugin-seed provides the utilities and components that makes dev mode possible: the `app-header`, `sidenav` and `subnav` components are the most visible ones as shown in the [screenshots above](#).

Each plugin view's html template includes the `app-header` component with one line at the top like this:

```
<!-- Dev mode header -->
<app-header *ngIf="gs.showDevUI()" ></app-header>
```

This header is hidden automatically in plugin mode by using API `showDevUI()` in `GlobalsService`.

`AppHeader` comes with default features and you can customize it for your own needs. Having such extra code used only in dev mode is negligible.

Search for `showDevUI()` in the `src/` directory to see all instances where

## View Routing

Since the dev mode shows an integrated app we must have code to handle the navigation between different views. This is done transparently through the `app-routing` component.

For instance clicking on the Settings icon in the header triggers this code in `app-headers.component.ts` which changes the view to Settings:

```
this.router.navigate(["/settings"]);
```

When the plugin runs in the vSphere Client the same Settings view is accessed through the Client UI, in this case by clicking in *Administration > MyPlugin > Settings* in the left navigator. This is done through the following extension in `plugin.xml`:

```
<extension id="com.mycompany.myplugin.adminSettings">
  <extendedPoint>wise.navigator.nodespecs</extendedPoint>
  <object>
    <title>#{settings}</title>
    <parentUid>com.mycompany.myplugin.adminCategory</parentUid>
    <navigationTargetUid>com.mycompany.myplugin.settingsView</navigationTargetUid>
  </object>
</extension>
```

See below the routes defined in `app-routing.module.ts`

```
export const ROUTES: Routes = [  
  // Route for plugin mode  
  { path: "index.html", component: AppRoutingModuleComponent }  
  
  // Routes for dev mode and internal routing  
  { path: "", redirectTo: "/main", pathMatch: "full" },  
  { path: "main", component: MainComponent },  
  { path: "settings", component: SettingsComponent },  
  { path: "monitor/:id", component: MonitorComponent },  
  { path: "host-action1/:id", component: HostActionComponent },  
  { path: "echo-modal/:id", component: EchoModalComponent },  
];
```

The 2nd one means that `http://localhost:4201/` without any path is redirected to the home page at <http://localhost:4201/main>. The top one means that a path starting with `index.html` will use the `AppRoutingModuleComponent` to display the correct view, this is the case for URLs used in `plugin.xml`. For instance the URL for the Settings view extension is:

```
<url>/vsphere-client/myplugin/index.html?view=settings</url>
```

In this case the `ngOnInit` function in `app-routing.component.ts` extracts the `view` parameter and uses the router to navigate to the proper component while the app is initialized in the plugin `iFrame`. The other parameters passed to the view url (`objectId`, `locale`, `actionUId`, `targets`) are processed accordingly.

## Mock Data

Separating UI development from backend data services by using mock data is a common practice that will also speed up your development process. This plugin-seed uses [json-server](#) as a *quick back-end for prototyping and mocking*. Other mocking solutions are available but `json-server` is very simple to setup and supports REST APIs.

Start the `json-server` if it's not already running:

```
json-server --watch db.json --static ./src/webapp
```

(the part `--static ./src/webapp` is not needed for mock data but necessary for the [view info](#) feature and [i18n support](#) in dev mode)

The file `db.json` contains the following mock data:

```
{  
  "echos": [  
    {  
      "id": 1  
    }  
  ],  
  "hosts": [  
    {  
      "id": 1,  
    }  
  ]  
}
```

```

    "name": "Host IP 1",
    "status": "yellow",
    "model": "VMware Virtual Platform",
    "vmCount": 10
  },
  {
    "id": 2,
    "name": "Host IP 2",
    "status": "green",
    "model": "VMware Virtual Platform",
    "vmCount": 2
  }
]
}

```

The **echos** entry allows to answer to EchoService POST request <http://localhost:3000/echos> in `echo.service.ts`.

The **hosts** entry allows to get a list of hosts, or individual host properties, with GET requests at `http://localhost:3000/hosts` and <http://localhost:3000/hosts/<objectId>> in `host.service.ts`.

Here is a code snippet from `HostService` for getting the list of hosts in dev mode:

```

private getHostsUrl(): string {
  let url: string;
  if (this.gs.useLiveData()) {
    ...
  } else {
    // Mock data => use json-server hosts API, see file db.json
    url = "http://localhost:3000/hosts";
  }
  return url;
}

getHosts(): Promise<Host[]> {
  ...

  return this.http.get(this.getHostsUrl(), headers)
    .toPromise()
    .then(response => response.json() as Host[])
    .catch(error => this.errorHandler.httpPromiseError(error));
}

```

It converts the json response into an array of Host objects. Type Host is defined in `host.model.ts`

```

// A simple Host object model
export class Host {
  id: string;
  name: string;
  status: string;
  model: string;
  vmCount: number;
}

```



```
...
}
```

## Live Data Mode

Mock data is nice to start prototyping fast but at some point you also want to test with real data coming from backend services running in your local Virgo setup. This is the case if the plugin UI runs in vSphere Client, however the “best of both worlds” would be to connect to real vSphere data while still in dev mode.

Plugin-seed provides a simple mechanism to achieve that!

The toggle button *Use Live Data* in the app header turns on and off a `GlobalsService` variable. Each service can then switch between mock data and live data at the lowest level, so that it remains transparent to callers of that service.

For instance here is how the `HostService` implements the “Live Data” mode:

```
getHostsUrl(): string {
  let url: string;
  if (this.gs.useLiveData()) {
    // Use plugin's REST endpoint to get list of object names with type HostSystem
    url = this.gs.getWebContextPath() + "/rest/data/list/"
      + "targetType=" + hostType + "&properties=name";
  } else {
    // Mock data => use json-server hosts API, see file db.json
    url = "http://localhost:3000/hosts";
  }
  return url;
}

getHosts(): Promise<Host[]> {
  let headers = this.gs.getHttpHeaders();
  let useLiveData = this.gs.useLiveData();

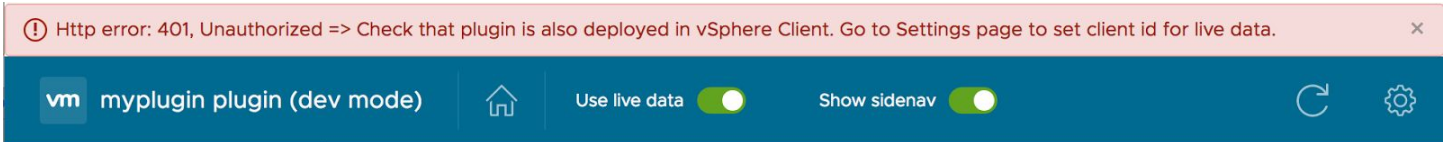
  return this.http.get(this.getHostsUrl(), headers)
    .toPromise()
    // Normal response has a data field, mock response from db.json doesn't
    .then(response => (useLiveData ? response.json().data : response.json()) as Host[])
    .catch(error => this.errorHandler.httpPromiseError(error));
}
```

In dev mode `gs.getWebContextPath()` becomes `http://localhost:4201/ui/myplugin`, and because the app was started with the Angular-CLI proxy-config option (`--proxy-config proxy.conf.json`) all requests going to the `/ui` end-point are proxied to the local Virgo runtime at <https://localhost:9443>. See the configuration in `proxy.conf.json`:

```
{
  "/ui": {
    "target": "https://localhost:9443",
    "secure": false
  }
}
```

Without such proxy configuration the standalone app at `http://localhost:4201` would not be able to send requests to `https://localhost:9443` without a [CORS error](#).

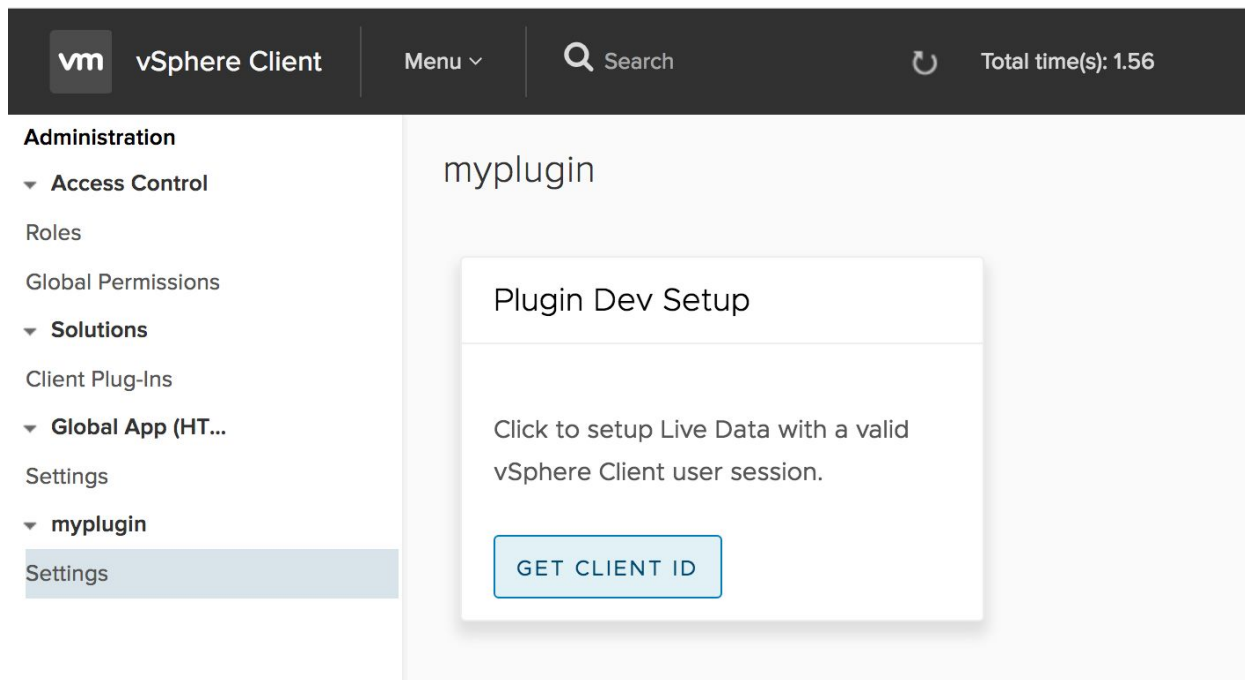
However this is not enough, as you can see when toggling the live data button... The request goes to the correct server but it is not authenticated yet:



Does every request requires authentication? It depends on your service REST endpoint. The error above occurs when the side navigator is opened because the HostService tries to get the list of hosts in your local vSphere setup. And accessing vSphere data requires authentication. But if you go to the Home page and click *Hello (Modal)* you will see the echo request return from your backend without error.

The way to make Live Data mode work with vSphere data is to inject a valid user session id in the standalone app. Unfortunately this step cannot be automated easily (but you should not have to do it more than once a day as long as you keep the session valid).

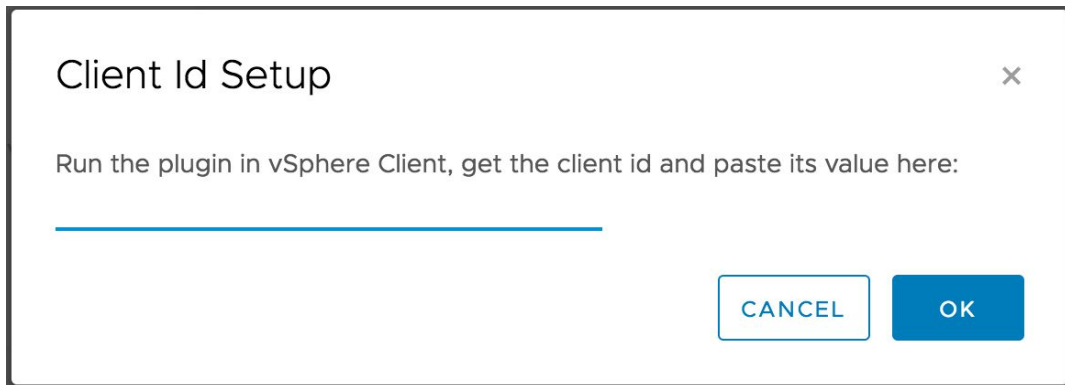
1. First you need to have your local vSphere HTML Client running (not the vSphere Flex Client)
2. You also need to deploy the plugin in vSphere Client as described in the next section [Plugin Mode](#).
3. Then go to the Settings view for that plugin and click *Get Client Id*



4. Copy the client id value from the text box



5. Go back to the browser window where the plugin is running in dev mode
6. Open the Settings view and click on Set Client Id, then paste the value.



7. Switch the Live Data button again, you should see the Hosts from your inventory

From that point on you can go back and forth, using Live Data or not. The client id is added to http requests automatically (see `getLiveDataHeaders` in `GlobalsService`) so the Virgo server treats them as authenticated requests as long as the session is valid.

**Note:** the Live Data mode value is saved in your browser cookie, so that you always go back to the same mode when the application is refreshed (see the `GlobalsService` constructor).

## Plugin Mode (i.e. Production)

Plugin mode means deploying and testing your plugin UI in vSphere Client, ready for a production release. The same application code is running but the dev mode specific logic is replaced by the normal plugin logic whenever there is a difference. The runtime switch `pluginMode` is initialized to true because each plugin view runs inside an `iFrame`. See this line in `globals.service.ts`:

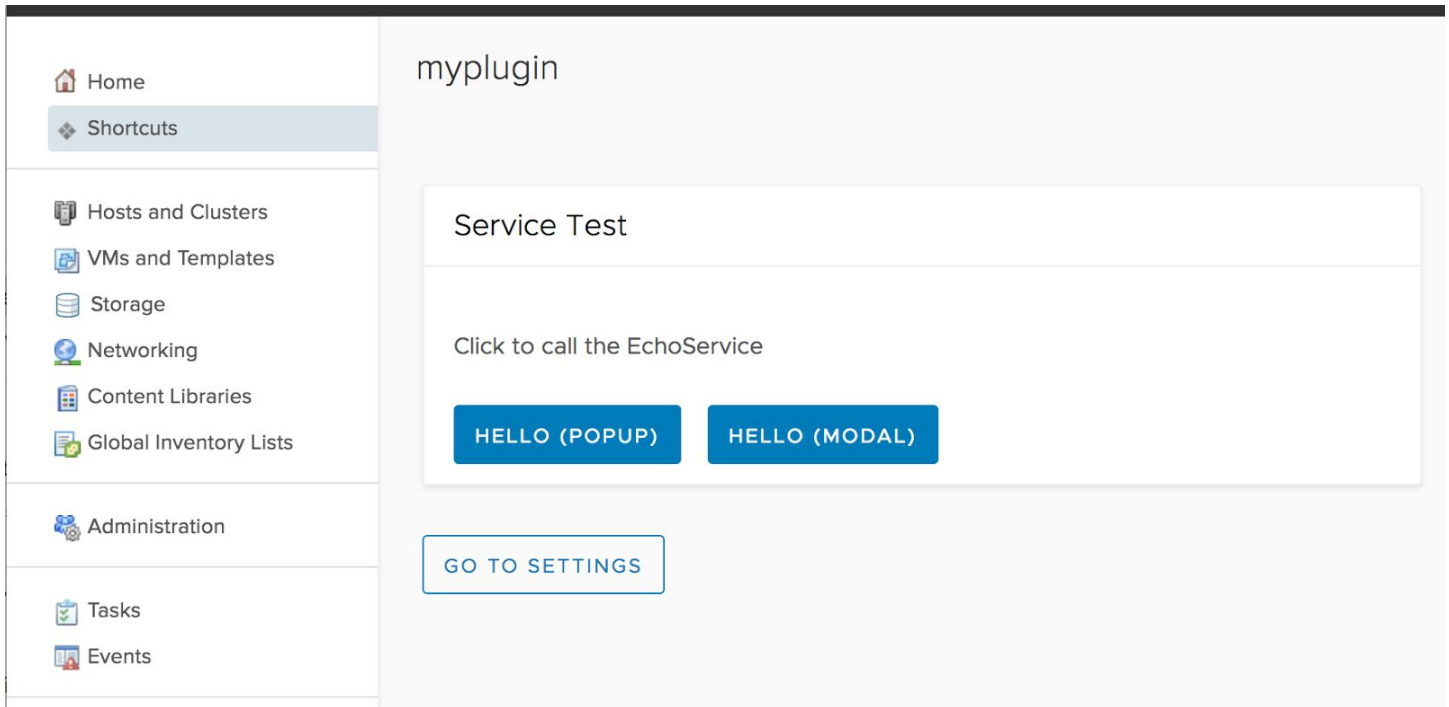
```
this.pluginMode = (window.self !== window.parent);
```

[Dev mode components](#) such as the application header are hidden automatically, so that users only see the “production UI”. The [view routing](#) handles each extension URL to display the proper view component for each extension, such as `<url>/vsphere-client/myplugin/index.html?view=main</url>`

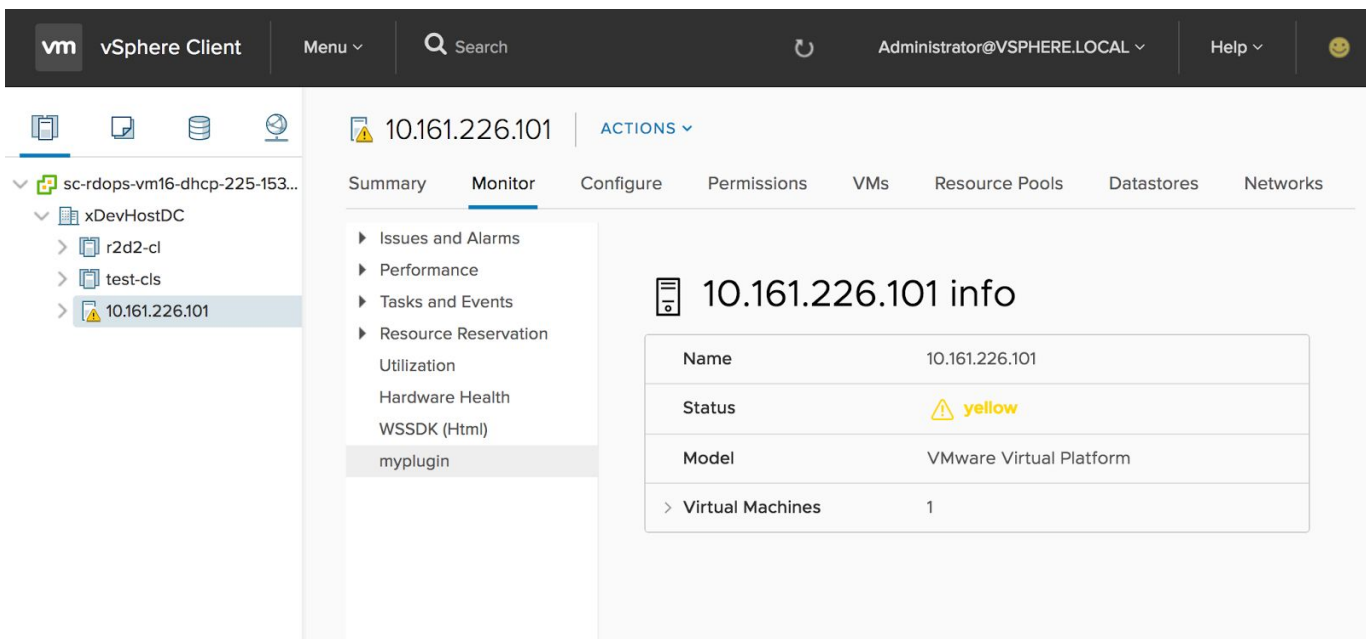
The process of building the plugin package and deploying it with your local vSphere Client setup is the same as in the SDK documentation for any plugin. Follow the steps [Deploying your plugin with vSphere Client](#) listed above.

## Differences with Dev Mode

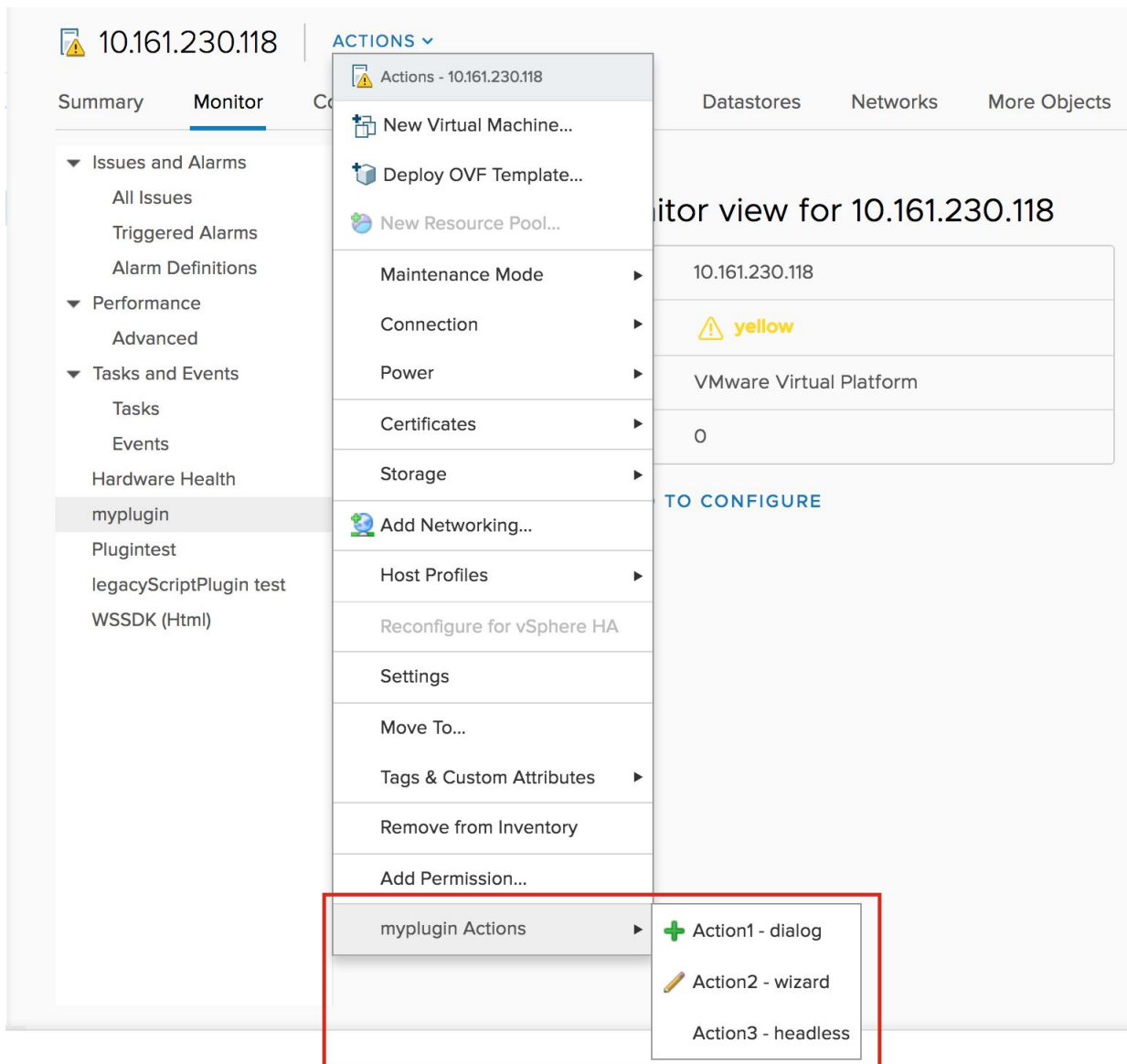
Going to the plugin's Main view, or Settings, or Host > Monitor tab, you can see that each view is a separate application, running the plugin code loaded in the view's iFrame. All extra components used in dev mode are hidden.



Select a Host in your inventory and go to *Monitor* > *myplugin* to see the plugin's monitor view.



The action buttons used in dev mode are no longer visible. The plugin's host actions are available through the normal Action menu:



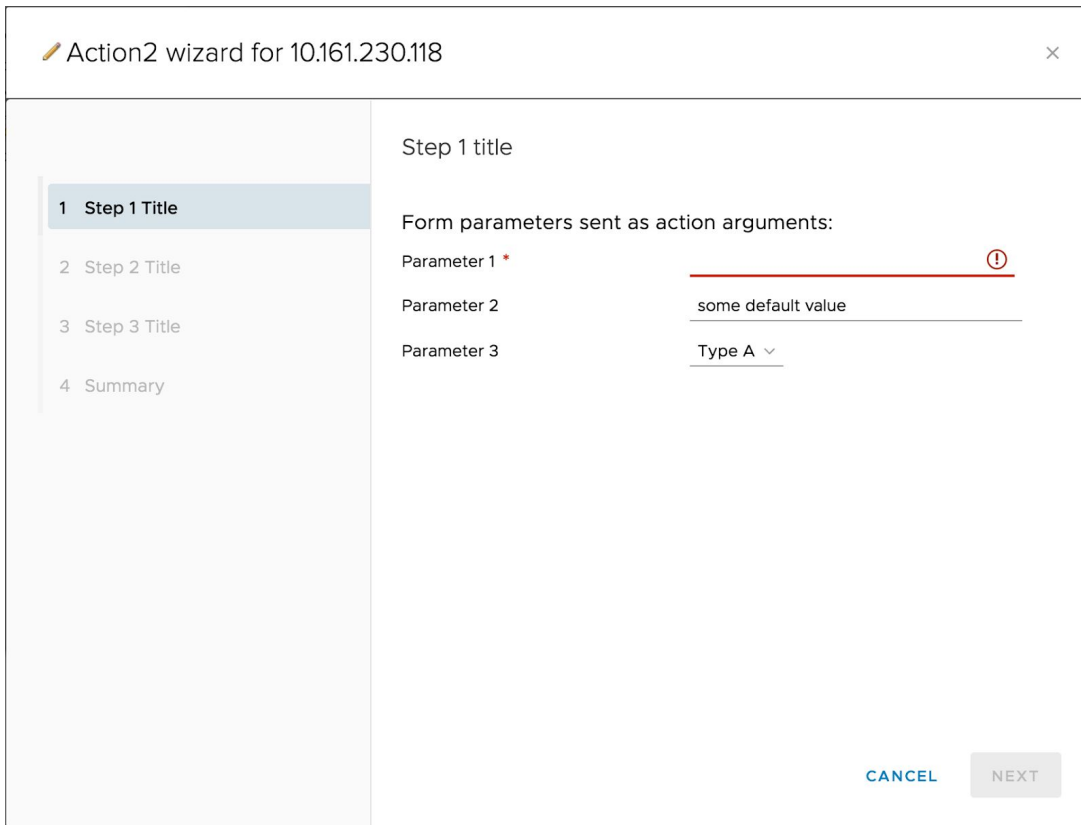
*Action1* and *Action2* open modal dialogs containing the same content as the dev mode version described above in [Actions - with modal dialog, wizard or headless](#).

**Note:** those actions are currently implemented to return a fake error containing the arguments used because it is an easy way to verify that the backend was called!

The difference with dev mode for *Action1* and *Action2* is that, in plugin mode, the modal dialog is created by the vSphere Client container, i.e. the frame and dialog title. The plugin content is limited to the iFrame under the title. For that reason `action1-component.html` doesn't contain any title, just the form representing the content. And `action2-wizard.html` has its title removed in plugin mode with this line:

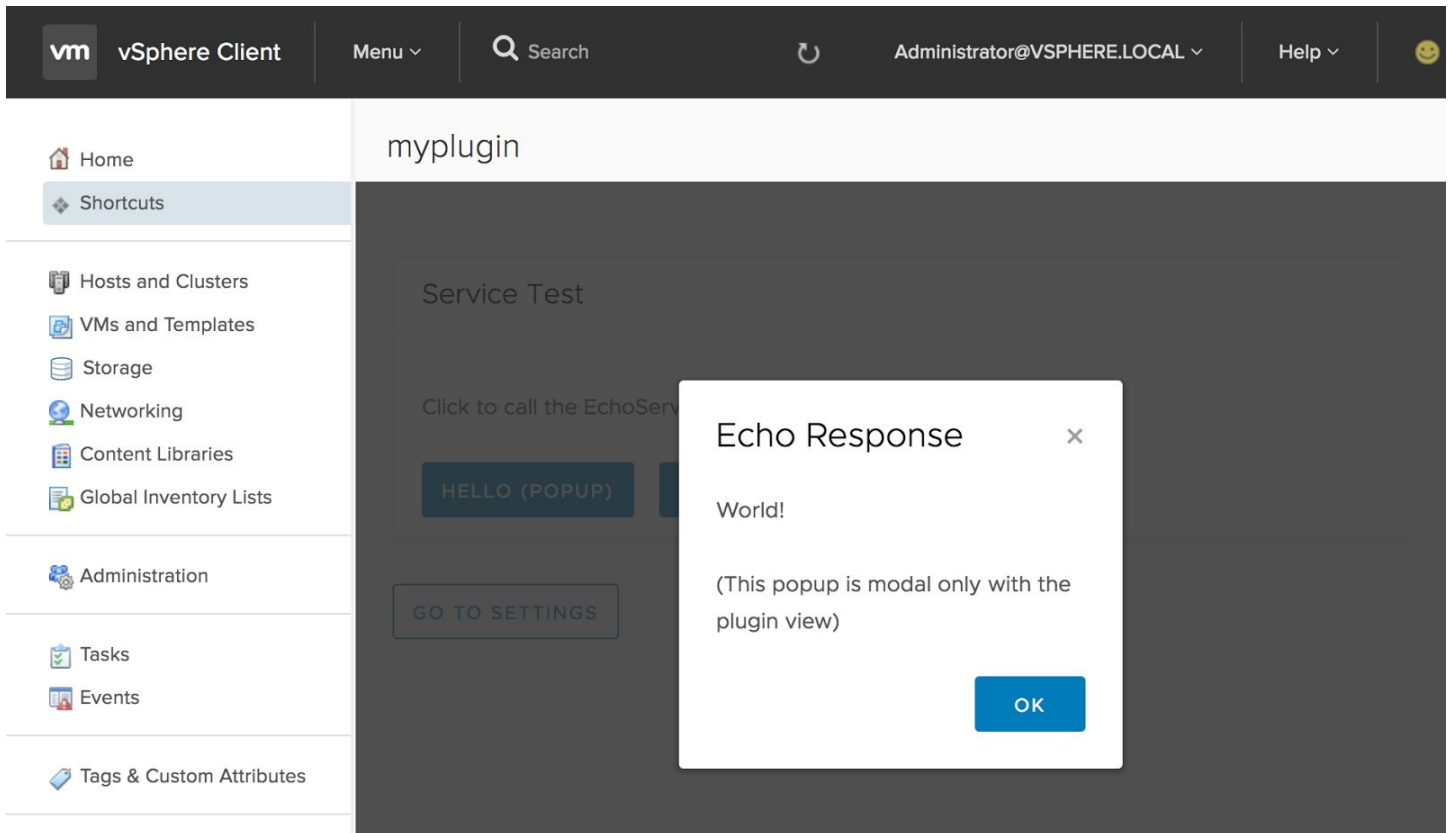
```
<div *ngIf="!gs.isPluginMode()" class="wizard-title">{{title}}</div>
```

In this case the wizard look & feel is slightly different as shown in the screenshot below. There was also some style adjustments done in `style.css`.



## Modal Dialogs - local vs. global

Button *Hello (Popup)* in the main view opens a simple popup which is modal only for that view, see the first picture below. This is because the Clarity Modal component is handled completely in page `main.component.html`, which is contained in the view's `iFrame`. Such implementation is best for simple modals such as an alert box: it is small in size and the user is expected to close the modal right away.

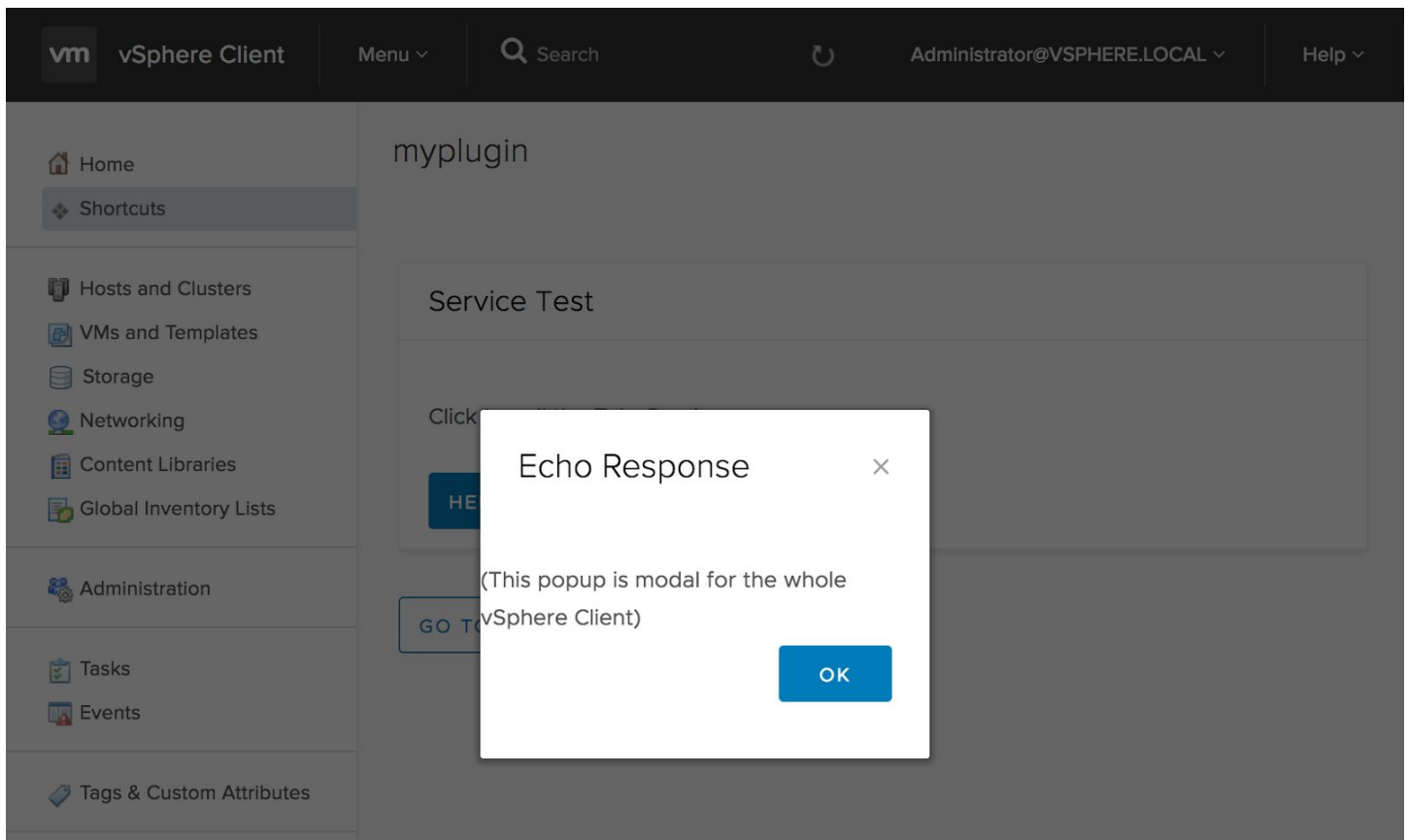


However a local popup won't work as well in two use cases:

- The dialog size may be bigger than the view itself, for instance a wizard  
=> A local popup will be clipped by the view frame.
- The dialog content may be complex and stay up for a long time  
=> The user may be tempted to click away from that plugin view, the plugin has no control over that. Either the user loses the dialog completely when going back to the same view, or you have to manage a local state to restore things as they were...

The solution is to use a real modal dialog as shown in the second Hello example below. See `main.component.ts` where we use the API `openModalDialog`. The difference is that the modal frame is created by the Client itself, the plugin only has to fill in the content (under the Dialog title). The drawback is that the dialog content takes longer to load because it is a separate web app like any other plugin view. See [Production Optimizations](#) for techniques to speed up that view.

**Note:** with a local popup use the Clarity modals option `[clrModalStaticBackdrop]="true"` in order to block mouse clicks outside the dialog because it is false by default.



## Fast Development in Plugin Mode

Although not as fast or convenient as “dev mode” it is possible to quickly refresh the code being tested in the vSphere Client. There are two approaches described below, each one requires that your first [deploy your plugin package](#) once:

1. The *automated way* is to use the `gulp watch` command in your plugin UI folder:
  - As soon as you change a source file it triggers a new build and copies the static files to the proper virgo runtime location
  - In vSphere Client, navigate to the plugin view you have changed to see the latest version.
2. The *manual way* is to decide when you want to refresh the plugin UI, then do:

```
cd tools
./deploy-war.[bat, sh]
```

  - This rebuilds the plugin `.war` file and copies it to the Virgo pickup directory at `<SDK_HOME>/vsphere-ui/server/pickup/`. You should see the redeploy trace in the Virgo console.
  - In vSphere Client, navigate to the plugin view you have changed to see the latest version.

### Notes:

- See the script `gulpfile.js`, it looks for your local server directory under `VSPHERE_SDK_HOME` as the default location but you can set a `VIRGO_BASE_DIR` variable to point to another location.



- The `ng build` command used in `gulpfile.js` still takes a few seconds. Angular CLI is based on webpack and we hope that a future version allows to re-use webpack's watch config for very fast build updates.
- Both the automated and manual ways are only valid while the server is running. If you restart the server for any reason it will re-deploy the "old" `.war` bundle from the `plugin-packages` directory.
- You should keep the plugin running in dev mode in parallel to validate changes in both environments.

## Testing

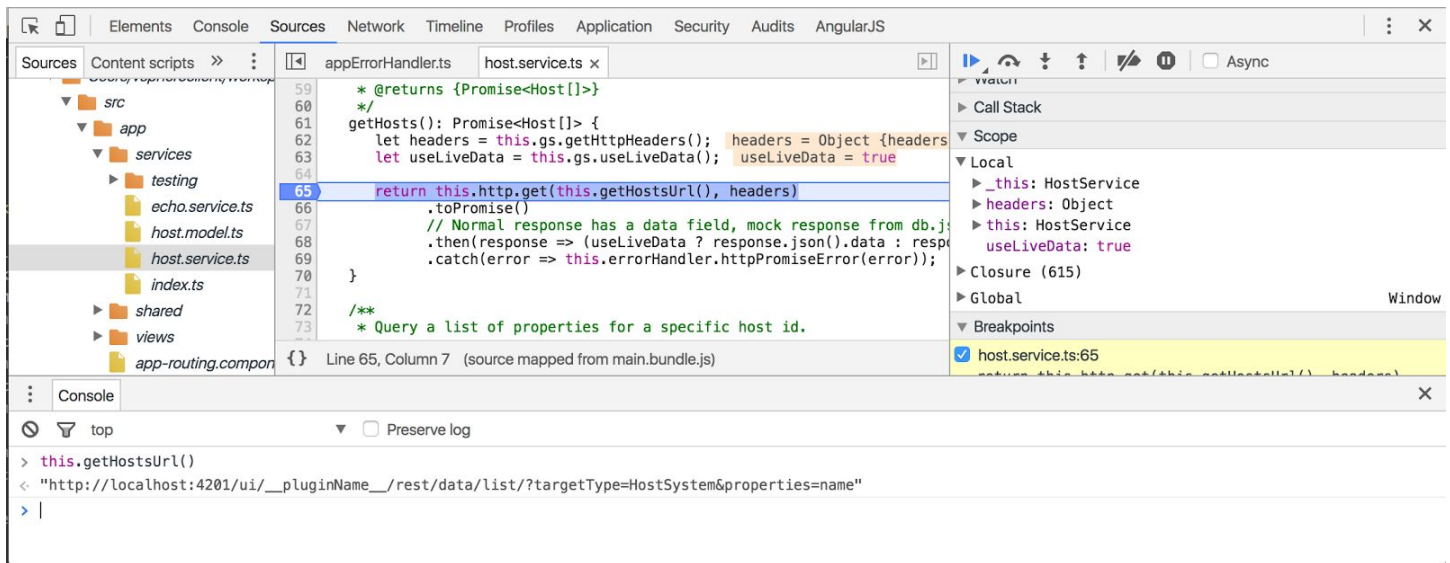
### Debugging

Debugging the Java service code requires that you start the local Virgo server in debug mode:

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

The default debug port is 8000, so you just need to configure your IDE (Eclipse, IntelliJ, etc.) for remote debugging on that port.

Debugging the UI code is done best in the browser's developer tools window. Here is a screenshot of Chrome's *Source* view with the code stopped in `HostService`. The typescript code is in clear, it's very easy to follow the code logic at any place. The *Network* view is also very useful to understand all http requests.



### Unit Testing with Jasmine

Angular 2+ provides many utilities to simplify unit tests, follow this documentation: [Techniques and practices for testing an Angular app](#).

Plugin-seed contains a fair amount of unit-tests to get you started and show off many good practices, see all the files ending with `.spec.ts`. For an example of view component unit test see `main.component.spec.ts`.

To run unit tests use the command `ng test`. In that terminal window unit tests will re-run automatically whenever the code changes until you kill the process, so it's easy to make sure you are not introducing errors!

The jasmine configuration is defined in `karma.conf.js`. Note that the browser is set to `PhantomJS` by default because they run faster with the headless browser. For debugging your test code we recommend to switch to `Chrome` because you can take advantage of the Chrome debugging tools.

## Code Coverage

Code coverage information is generated automatically when running the command:

```
ng test --code-coverage
```

Each time the tests run the coverage data is generated in the folder `coverage`. Open `index.html` and see the coverage report. It will look like this, the red showing areas with less than 50% coverage:

64.82% Statements 422/651 50% Branches 58/116 33.74% Functions 55/163 64.88% Lines 388/598

File	Statements	Branches	Functions	Lines
src/	100% 30/30	100% 0/0	100% 1/1	100% 30/30
src/app/services/	87.37% 83/95	70.83% 17/24	60% 15/25	86.52% 77/89
src/app/services/testing/	39.29% 11/28	0% 0/4	12.5% 1/8	40% 10/25
src/app/shared/	80.1% 161/201	70.69% 41/58	52.08% 25/48	80.33% 147/183
src/app/shared/dev/	47.06% 72/153	0% 0/10	15.22% 7/46	47.14% 66/140
src/app/testing/	65.63% 21/32	0% 0/4	37.5% 3/8	65.38% 17/26
src/app/views/main/	39.58% 19/48	0% 0/6	7.14% 1/14	40% 18/45
src/app/views/modals/	39.06% 25/64	0% 0/10	15.38% 2/13	38.33% 23/60

Drill down in each directory and file to see exactly what is not covered yet but unit tests.

## End-to-end testing with Protractor

(To be completed)

## End-to-end testing within the vSphere Client

(To be completed)

---

# Internationalization

For globalization purpose your plugin should support the same locales as the vSphere Client: en\_US, de\_DE, fr\_FR, ja\_JP, ko\_KR, zh\_CN and zh\_TW.

The generated plugin comes with text resource bundles in English and French under `src/webapp/locales`, see `com_mycompany_myplugin_en_US.properties` and `com_mycompany_myplugin_fr_FR.properties`

Each view component must inject `I18nService`, as an `i18n` variable for instance. The view HTML template can then use `{{i18n.translate("key-name")}}` in place of static english for all visible text. For instance in `main.component.html`:

```
<div class="content-area">
  <p style="padding-bottom: 10px">{{i18n.translate("mainView.content")}}</p>
  <div class="card-columns card-columns-2">
    <div class="card clickable">
      <div class="card-header">{{i18n.translate("mainView.serviceTest")}}</div>
      <div class="card-block">{{i18n.translate("mainView.clickToCall")}}</div>
    </div>
  </div>
  ...
</div>
```

The translation for those three messages is found in the `.properties` files:

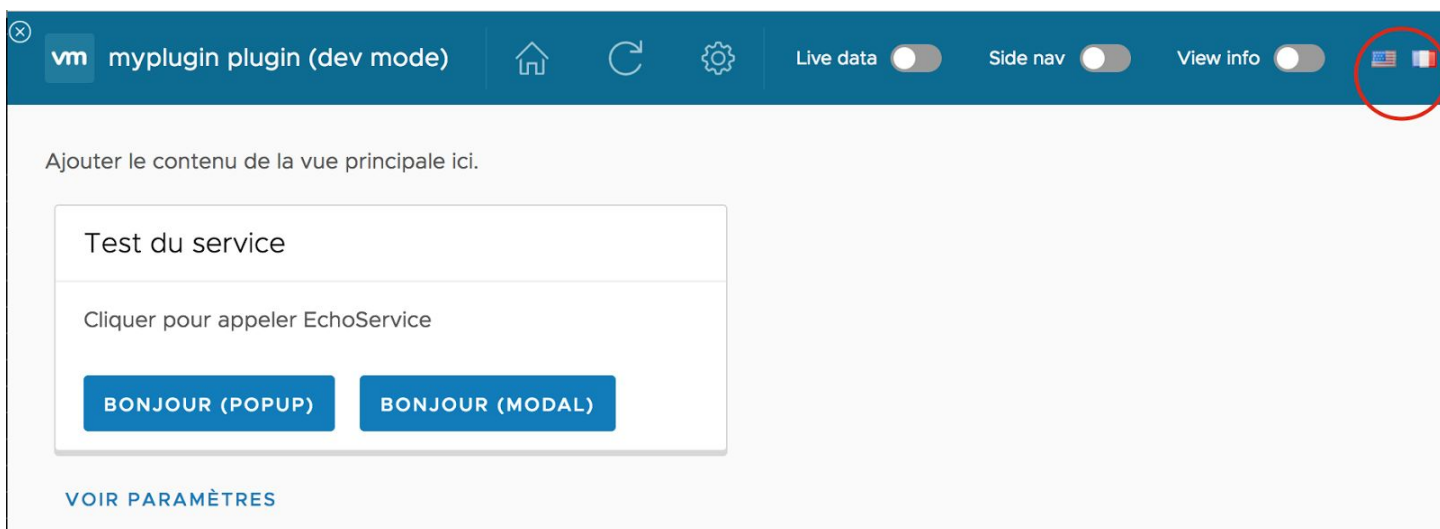
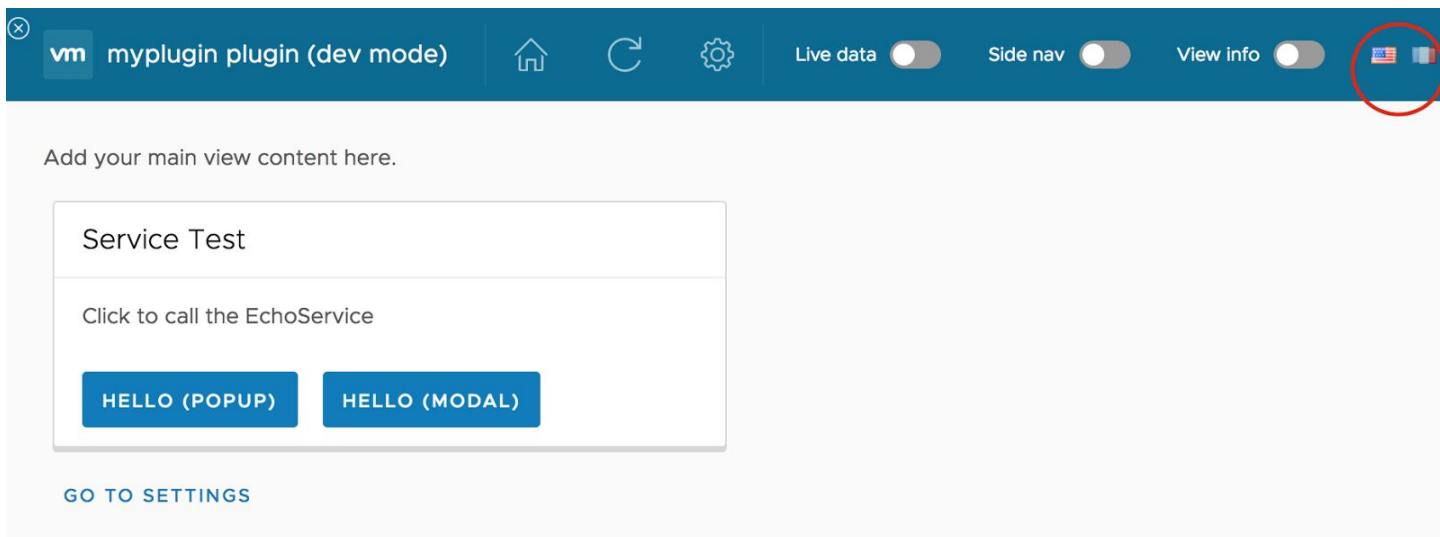
**en\_US:**

```
mainView.serviceTest = Service Test
mainView.clickToCall = Click to call the EchoService
mainView.content = Add your main view content here.
```

**fr\_FR:**

```
mainView.serviceTest = Test du service
mainView.clickToCall = Cliquer pour appeler EchoService
mainView.content = Ajouter le contenu de la vue principale ici.
```

The application header provides an easy way to test your translation in dev mode. Click on the US or French flag on the right to switch between the two languages:



## Notes:

1. There is no need to translate any text inside the “dev UI” components (header, sidenav, extra text, etc.)
2. In order for the translation to work in dev mode each `.properties` file must be converted to a `.json` file that will be loaded dynamically (see `initLocale` in `i18n.service.ts`). The initial `.json` files are provided in the generated plugin, but as you make changes you must run the script `tools/convert-properties.[sh,bat]` to update the `.json` files. Changes will be picked up right away after you switch back and forth between 2 languages.
3. In plugin mode the `.properties` files are used as-is in the HTML Client. For resource bundle to work in the Flex client they must be compiled into `.swf` files by the `build-war.[sh,bat]` script.
4. In plugin mode, any change made to `.properties` files cannot be loaded without redeploying the plugin.

The text translations that cannot be seen in the standalone app are the “external” text used in `plugin.xml`: view names, menu action labels, etc. This text is displayed directly by the vSphere Client, i.e. not by the plugin.

Angular 2+ includes its own [i18n tools](#) that we didn't have time to try out yet. In theory it should work in a plugin environment but it will be restricted to translations appearing inside a view, i.e. properties files will still be needed for the external text used in `plugin.xml`.

---

## Production Optimizations

There are several ways to optimize your application for production. We cover only the basics here, see the Angular doc [Optimize for production](#) for more ideas.

### Build Options

The build described in [Deploying your plugin with vSphere Client](#) is the “normal build” without any optimization. It uses the `ng build` command and generate several “...bundle.js” files in the `/dist` folder:

```
> ng build
Hash: 79ef83e65a656f9da059
Time: 14657ms
{0} main.bundle.js, main.bundle.js.map (main) 148 kB {3} [initial] [rendered]
{1} scripts.bundle.js, scripts.bundle.js.map (scripts) 472 kB {4} [initial] [rendered]
{2} styles.bundle.js, styles.bundle.js.map (styles) 770 kB {4} [initial] [rendered]
{3} vendor.bundle.js, vendor.bundle.js.map (vendor) 3.61 MB [initial] [rendered]
{4} inline.bundle.js, inline.bundle.js.map (inline) 0 bytes [entry] [rendered]
```

It is normal for those bundles to be large, there is no optimization and the majority of the content is the `.map` information to help debugging. See the Angular-CLI and webpack documentation to learn more about the module bundling features.

The production build is done by using `ng build --prod`

When using the Ant script in `tools/build-war.xml` you should edit the `make-bundle` target and use `ng-build-prod` instead of `ng-build`, like this

```
<target name="make-bundle" depends="clean,ng-build-prod, compile-resources">
```

The generated bundles are smaller because they are minified, uglified and have hashes in their names, although here again the size doesn't reflect what is truly loaded at runtime. Once you deploy your plugin using the `--prod` build option you can verify in the browser's Network console that the the load time is much faster because the true data being loaded is much smaller. For instance in the current version `vendor.bundle.js` shows 176K to load and `main.bundle.js` shows 50K to load.

**Note:** the `--prod` build enables the production environment in `src/environment/environment.prod.ts` which in terms triggers `enableProdMode()` in `main.ts` and Angular code runs faster because development specific checks are disabled.

## AOT and Lazy loading

Other ways to optimize your Angular 2 plugin is to use [AOT compilation](#) (Ahead-of-time) and [Lazy loaded modules](#). Besides the official Angular doc there are many blogs available online explaining the benefits of such techniques. Here is [one](#) on AOT.

The standard way to enable AOT in Angular CLI is to add the `--aot` option to `ng serve`. That option is also required at the moment to enable lazy loading.

```
ng serve --aot is not used by default in the start script of packages.json because it slows down live update. We recommend to use --aot only temporarily in dev mode, to verify that your lazy loaded modules are setup correctly (see below). On the other hand it is used by default in the production build target of build-war.xml, see <target name="ng-build-prod"> .
```

Lazy loading means that you can split your application into separate modules and defer loading the code for these modules until the router goes there. Typically each plugin view can be a separate lazy loaded module so that the app only loads the view's code plus the common code. The performance improvement become noticeable only for large amounts of code of course, but it's a good idea to start with such architecture. The `SettingsComponent` in `app/views/settings` is configured to be lazy loaded, you can repeat this pattern elsewhere if you want:

- It has its own module definition, `settings.module.ts`, which is declared in the main `app.module.ts`
- It has its own routes in `settings-routing.module.ts`
- `SettingsComponent` is not referenced from anywhere else in the app code, and in particular not declared in `app.module.ts` like other view components.
- The build generates a separate chunk of code for that module: `0.chunk.js`.
- You can see in dev mode, in the Browser network console, that `0.chunk.js` is loaded only after clicking to go to Settings.
- Similarity, In plugin mode, `0.chunk.js` is never loaded except for the Settings view extension.
- Once you apply this pattern to multiple views and with more code than in this sample the benefits can become noticeable.

---

## Plugin-seed in Details

In this section we review some good style guide practices and we explore the particularities of the most important files, grouped by folders and by names. *This review is limited to the UI code, please see the SDK documentation for more information on the service Java code.*

### UI code organization and syntax

We follow the [Angular Style Guide](#) as much as possible, it is a very good read!

In particular:

- Only one service or component per file
- Small functions
- Consistent file names + separate css and html files, like:
  - `foo.component.html`
  - `foo.component.scss`
  - `foo.component.spec.ts`
  - `foo.component.ts`
- Test files (`.spec.ts`) are kept right next to the original source and easy to find!
- Consistent coding conventions, including these [TypeScript guidelines](#)
- Intuitive folder and file names
- Code as clear as possible

Regarding [Angular modularity](#), there is still room for improvement. If your application gets bigger it's definitely advised to define multiple modules. As a starting point there is a `SharedModule` in `src/app/shared` for common UI components, a `SettingsModule` in `app/view/settings` (lazy loaded) and the main `AppModule`.

**Note:** you will notice that each directory under `src/app` contains an `index.ts` [barrel file](#) to simplify imports. This works pretty well but this is not required, you can always import a component explicitly by name. One important point with barrel files is that the order is significant if the components listed in the file have dependencies on each other, i.e. always import the lower component first.

## Source files

### `src/app/services`

`echo.service.ts` shows how to use [Observables](#) or Promises for your backend calls.

`host.model.ts` is a simple object model for the Host properties we're interested in. A good practice is to match the json data returned by the backend to make the conversion easy. In this case we must do some additional conversion for the property names.

`host.service.ts` handles the rest calls to get either the list of hosts or specific properties for a host. Note the following:

- Using an interface can help testing, see `testing/fake-host.service.ts`
- `getHostsUrl()` allows to switch easily between mock data and live data. It is better to do it at this lowest level, rather than having different APIs like `getMockHosts` and `getLiveHosts`.
- All http requests require a special header for Live Data to work, see `getHttpHeaders()`.
- The hosts response (coming from `DataAccessController.java` in the service layer) uses an extra data field. Unfortunately it is not possible to do the same in `db.json` (mock data).
- Error handling is a must for any backend call.
- `getHostProperties()` is written to accept any list of host properties, although we only used it for one set of properties (see `app-config.js`)

`nav.service.ts` is a generic service for navigating between views, regardless of the dev or plugin mode.

### `src/app/shared`



The `shared` folder could also be called `common` as it contains several utilities that are not application specific and could be reused in other apps. The `dev` subfolder is specifically about utilities that are only used in dev mode.

`app-alert.component` and `app-alert.service` allow to display simple alerts at the top of the window, with the help of [Clarity's Alert](#). In plugin mode the alert banner is at the top of the plugin view.

`AppAlertService` uses observable streams to open or close alerts. Currently only one alert can be visible at a time, so when two errors occur in separate calls the user only sees the last one. Another improvement could be to make the alert go away automatically after a timeout value. If you search the source for `closeAlert()` you can see the various events that close an existing alert.

`app-config.ts` is used to keep application level constants.

`appErrorHandler.ts` provides centralized error handling. It is important to display meaningful error messages to users but also during the development process.

`globals.services.ts` is the service which facilitates the mixed-mode development process. It is injected in many components, for instance to show and hide elements with `showDevUI()`, `isPluginMode()` or to switch the data mode at runtime with `useLiveData()`. Note the following:

- The initialization of `webPlatform` is compatible with Flex Client 6.0, 6.5 and HTML Client 6.5. (There is no need to import the file `web-platform.js` used in older style plugins)
- Also `webPlatform` has the type `WebPlatform` in order to provide type checking for all SDK calls.
- Flags are stored in localstorage in dev mode (i.e. Browser cookies) so that the app can reload in the same state.
- `getHttpHeaders` is necessary to enable the Live Data mode

`i18n.service.ts` handles text translation.

`refresh.service.ts` is a simple service used to send a "refresh event" to any observer view. It is called from the *Refresh* button on the toolbar.

`shared.modules.ts` groups the shared UI components into one module so that it can be imported by different other modules (i.e. a separate module is required for those components because they are being used in more than one other module)

`vSphereClientSdkTypes.ts` defines the `WebPlatform` TypeScript "binding"

## **src/app/shared/dev**

This folder contains only utilities used in dev mode. It is not a lot of additional code but the production build could be optimized to remove this code.

`app-header-component.ts` is the blue header used in dev mode to ties all the plugin views together. Feel free to add other tools.

`clientid.component.ts` is the modal dialog used to set or get the session client id for [Live Data](#).

`dialog-box.component.ts` is a generic OK/Cancel dialog used to wrap modal dialog sub-components in dev mode. It is used in the main view for one of the Echo service dialogs, and in the Host monitor view for the



action1 example. It relies on `dynamic-dialog.component.ts` which implements a way to inject components dynamically.

Let's take the example of `src/app/views/main/main.component.html` which contains:

```
<dialog-box *ngIf="!gs.isPluginMode()"></dialog-box
```

In `main.component.ts` that `dialogBox` element is used to open the modal dialog with:

```
this.dialogBox.openEchoModal(this.echoMsg, title);
```

and `openEchoModal` is an APIs of `dialog-box.component.ts` which uses the separate component `EchoModalComponent` to display the content of the Echo service test dialog.

There are two reasons to implement modals that way:

1. [Modal dialogs](#) that are modal for the entire app must use a component restricted to the dialog content, because in plugin mode the dialog frame and title is provided by the vSphere Client.
2. Dynamic injection allows to leverage a single `dialog-box` component for all modals that share the same footer.

All components injected in `dialog-box` must have a `handleSubmit()` method that gets called when OK is clicked.

`sidenav.component.ts` is a generic side navigation bar used to emulate the vSphere Client navigator in dev mode. It is populated by default by the list of hosts (either mock data or live data) but it can be modified to display any list of course. Note that selecting a host triggers `showObjectView()` which is the way to display the current object view for that particular host.

`subnav.component.ts` adds the standard tabs to switch between object views. It can be customized to add your own tabs.

`webPlatformStub.ts` is the stub used both in dev mode and for unit testing.

## src/app/testing

This directory contains testing stubs.

## src/app/views

`main/main.component.ts` The plugin's main view, that is opened when clicking on the shortcut icon in the vSphere Client home page.

`modal/action1-modal.component.ts` The Action1 modal component, see also `dialog-box.component.ts`

`modal/action2-wizard.component.ts` The Action2 wizard component

`modal/echo-modal.component.ts` The Echo modal component, see also `dialog-box.component.ts`

`monitor/monitor.component.ts` The Host Monitor and Configure views

`settings/` contains the plugin's Settings view which is visible in the Administration area. It is configured differently than other views because it is an example of lazy loaded module. See [AOT and Lazy loading](#) above for details.

`summary/summary.component.ts` The Host Summary view.

## **src/app/**

`app.component.ts` is the top app component. Its HTML template contains the top router outlet  
`<router-outlet></router-outlet>`

`app.module.ts` is the main app module.

`app-routing.component.ts` is the component used to redirect all `index.html` views in plugin mode.

`app-routing.module.ts` is the main routing module

## **src/assets/**

`css/plugin-icons.css` External icons, i.e. used in menus, shortcuts or object lists

`images/` Used for both external icons and other image assets

## **src/webapp/locales**

This folder contains the resource bundles for i18n support. See [Internationalization](#).

## **src/webapp/**

`META-INF/MANIFEST.MF` Plugin bundle Manifest.

`WEB_INF/` webapp configuration

`plugin.xml` Plugin extensions meta-data.

## **src/**

`index.html` will be updated automatically by the build. Note the logic to set `base href`.

`main.ts` contains the bootstrap code and calls `enableProdMode()` for production builds.

`styles.css` contains global styles that apply to the whole app. It is used to override two styles in order to adjust the wizard component (size lg) inside the plugin dialog, see screenshot in [Differences with Dev Mode](#).

---

## Summary of the dev mode advantages

Below is a list of advantages of the integrated dev mode approach. Please send us your feedback once you try it out!

- The whole plugin can be developed as an integrated single page web app outside the vSphere Client.
- It is only one app, with a runtime flag to show/hide the extra dev UI and handle view routing.
- The extra code and patterns used for dev mode are simple to use.
- It is easier to focus entirely on the UI/UX in each view, without vSphere Client in the picture.
- It is easier to build quickly, prototype quickly, test quickly, make errors and fix them quickly!
- Debugging is easier
- Its simple and natural to use mock data to abstract backend services
- It provides a live-data mode to end-to-end integration with backend services
- End-to-end UI testing is easier (than doing everything in vSphere Client)
- This provides a conversion path for an existing standalone app to become a plugin.
- Vice-versa, a plugin can be converted more easily into a standalone app for other use cases.

---

## Developing without standalone dev mode

The traditional way of developing plugins is to deploy the UI code directly to vSphere Client whether you are prototyping or testing. It is still possible to keep the development cycle pretty fast, i.e. have the UI refreshed in seconds. Here are some guidelines for using this approach or for combining the two:

1. Check out the [SDK Fling](#) sample `custom-object` that uses the SystemJS module loader. That sample doesn't include a standalone mode but it comes with a gulp script that handles the plugin packaging and provides a watch mode: after each code change the modified files are compiled and copied to the correct Virgo runtime location, so it's easy to test what was changed right away.
2. A plugin with only one or two views may be a good candidate for ignoring the extra dev mode UI (header and side nav) as not much navigation is involved. For instance if your plugin consists only of one global view you could just keep `MainComponent` from the generated code and remove all dev UI code. You can still do `npm start` to run the view by itself (for instance to use mock data), or you can package the UI bundle, deploy it once and use `gulp watch` to keep refreshing the UI in vSphere Client.
3. Another use case consists in using the dev mode approach only at the beginning to take advantage of the generated code and to learn the tech stack - Angular 2, Typescript, Clarity - without backend integration. Later on, you can include backend services and begin using the [gulp watch](#) command to refresh the UI directly for the plugin running in vSphere Client. You can go back and forth between the two mode, or just drop the standalone mode completely. (Remember that it doesn't cost much to keep the extra dev UI components code, and that they can [hide easily](#))

---

## Known Issues

Clarity components issues are tracked at <https://github.com/vmware/clarity/issues>

Angular 2+ issues are tracked at <https://github.com/angular/angular/issues>

vSphere Client SDK issues should be reported on [this VMware community forum](#).

Regarding the integration of Clarity components in HTML plugins:

- You should use `[clrModalClosable]="false"` in modal dialog to remove the X icon in the top right corner, because closing a modal with X or Escape can lead to problems.
- Wizards require css adjustments. The title must be hidden in plugin mode. Also, the wizard content will scroll within the modal dialog box unless you use an HTML Client Fling > 3.5.
- Wizard content is cached in dev mode (it is difficult to reset it), but it works in plugin mode because each time the wizard is opened a fresh app reloads. Note that Clarity will soon provide a more advanced wizard fixing this and providing better APIs.

Unit tests issues:

- Note that `main.component.spec.ts` has 2 implementations of the same tests. They both work with Chrome but the tests excluded with `xit` fails with PhantomJS because the Page elements are not initialized properly for an unknown reason.

---

## References

- Angular's [Quickstart](#) and [Tutorial](#)
- [Clarity Design System](#)
- [Typescript language](#)
- [Angular-CLI](#)
- [Json-server](#)

---

# FAQ

## What versions of vSphere Client is plugin-seed compatible with?

vSphere Web Client (Flex) 6.0 and 6.5, vSphere HTML Client 6.5.

## Do I have to maintain two apps with this approach, one for standalone and one for plugin mode?

No. You are building only one application, which can run either in dev mode during development and plugin mode once deployed with vSphere Client.

## Can I re-use only parts of the generated plugin if I don't need all the features?

Yes, feel free to remove what you don't need from the `src/` directory. You can also treat the generated plugin as a template containing many different patterns and tools, and copy what you like in another project.

## Should I use plugin-seed if I am not interested in the hybrid mode approach?

You can pick and choose the tools and patterns that can help you in your development, and remove the logic dealing with dev mode. Here is a list of patterns and best practices included in plugin-seed:

- The lazy loading optimization to minimize the code loaded in each plugin view
- The `app-alert` component for standard use notifications
- The common `AppErrorHandler`
- The `RefreshService`
- The `i18nService`
- The `NavService`
- The initialization of a global `webPlatform` object without using `web-platform.js`
- The `WebPlatform` type for TypeScript
- The injection of modal components
- The use of mock data with `json-server` (still valid in plugin mode)

See also [Developing without standalone dev mode](#) above.

## Doesn't the dev mode approach add code that should not run in production?

The dev mode code is never executed when the plugin run inside vSphere Client because the runtime flag `pluginMode` is true (this is independant of the build type and [production optimizations](#)). The size of dev UI components is fairly small so it doesn't impact your plugin's overall size.

## Can I use SystemJS or WebPack instead of Angular-CLI?

Angular-CLI includes WebPack already but if you prefer to use just [WebPack](#) directly or if you prefer [SystemJS](#) you have several choices:

1. Use the generated plugin and change the build system yourself.
2. Start from the [Clarity Seed](#) version that uses WebPack or SystemJS, and add the code back.
3. Look at the [SDK Fling](#) sample `custom-object` that uses SystemJS

## How can I use newer Clarity components?

Check the current state of [Clarity components](#) and their [release notes](#), then bump the Clarity versions in `package.json`. Let us know if anything is not compatible.

```
"clarity-angular": "^0.8.6",  
"clarity-icons": "^0.8.6",  
"clarity-ui": "^0.8.6",
```

## Can I use another UX framework in place of, or in addition to Clarity?

Sure. Plugin-seed uses only [Clarity components](#) because it is the best way to be consistent with the HTML Client, but other [Angular UI components libraries](#) are available. It should be possible to combine multiple libraries as long as there is no CSS name conflict. If you end up using other components in addition to Clarity components please send us feedback!

---

## Feedback Welcome!

Your feedback is very important to us! You can report problems or ask questions in the [vSphere Web Client SDK forum](#). Please do not use private emails unless your question contains confidential data, or if you want to reply to the survey below privately.

We would like to get your opinion on the following topics in particular:

1. Are you likely to use this plugin-seed as the starting point for your new HTML plugin? What are your main reasons for using it or not using it?
2. Are you more likely to develop your UI directly with vSphere Client? (i.e. no standalone dev mode)
3. What aspects of the plugin-seed do you prefer and which parts need improvement?
4. Are you already using (or planning to use) another stack than Angular 2 + Typescript + Clarity?
5. What other tools would you like to see from VMware to help you in your HTML plugin development?
6. What new APIs or patterns would you like to see in the next SDK?

Thanks!

The vSphere Client team.