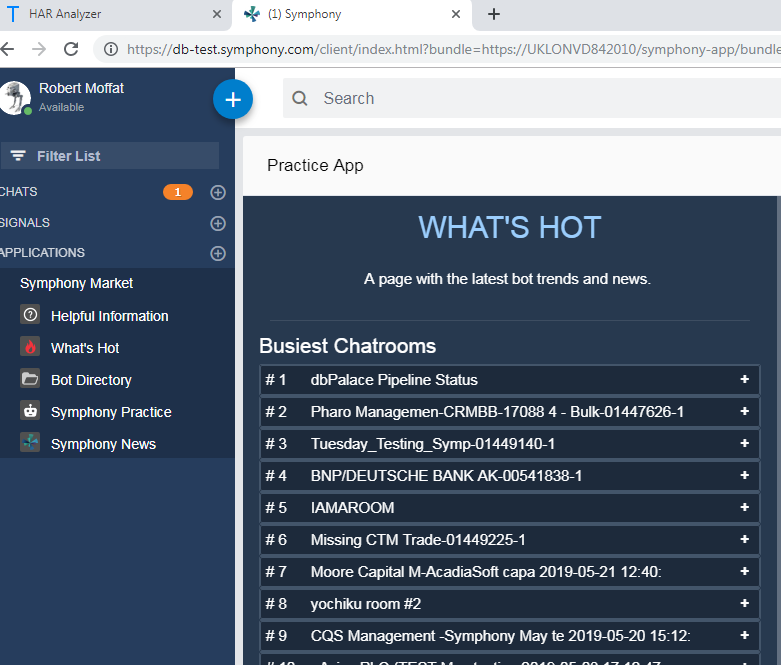
# 4. Building A Symphony App With Spring Boot

This tutorial will take you through the process of creating a new app and running it inside the Symphony container.

An app looks something like this:



That is, it lives within the Symphony client, and occupies space inside Symphony much like a chat would,

You can install apps from the **Symphony Market,** which is at the bottom of the Left-Hand Navigator.

## What Can Apps Do?

There are lots of examples of apps [here](https://workflow.symphony.com/#/).  Basically, they are useful for:

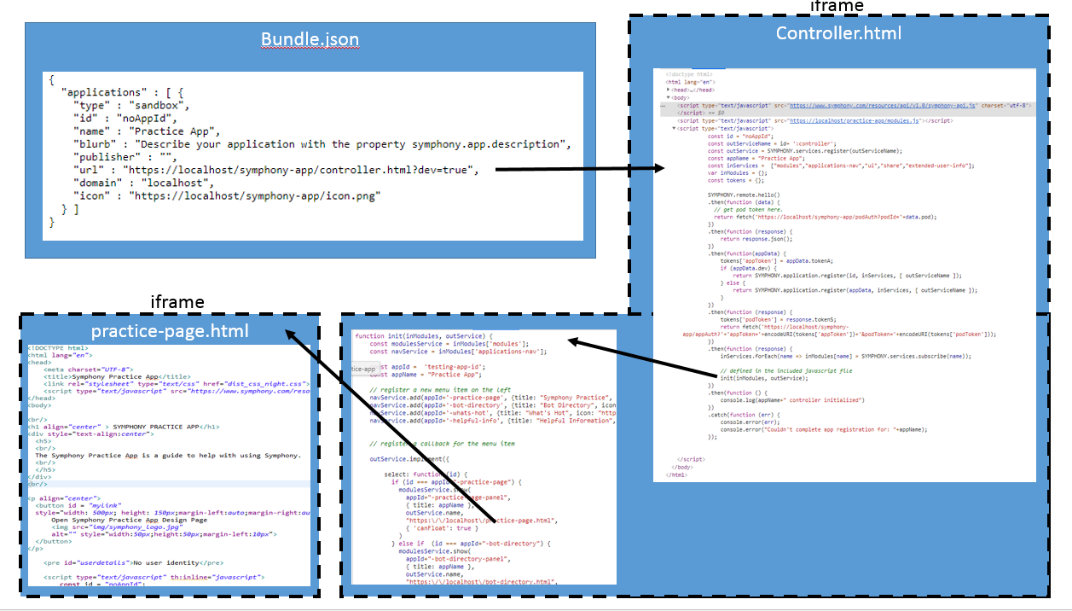
* Providing extra functionality within the client
* Adding buttons within chat windows
* Adding rendering capabilities for custom message types.
* Adding the ability to share different things within symphony
* Performing actions On-Behalf-Of a user.

## Overview

Building an app from scratch can be quite hard.  There are several basic components you need:

1. **An application server** (in this guide, we'll use Spring Boot)
2. **A Bundle Page** which describes your application to symphony, and gives a description for your app in the **Symphony market.**
3. **A Controller Page** this is loaded up by the Symphony Client, and initializes your app.
4. **Application Pages** to display to the user, like the "What's Hot" page above.

This diagram demonstrates how these different components interact:



The [spring-boot-starter-symphony-app](https://stash.gto.intranet.db.com:8082/projects/SYMPHONYP/repos/spring-boot-starter-symphony-app/browse) takes care of providing all this boilerplate for you.

## Security

Symphony takes security very seriously, and there is a complex security model to allow your application to have access to the user's identity, or perform On-Behalf-Of operations.  This involves completing a **circle-of-trust** with the Symphony back-end, and is quite a cumbersome process to code.

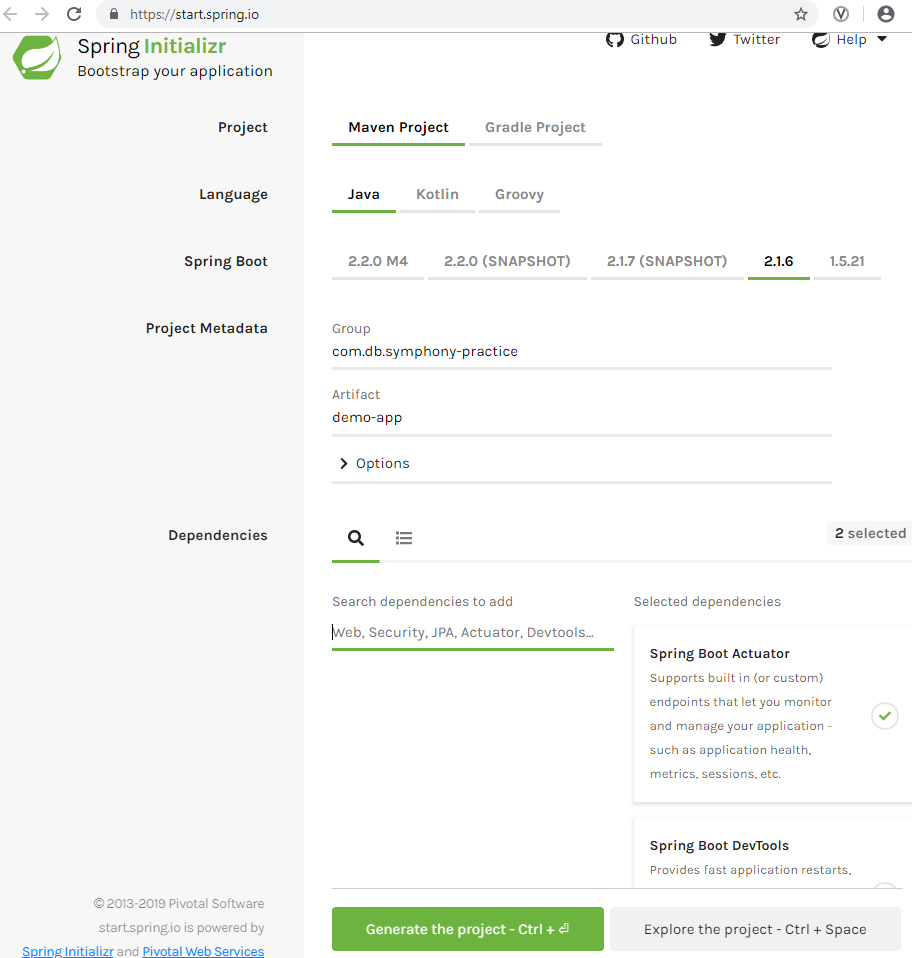
Additionally, this process only works **once your application's bundle.json file has been added to the Symphony Market on your pod.**

Here, we're going to develop the app first and then add this feature afterwards.

Let's go!

## 1. Create a New Spring Boot Project

Let's head over to <https://start.spring.io> and create a new app project:

****

* I've created a maven project, with actuator and devtools support built in.
* When you click download, it will deliver a zip file which you can unpack in your filesystem

## 2. Import Into Your IDE

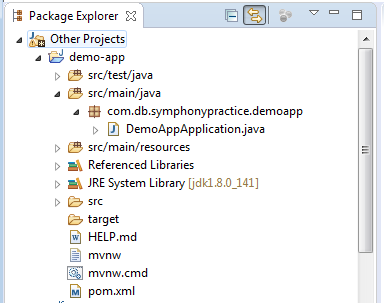
## C:\555d19af83901370e6eab515cd202359

I am going to use eclipse, which means I have to run:

mvn eclipse:eclipse -DdownloadSources

This will allow me to import my project into eclipse.  If you are using a different IDE, substitute your step here.

Here is my project in eclipse:

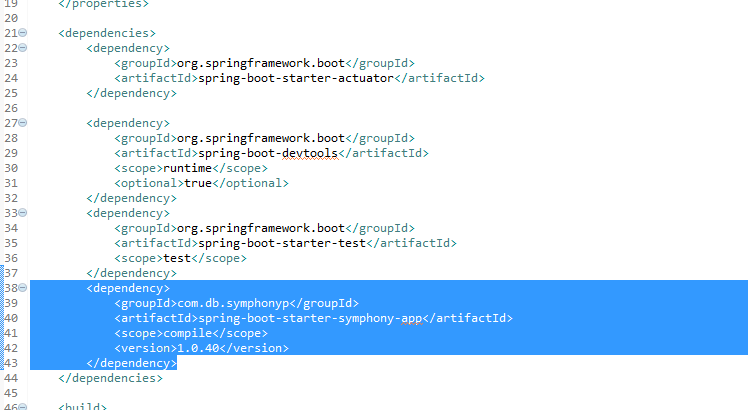


* If I run the project now, I should see Spring start, and then finish (as we haven't actually added any code).

## 4. Add The Spring-Boot-Starter-Symphony-App Dependency

In pom.xml add this dependency (please check artifactory for later versions):

<dependency>  
 <groupId>com.db.symphonyp</groupId>  
 <artifactId>spring-boot-starter-symphony-app</artifactId>  
 <scope>compile</scope>  
 <version>--latest version --</version>  
</dependency>



## 5. Add Jax-RS Dependency

JAX-RS provides the code to call REST endpoints.  You can either add Jersey, or CXF.  Both are detailed on this page under Step 2: [Choose a JAX-RS Implementation.](https://stash.gto.intranet.db.com:8082/projects/SYMPHONYP/repos/spring-boot-starter-symphony-api/browse)

#### CXF

Include this dependency:

<dependency>

<groupId>org.apache.cxf</groupId>

<artifactId>cxf-rt-rs-client</artifactId>

<version>...</version>

<scope>compile</scope>

</dependency>

#### Jersey

Include this:

<dependency>

<groupId>org.glassfish.jersey.core</groupId>

<artifactId>jersey-common</artifactId>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.core</groupId>

<artifactId>jersey-client</artifactId>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.inject</groupId>

<artifactId>jersey-hk2</artifactId>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.media</groupId>

<artifactId>jersey-media-json-jackson</artifactId>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.connectors</groupId>

<artifactId>jersey-apache-connector</artifactId>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.media</groupId>

<artifactId>jersey-media-multipart</artifactId>

</dependency>

You can use the Jersey BOM to avoid specifying version numbers like so:

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.glassfish.jersey</groupId>

<artifactId>jersey-bom</artifactId>

<version>...</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

## 6. Rebuild your Project

mvn eclipse:eclipse -DdownloadSources

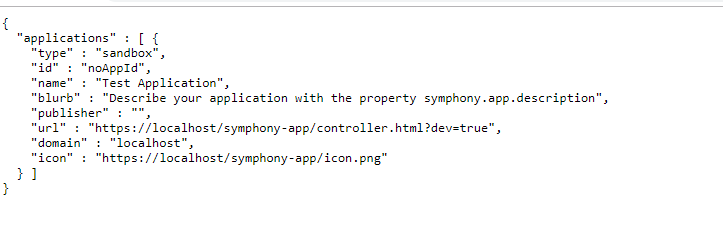
* Then, refresh in eclipse.

## 7.  Run The Application

At this point, you should be able to start the application using:

mvn spring-boot:run

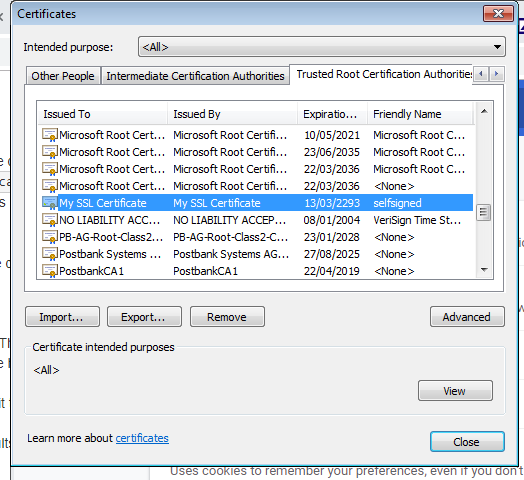
And navigate to the bundle at [https://localhost/symphony-app/bundle.json](https://localhost/symphony-app/bundle.json:)



## 8.  Trust The Certificate

By default, the app comes with a self-signed certificate for localhost.

* Save the certificate by clicking the padlock in the url bar, certificate → details → Copy To File
* You will need to add this to your trust store in Chrome.  Head to chrome://settings and add the certificate in to your "Trusted Root Certification Authorities" list



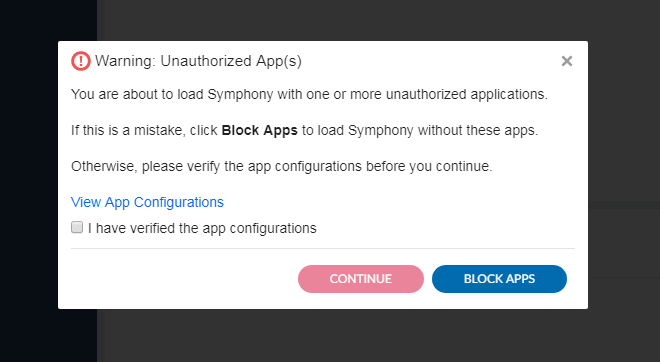
* Restart Chrome.

## 9. View Your App

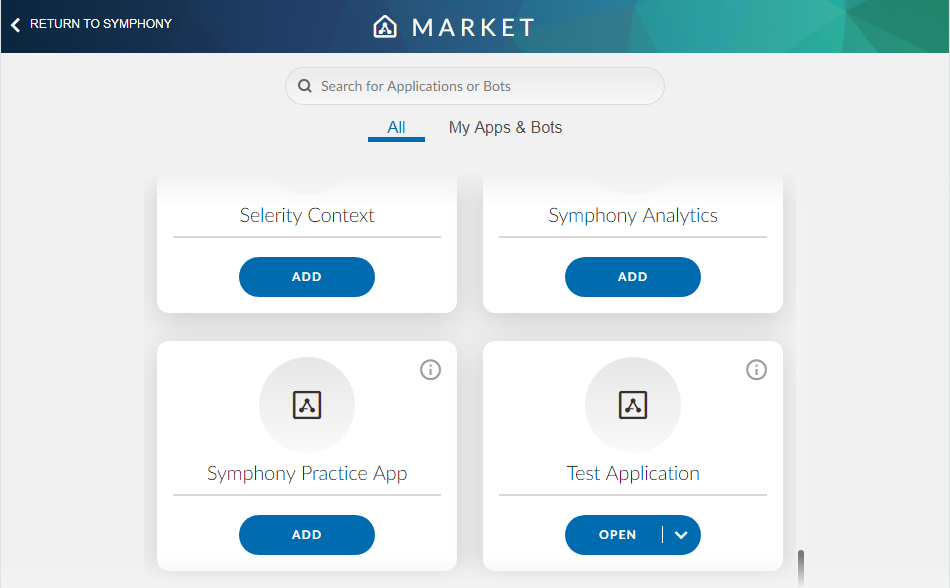
* You can load the app in Symphony now, by supplying the ?**bundle= extension** to the URL.

e.g. [https://<your pod>.symphony.com/client/index.html?bundle=https://localhost/symphony-app/bundle.json](https://db-test.symphony.com/client/index.html?bundle=https://localhost/symphony-app/bundle.json)

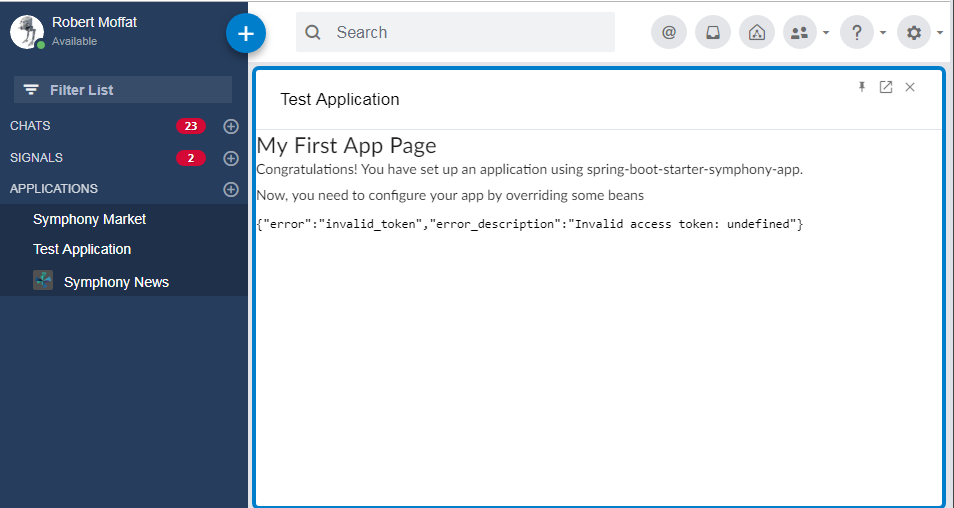
* If Symphony loads the bundle ok, you will get this message:



* You should be able to click the check-box and hit continue.
* Your app will now be available in the **Symphony Market.**  Navigate to that and install the app "Test Application"



Finally, your app is visible in Symphony on the left-navigator:



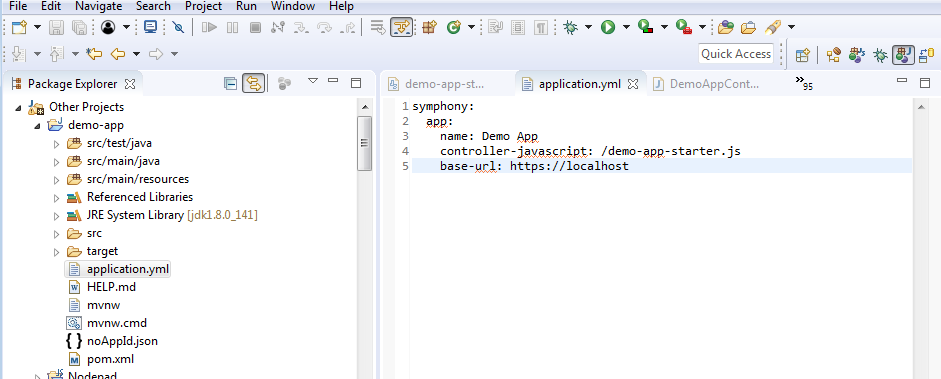
Note, because the app is not installed in the pod, there is no circle-of-trust and therefore the app doesn't have the user's identity.

Once we install in the pod, this test page will come back with the user's details (see below)

## 10. Customizing the Bundle

<https://stash.gto.intranet.db.com:8082/projects/SYMPHONYP/repos/spring-boot-starter-symphony-app/browse> provides details on how to customize the contents of the bundle file.  You can set a description for the application, it's name, permissions it is allowed to use On-Behalf-Of the user, and URL used for the application. (To determine the controller.html location, and hence where your app is installed).

We're going to make a couple of changes to the bundle now, including changing the app name:

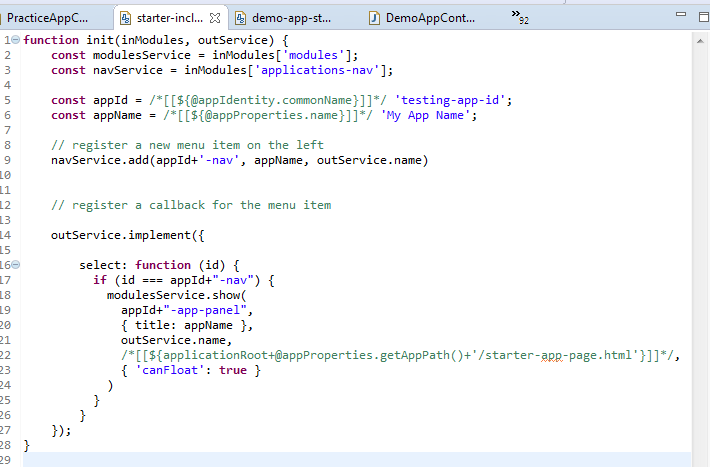


* We're changing the name of the application to **Demo App.**This will be reflected in the Symphony Market.
* Also, we're going to override the main controller javascript, to add our own html page (see 11 below)

## 11.  Adding Functionality

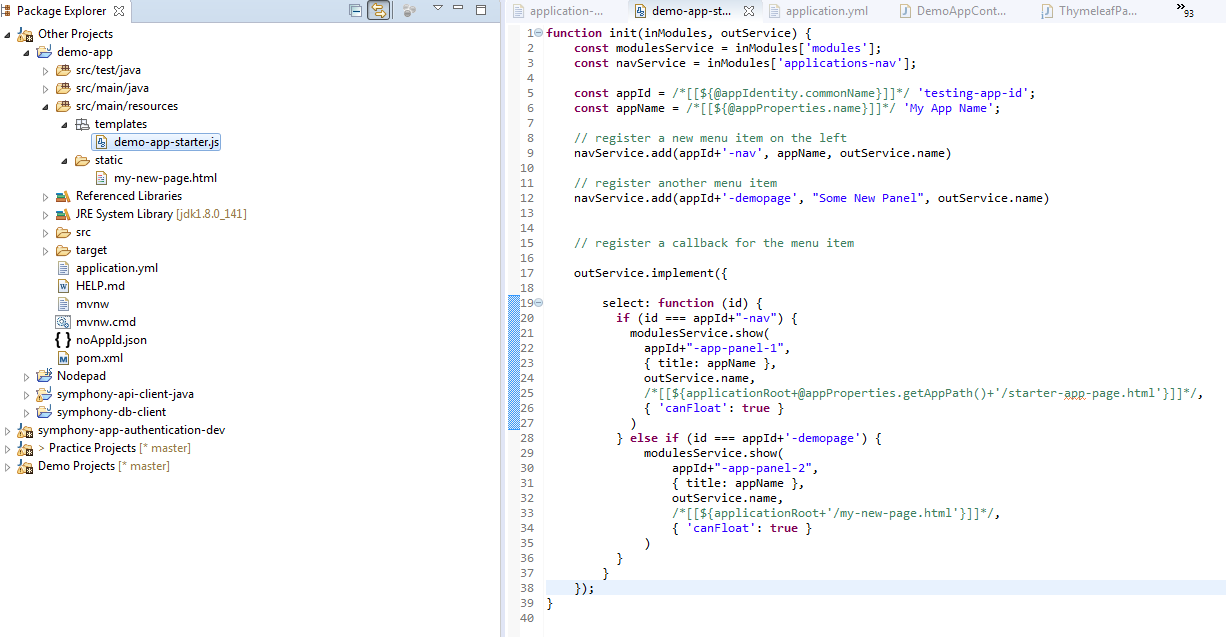
By default, the application calls a javascript file **/symphony-app/starter-include.js** when it starts up.  This provides the "Test Application" link on the left-hand navigator.

This is a Thymeleaf Template, and the source looks like this:



The code in /\* \*/ is actually part of Thymeleaf's Javascript templating system.  It will replace these parameters when the javascript is loaded.

* Let's copy this into a new file (**templates/demo-app-starter.js**), and add code to add a second page to the left-hand navigator:



Here is the same code again so you can copy-paste:

function init(inModules, outService) {  
 const modulesService = inModules['modules'];  
 const navService = inModules['applications-nav'];

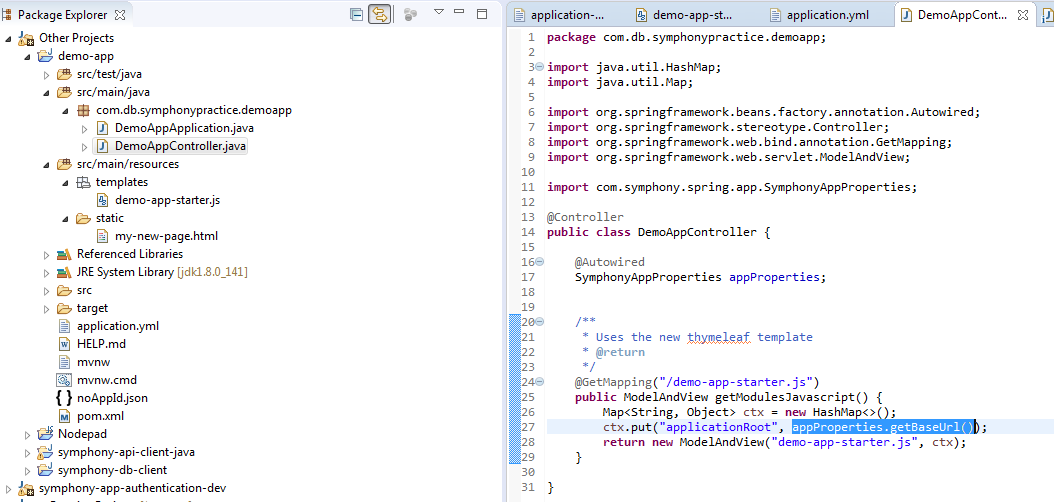
const appId = /\*[[${@appIdentity.commonName}]]\*/ 'testing-app-id';  
 const appName = /\*[[${@[appProperties.name](http://appProperties.name)}]]\*/ 'My App Name';  
   
 // register a new menu item on the left  
 navService.add(appId+'-nav', appName, [outService.name](http://outService.name))

// register another menu item  
 navService.add(appId+'-demopage', "Some New Panel", [outService.name](http://outService.name))

// register a callback for the menu item

outService.implement({  
   
 select: function (id) {  
 if (id === appId+"-nav") {  
 modulesService.show(  
 appId+"-app-panel-1",  
 { title: appName },  
 [outService.name](http://outService.name),  
 /\*[[${applicationRoot+@appProperties.getAppPath()+'/starter-app-page.html'}]]\*/,  
 { 'canFloat': true }  
 )  
 } else if (id === appId+'-demopage') {  
 modulesService.show(  
 appId+"-app-panel-2",  
 { title: appName },  
 [outService.name](http://outService.name),  
 /\*[[${applicationRoot+'/my-new-page.html'}]]\*/,  
 { 'canFloat': true }  
 )   
 }  
 }  
 });   
}

* In order to serve this javascript, we need to create a spring controller like so:



Here's the code for that:

package com.db.symphonypractice.demoapp;

import java.util.HashMap;  
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.servlet.ModelAndView;

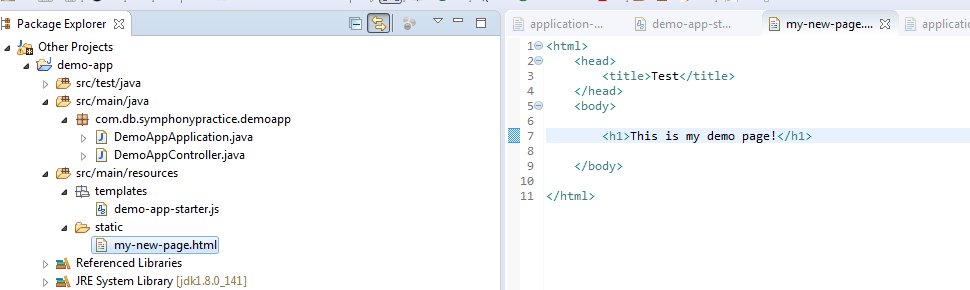
import com.symphony.spring.app.SymphonyAppProperties;

@Controller  
public class DemoAppController {

@Autowired  
 SymphonyAppProperties appProperties;  
   
   
 /\*\*   
 \* Uses the new thymeleaf template  
 \* @return  
 \*/  
 @GetMapping("/demo-app-starter.js")  
 public ModelAndView getModulesJavascript() {  
 Map<String, Object> ctx = new HashMap<>();  
 ctx.put("applicationRoot", appProperties.getBaseUrl());  
 return new ModelAndView("demo-app-starter.js", ctx);  
 }

}

* Finally, we'll need some HTML to serve.  By placing it in **/src/main/resources/static,** spring will happily serve it as a regular, static html file without further modification.

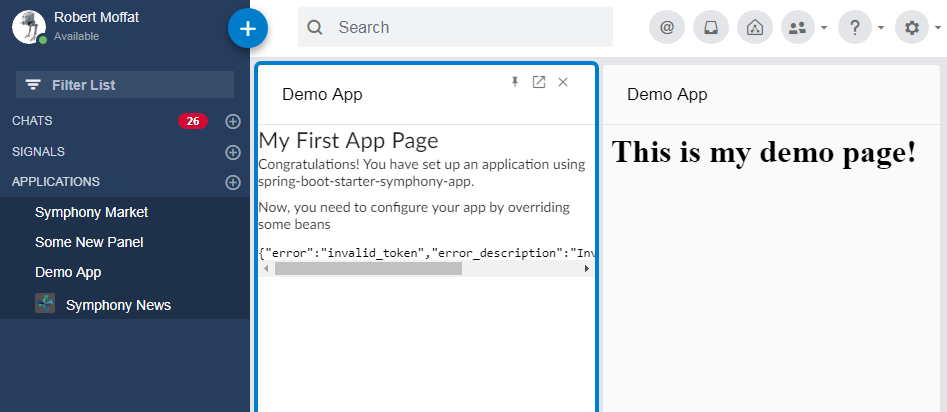


That looks like this:

<html>  
 <head>  
 <title>Test</title>  
 </head>  
 <body>  
   
 <h1>This is my demo page!</h1>  
   
 </body>

</html>

If all is well, your application should load up the new javascript, and display **two**menu options:



## 12.  Uploading the Bundle To The Pod

There are two bundle endpoints:

* **/symphony-app/bundle.json**: This endpoint is used for testing with the **?bundle=url**
* **/symphjony-app/<secret>/bundle.json:**This endpoint is used for uploading the bundle into the pod.  The structure is slightly different.
* You can customize the <secret> by setting the **symphony.app.apiKey** property for the application.
* In order to get your application installed permanently in the pod, the second bundle must be loaded into the Symphony admin console.  Provide the admin team with this bundle file to allow them to upload it.

You can tell your administrator whether to install the app for all users, or just for a select few.

## 13. Completing the Circle Of Trust

Add pod details