Table of Contents

Summary

- App Architecture
  - Backend+Client
    - Clustering
  - Client side only
- Components
  - Workbench
    - Components
      - Perspective
        - Type
        - Components
      - Panel
        - Components
      - Part
        - Components
  - Security
  - VFS
    - VFS
  - Javascript API
    - Angular Integration
  - App Styling
    - Bootstrap
    - PatternFly
- UberFire Extensions
  - Wires
  - Widgets
    - Ace Editor
    - Markdown
  - Property Editor
  - Security
  - Metadata
    - Automatic VFS Index/Search
  - Self Service App
  - RAD Extensions (tech module: Runtime Plugins)
  - App Directory
  - Third Part Extensions
  - Social Activities
  - Guvnor
  - Dashbuilder
  - Simple Docks
- Integration

Overview

UberFire is a Rich Client Platform to help rapidly build workbench or console applications for the web.

UberFire provides declarative APIs for building workbench-style apps, a rich virtual file system with a jgit backend, full-text search, a role-based declarative security framework, and more.

What is Uberfire

UberFire is a web framework for a superior experience in building extensible workbenches and console type applications

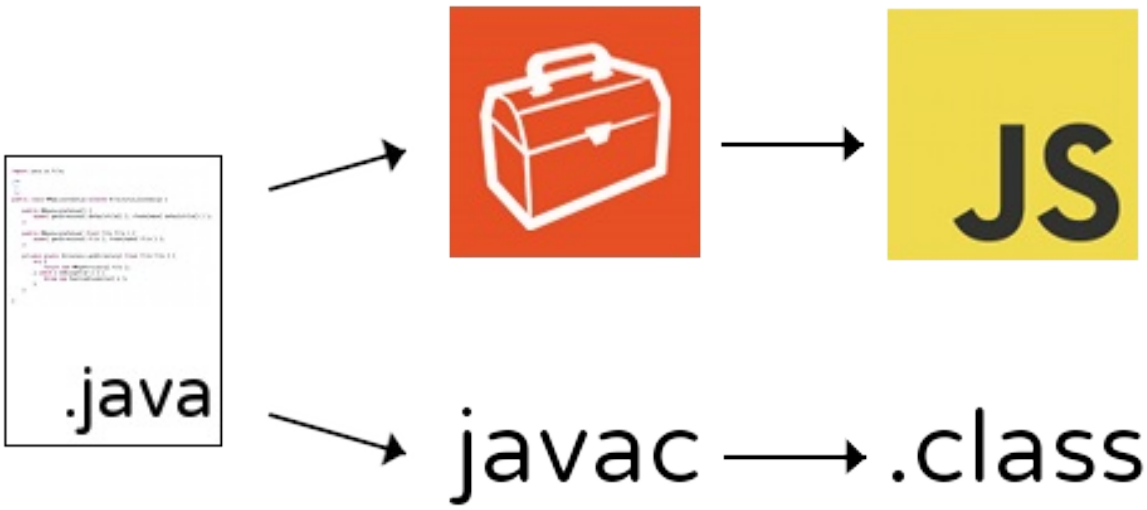
It provides an Eclipse like workbench experience for the web, helping you to make maintainable, customizable workbench-style apps in no time flat.

Our ultimate goal in Uberfire is to provide a strong ecosystem around it, based on a rich set of pluggable components and a strong infrastructure, allowing different type of users easily build Rich Web Apps on top of it.

Uberfire is the technology behind of Drools and jBPM web tooling and based on challenges and lessons learned during the workbench development of these projects.

How it all fits together

Build on the strengths of GWT and ERRAI, Uberfire allow you to write and maintain your application code in the Java programming language with all of the Java tooling at your disposal, then deploy it to the browser as a native JavaScript + HTML application.



Killer Features

Here we will highlight some key features of the framework.

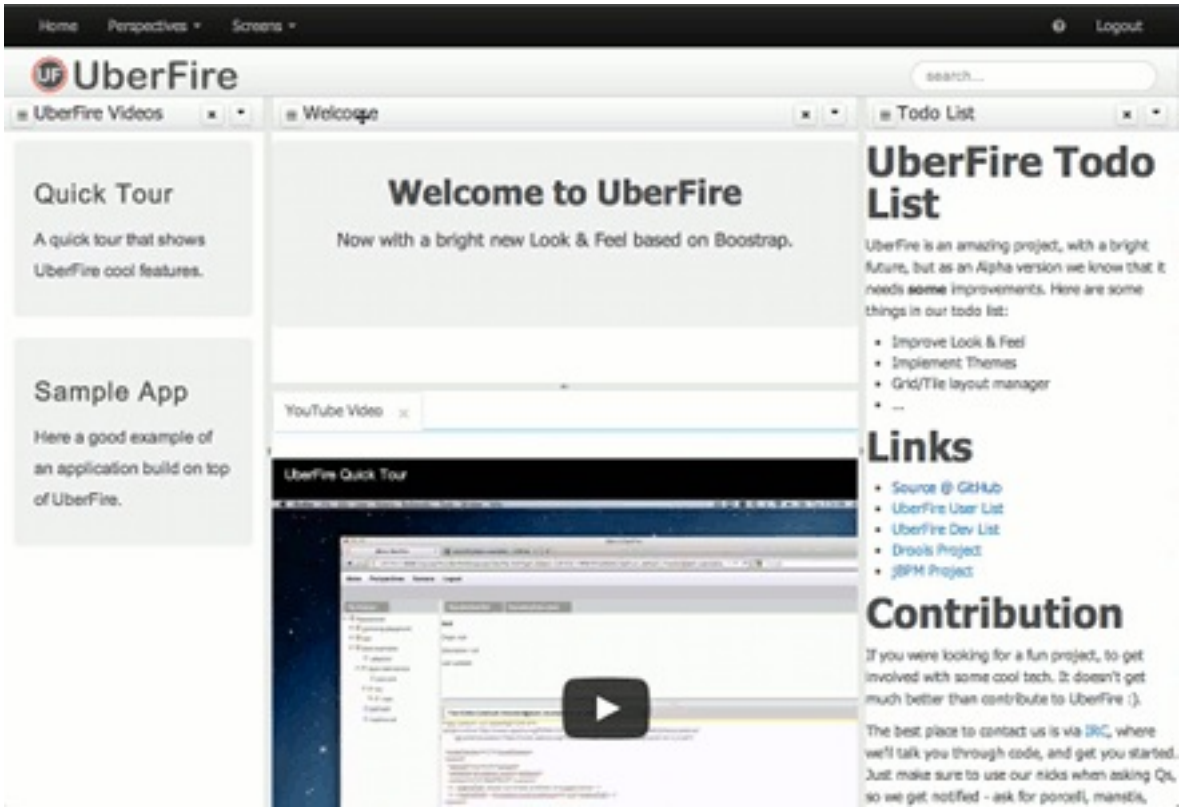
Extensible Plugins Architecture

One key aspect for UberFire is the compile time composition of plugins. Everything is a plugin, so it's very extensible. Uberfire also defines a set of interfaces and life cycle events, making it simple to build extensions of the framework.

Each plugin is a maven module, so when building a distribution, you simple need to add those maven modules as dependencies and they are avaliable to use in your web app.

Flexible Layout

Drag-and-drop layouts give your users control over their work environment.



Powerful VFS

UberFire has the power of GIT built in. A simple and clean NIO.2 based Virtual File System, using JGIT, ensures consistent APIs for both client and server. Supports change tracking and includes a metadata engine, full-text search, and security integration.

Client-side code that creates a file in the server-side VFS:

@Inject private FileSystem fs; @Inject private Caller vfsServices;

```
public void onSaveButtonClicked() {
  Path path = PathFactory.newPath(
      fs, "readme.txt", "default://readme.txt");
  vfsServices.call().write(path, "Hello World!");
}
```

Fine-Grained Security

Fully pluggable authentication and authorization system. Includes file, database, PicketLink and JAAS out-of-the-box.

Native Plugin System

Develop plugins in Java against our declarative, typesafe APIs, or choose your favorite JavaScript framework and develop using that.

Plugins can contribute new perspectives, views, editors, menu items, and more to an UberFire project.

Modular Design

Use the parts you want, and leave the rest behind.

Ready For Clustering

UberFire works flawlessly in clustered and highly-available deployments. GIT allows for a decentralised cluster, with efficient binary replication of content between nodes. Enjoy HA out-of-the-box: load balancing and failover just work!

Open Source

Developed in the open by the people who use it. Join us!

```
$ git clone https://github.com/uberfire/uberfire.git
$ cd uberfire
$ mvn clean install
```

Getting Started

To demonstrate how easy it is to create your first web application in Uberfire Framework, this guide will help you create and install your first UberFire application on your own computer.

This will let you try out an UberFire application, prove the UberFire apps work with your setup and make some changes in order to taste Uberfire concepts.

5 minutes Introduction

This session teaches you how to build your first Uberfire App in just 5 minutes.

Prerequisites

This guide assumes you have the following software set up and working on your computer:

- A Java Development Kit (JDK) version 6 or newer
- Maven 3.x

Creating your first App

In a command line, run the archetype to create your first app. [TODO Change archetype version after release]

```
$ mvn archetype:generate -DarchetypeGroupId=org.uberfire -DarchetypeArtifactId=uberfire-proj
```

Note: If you get an error including "The desired archetype does not exist...", you may need to change the version "0.7.0.Beta2" to "0.7.0-FINAL".

Maven will ask for your groupId, let's use:

```
org.uberfire
```

For the artifactId value:

```
demo
```

For version

```
1.0-SNAPSHOT
```

For package

```
org.uberfire
```

Use default value for package, for capitalizedRootArtifactId use:

```
Demo
```

Confirm your changes and then you should see the message:

```
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 3:32.347s
[INFO] Finished at: Wed Jan 14 20:30:27 BRST 2015
[INFO] Final Memory: 29M/959M
[INFO] ------------------------------------------------------------
`
```

Building your first App

To build your first app, go to directory demo, build the project in maven and wait for the build finish.

```
$ cd demo
$ mvn clean install
```

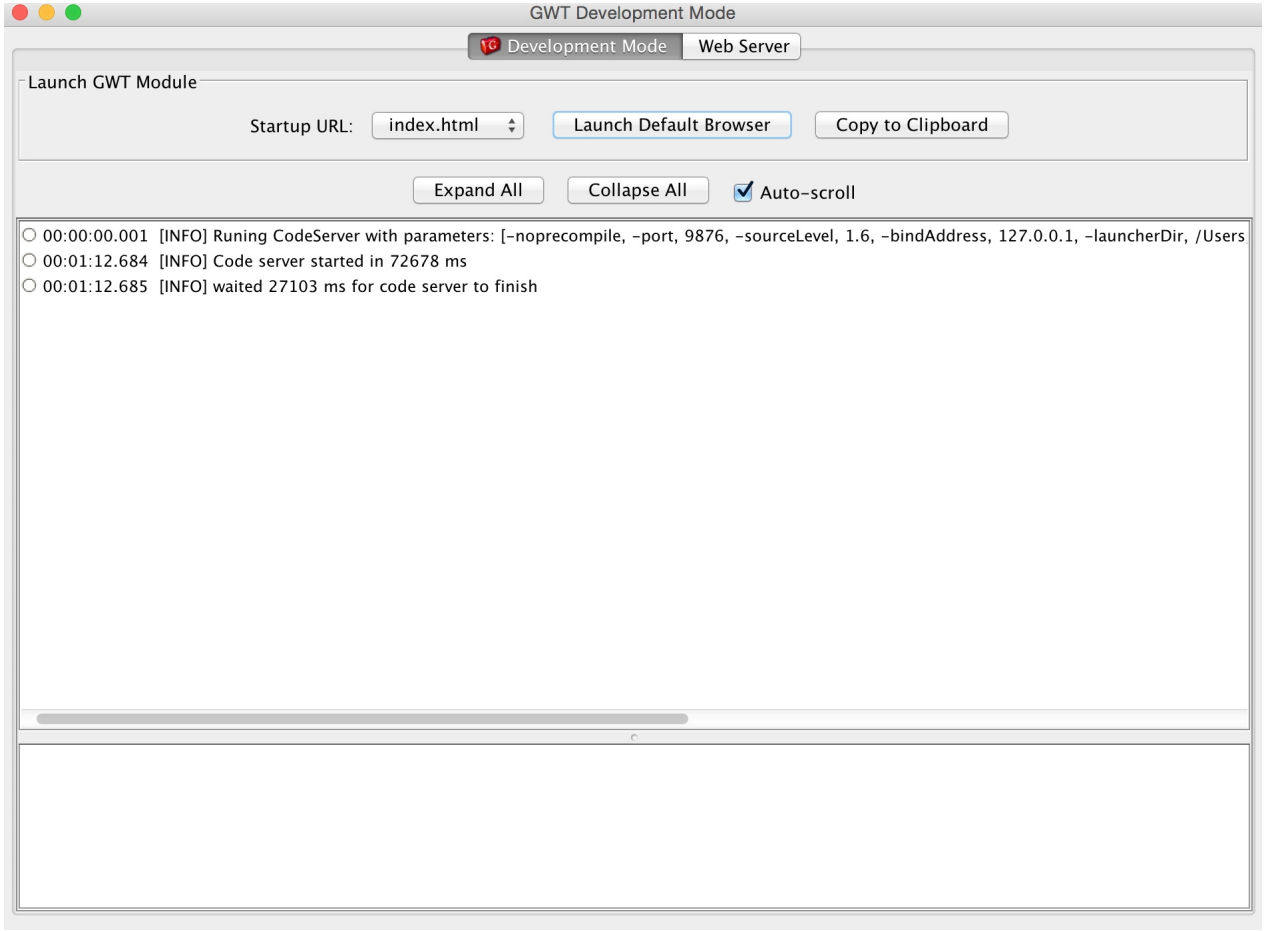
You should see the maven build success message again.

See it work!

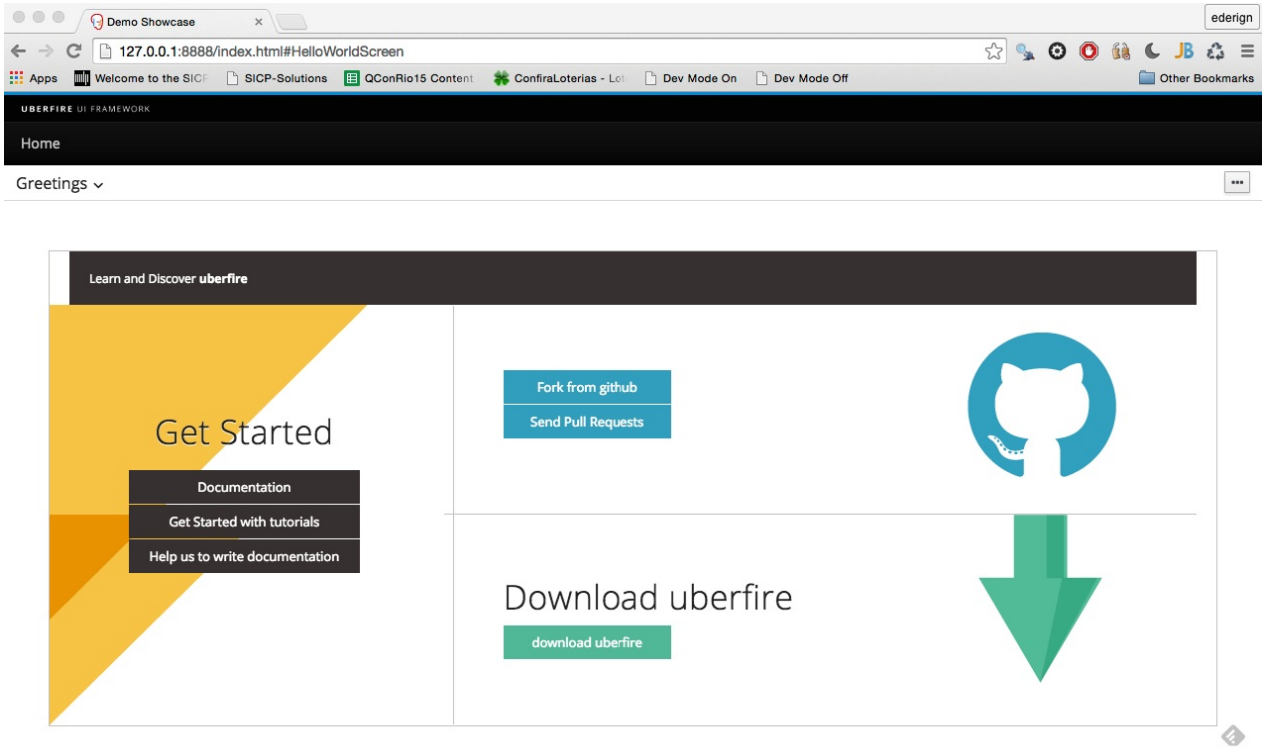
How about running our first project?

```
cd demo-showcase/demo-webapp
mvn clean gwt:run
```

Wait for GWT console build your app:



Click on "Launch Default Browser" to open your Uberfire App. Log in as admin (Username) with admin (as Password). You should see our Hello World.



When you see that Hello World, you are able to soon taste Uberfire power.

Improving your first App

In this session, we will create some basic Uberfire components aiming to give you an idea of how Uberfire works. For now, doesn't pay to much attention to new terms and concepts that we will present, it's time to only have fun. The Uberfire Architeture and details of how everything glues together will be presented in the Tutorial section.

Felling Uberfire

Let's change our App so we can get a better feel for how Uberfire workbench perspectives and panels fit together.

We'll create two screens backed by a simple model class to demonstrate how you'd typically separate model from view in an UberFire application and how screens communicate in a decoupled way.

Creating our model

The data model in an UberFire app is typically represented by Plain Old Java Objects, (POJOs). This leaves you the flexibility to use them in other frameworks that like POJOs such as JPA, JAXB, Errai Data Binding, and much more by adorning them with annotations. For now, our extremely simple data model will just be an unadorned POJO.

The model class will be called Mood, and it will represent how the current user is feeling at the moment. Place it on org.uberfire.shared package of your web app.

```
package org.uberfire.shared;

public class Mood {

    private final String text;

    public Mood( String text ) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    @Override
    public String toString() {
```

```
        return text;
    }
}
```

Creating MoodScreen, a Templated Widget

For MoodScreen, let's use the Errai UI Template system. This approach is similar to GWT UiBinder, but it lets you create the template in a plain HTML 5 file rather than a specialized UiBinder XML file.

Create a HTML file named MoodScreen.html inside Java package org.uberfire.client.screens with this content:

```
<div>
    <div style="border: 1px solid red; padding: 30px">
      <input data-field=moodTextBox type=text placeholder="How do you feel?">
    </div>
</div>
```

Create a Java class "MoodScreen.java" in the package org.uberfire.client.screens. This file will be used as a client-side template for the new MoodScreen widget. Here's what that looks like:

```
package org.uberfire.client.screens;

import javax.enterprise.context.Dependent;
import javax.enterprise.event.Event;
import javax.enterprise.event.Observes;
import javax.inject.Inject;

import org.jboss.errai.ui.shared.api.annotations.DataField;
import org.jboss.errai.ui.shared.api.annotations.EventHandler;
import org.jboss.errai.ui.shared.api.annotations.Templated;
import org.uberfire.client.annotations.WorkbenchPartTitle;
import org.uberfire.client.annotations.WorkbenchScreen;
import org.uberfire.client.util.Layouts;
import org.uberfire.shared.Mood;

import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyDownEvent;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.TextBox;

@Dependent
@Templated
@WorkbenchScreen(identifier="MoodScreen")
public class MoodScreen extends Composite {
```

```
    @Inject
    @DataField
    private TextBox moodTextBox;

    @Inject Event<Mood> moodEvent;

    @WorkbenchPartTitle
    public String getScreenTitle() {
        return "Change Mood";
    }

    @EventHandler("moodTextBox")
    private void onKeyDown(KeyDownEvent event) {
        if (event.getNativeKeyCode() == KeyCodes.KEY_ENTER) {
            moodEvent.fire(new Mood(moodTextBox.getText()));
            moodTextBox.setText("");
        }
    }

}
```

MoodScreen is very similar to HelloWorldScreen. The only structural differences are related to our choice to use an Errai UI Template. See more about Errai UI templates in this guide.

Creating MoodListenerScreen

Create a HTML file named MoodListenerScreen.html inside Java package org.uberfire.client.screens with this content:

```
<div>
    <div style="border: 1px solid red; padding: 30px">
        <input data-field=moodTextBox type=text placeholder="I understand that you are feeli
    </div>
</div>
```

And create MoodListenerScreen.java, inside org.uberfire.cliente.screens:

```
package org.uberfire.client.screens;

import javax.enterprise.context.Dependent;
import javax.enterprise.event.Event;
import javax.inject.Inject;

import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyDownEvent;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.TextBox;
```

```
import org.jboss.errai.ui.shared.api.annotations.DataField;
import org.jboss.errai.ui.shared.api.annotations.EventHandler;
import org.jboss.errai.ui.shared.api.annotations.Templated;
import org.uberfire.client.annotations.WorkbenchPartTitle;
import org.uberfire.client.annotations.WorkbenchScreen;
import org.uberfire.shared.Mood;

@Dependent
@Templated
@WorkbenchScreen(identifier="MoodListenerScreen")
public class MoodListenerScreen extends Composite {

    @Inject
    @DataField
    private TextBox moodTextBox;


    @WorkbenchPartTitle
    public String getScreenTitle() {
        return "MoodListenerScreen";
    }

}
```

Giving MoodScreen, a perspective

Let's create our first perspective, using Uberfire Templated Perspectives.

First, we need to create the perspective Errai UI template, named "MoodPerspective.html" on org.uberfire.client.perspectives package:

```
<div>
    <div id="home1">
        <span><b>Our MoodScreen</b></span>

        <div data-field="moodScreen"></div>
    </div>
    <div id="home2">
        <span><b>Mood Listener</b></span>

        <div data-field="moodListener"></div>

    </div>
</div>
```

Now, let's create the Perspective class MoodPerspective on org.uberfire.client.perspectives package:

```
package org.uberfire.client.perspectives;
```

```
import javax.enterprise.context.ApplicationScoped;

import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.FlowPanel;
import org.jboss.errai.ui.shared.api.annotations.DataField;
import org.jboss.errai.ui.shared.api.annotations.Templated;
import org.uberfire.client.annotations.WorkbenchPanel;
import org.uberfire.client.annotations.WorkbenchPerspective;
import org.uberfire.client.workbench.panels.UFFlowPanel;

@ApplicationScoped
@WorkbenchPerspective(identifier = "MoodPerspective")
@Templated
public class MoodPerspective extends Composite {

    @DataField
    @WorkbenchPanel(parts = "MoodScreen")
    UFFlowPanel moodScreen = new UFFlowPanel(100);

    @DataField
    @WorkbenchPanel(parts = "MoodListenerScreen")
    UFFlowPanel moodListener = new UFFlowPanel(100);

}
```

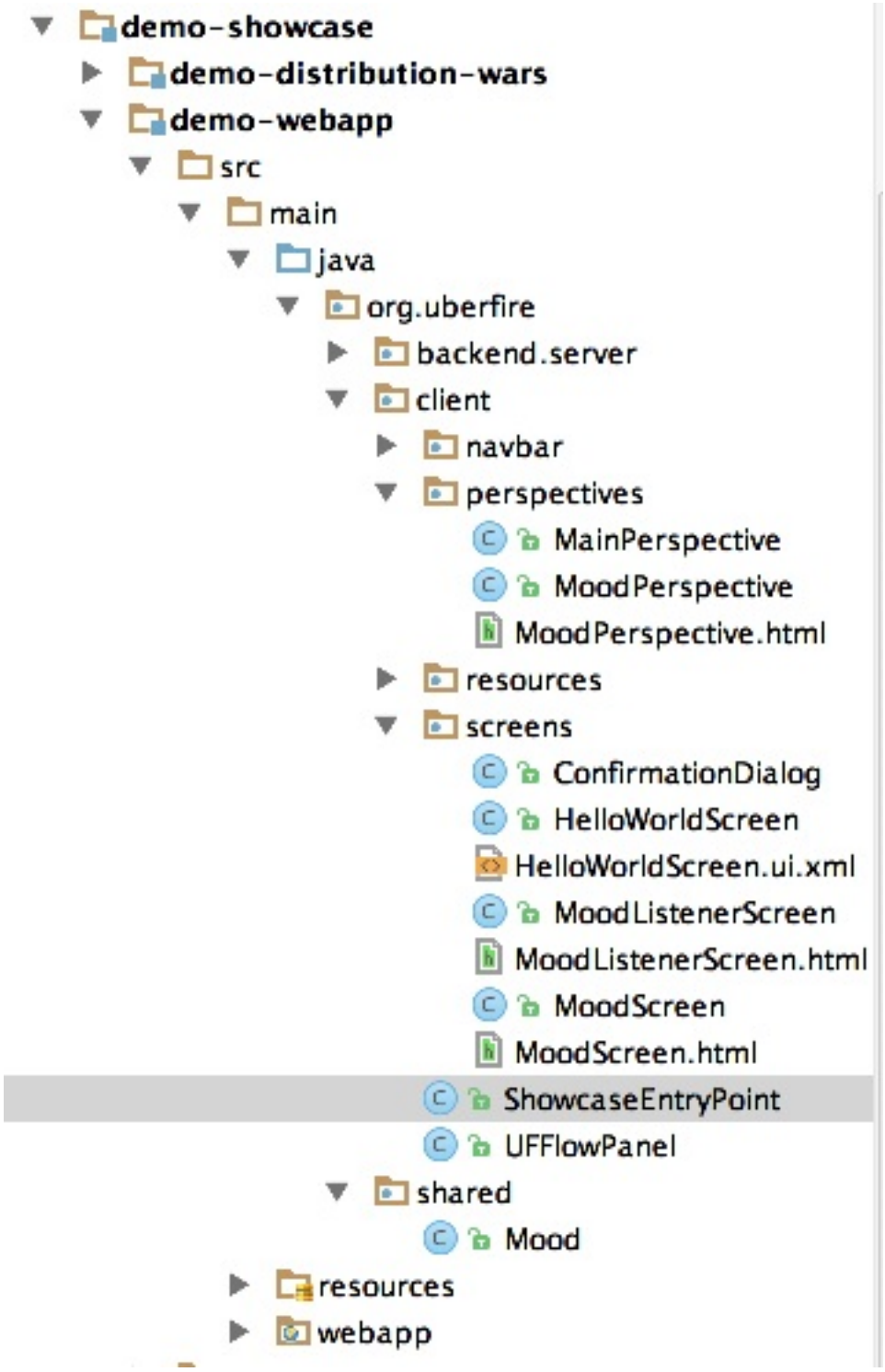
Adding MoodPerspective

Moving on, let's add MoodPerspective to the menu bar of our app. We need to update org.uberfire.client.ShowcaseEntryPoint and replace setupMenu method to that:

```
 private void setupMenu( @Observes final ApplicationReadyEvent event ) {
        final Menus menus =
                newTopLevelMenu( "Home" )
                        .respondsWith( new Command() {
                            @Override
                            public void execute() {
                                placeManager.goTo( new DefaultPlaceRequest( "MainPerspective
                            }
                        } )
                        .endMenu()
                        .newTopLevelMenu( "Mood Perspective" )
                        .respondsWith( new Command() {
                            @Override
                            public void execute() {
                                placeManager.goTo( "MoodPerspective" );
                            }
                        } )
                        .endMenu()
                        .build();

        menubar.addMenus( menus );
    }
```

Check your work

It's time to check your classes and package created. See an example here:

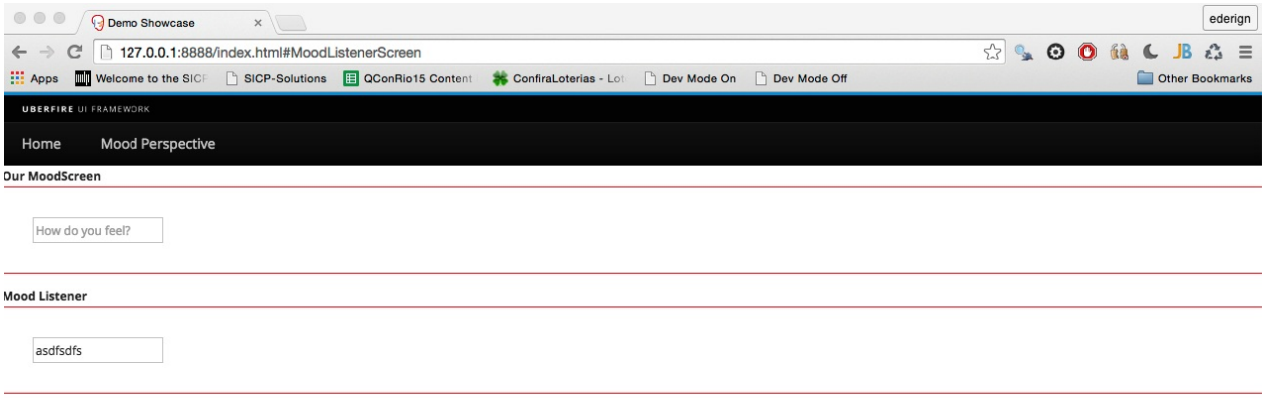


See it work!!!

How about seeing our changes?

```
cd demo-showcase/demo-webapp
mvn gwt:compile gwt:run
```

Click on MoodPerspective menu:



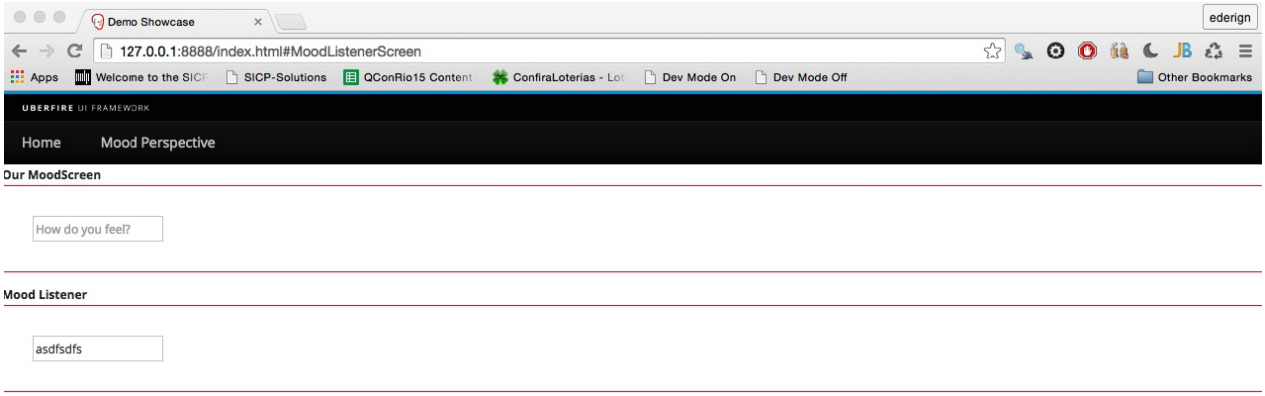
Let's make the screens communicate

Did you notice the CDI event raised by MoodScreen? If no, take a look at onKeyDownMethod.

Now let’s do something in in response to the the event we fire in MoodListenerScreen when the user presses Enter. To do this we’ll add a CDI observer method at MoodListenerScreen:

```
public void onMoodChange(@Observes Mood mood) {
        moodTextBox.setText("You are feeling " + mood.getText());
}
```

Build and run your App again (mvn gwt: clean gwt:compile gwt:run), write a text on "How do you fell" textbox and press enter to see screens communicating:



A taste on Uberfire lifecycle events

Uberfire support a lot of workbench events, let's see them working?

Edit MoodPerspective.java and add this two methods, run the app again and change perspectives to see the events happening.

```
@OnOpen
public void onOpen(){
    Window.alert( "On Open" );
}

@OnClose
public void OnClose(){
    Window.alert( "On Close" );
}
```

Build (mvn clean install) and run your App again (mvn clean gwt:run) and change perspectives to see the events raising.

Running Uberfire Showcase

This guide will help you install an UberFire demo application on your own computer. This will help let you try out an UberFire application and see our show case demo.

Get an app server

The pre-built UberFire Showcase Web App is available for JBoss AS 7.1.1, EAP 6.2, and Tomcat.

If you don't already have one of these app servers installed on your computer, don't worry. In all cases, installing is as easy as downloading and unzipping.

Wildfly 8.1	**Download**
Tomcat 7	**Download**

Once the file has finished downloading, unzip it wherever you like.

Start the app server

Now start the app server using a command line terminal. Use the *cd* command to change the working directory of your terminal to the place where you unzipped the application server, then execute one of the following commands, based on your operating system and choice of app server:

Server	*nix, Mac OS X	Windows
WildFly	bin/standalone.sh	bin\standalone.bat
Tomcat	bin/startup.sh	bin\startup.bat

Then visit the URL http://localhost:8080/ and you should see a webpage confirming that the app server is running.

Get the pre-built WAR file

Now download the pre-built WAR file for the server you've installed and started.

Wildfly	Download
Tomcat 7	Download

Note: If you find that the war file has limited functionality, e.g. some perspectives do not work correctly, please build it from source.

Build from source

You can build your archetype from the lastest source.

```
$ git clone git@github.com:uberfire/uberfire.git
$ cd uberfire
$ mvn clean install
```

The war's will be located at:

```
../uberfire/uberfire-showcase/showcase-distribution-wars/target
```

Deploy the WAR

Rename the downloaded WAR file to **uberfire-showcase.war** and copy it into the auto-deployment directory for your app server:

Wildfly 8.1	standalone/deployments/
Tomcat 7	**webapps/**

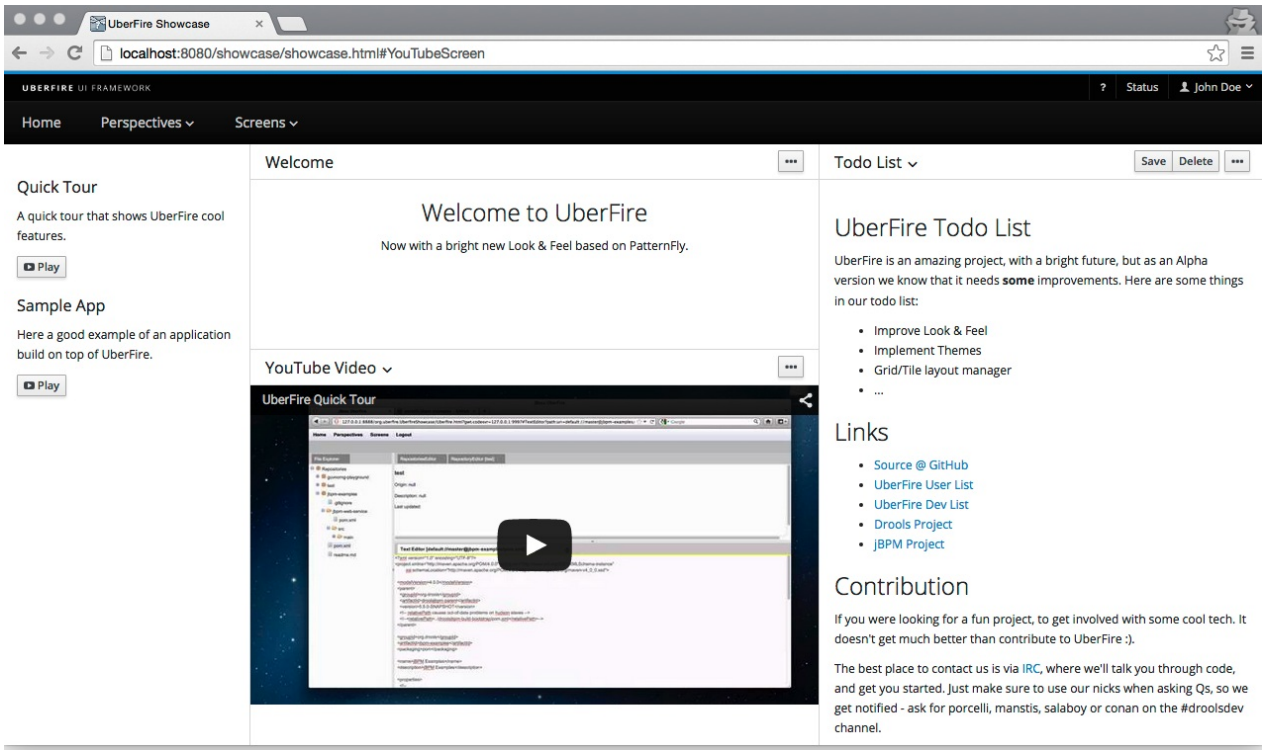
Example: Wildfly 8.1 on Unix/Linux/Mac:

```
$ mv ~/Downloads/showcase-distribution-wars-0.4.0-20131125.172155-223-jboss-as7.0.war standa
```

You should notice some disk activity, and if you're using Wildfly, you will see some logging on the console where you started the server. Tomcat logs only to files by default, so you won't see logs in your Tomcat terminal at this stage.

See it work!

Now visit http://localhost:8080/uberfire-showcase/ and sign in with username **admin**, password **admin**.



Once you're logged in, here are some things to try:

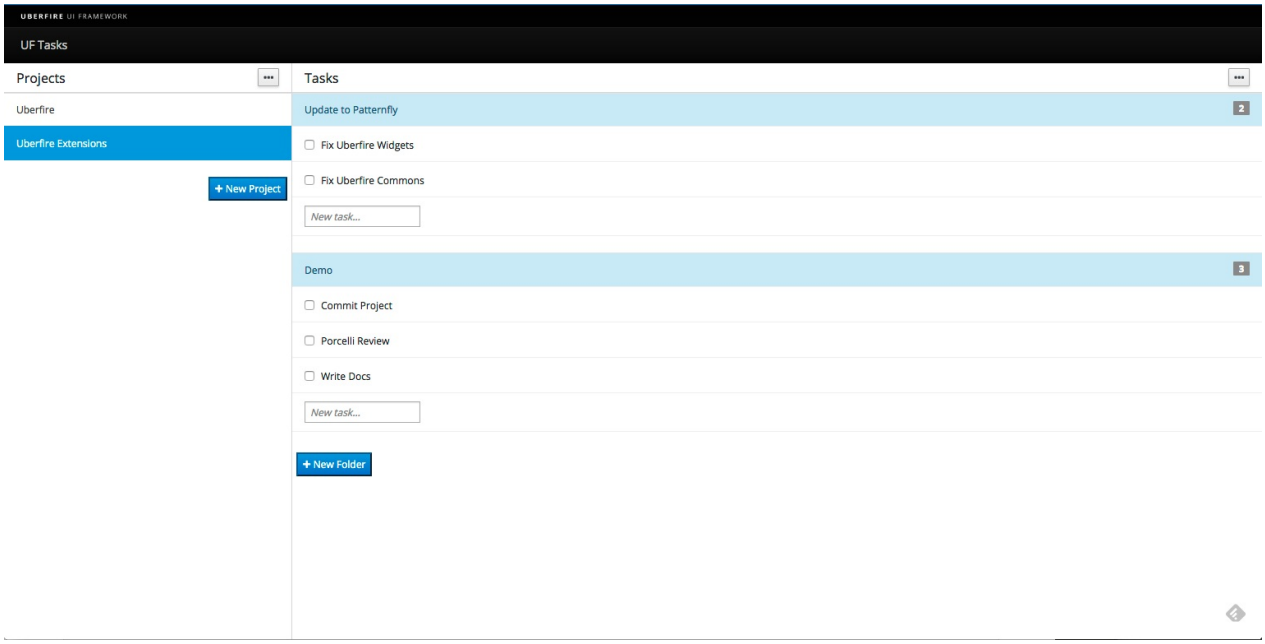
- Try switching perspectives in the perspective menu.
- On Home Perspective, try dragging a view clicking on TODO List and moving it from one panel to another.
- Try adding a new location for views by dragging a view near the edge of the screen.
- See some useful features on Screens menu..

What next?

Now that you've created and deployed an UberFire app and had some hands-on time with it, why not create a more complex App? Our tutorial guide provides a detailed walkthrough to create a Task Manager using Uberfire.

Tutorial

This guide will walk you through the process of setting up a new UberFire application. Starting from an empty directory, you will create a simple Task Manager, learning how everything works along the way.



If you've never seen a full-blown working UberFire app before, why not check out our Getting Started Guide first? It will help you get UberFire's pre-made showcase application up and running on your system without getting bogged down in details.

This in-depth guide will be here waiting for you when you're ready to scratch the surface and build something of your own.

So, assuming you're already familiar with what UberFire can do, let's get started!

Prerequisites

This guide assumes you have the following software set up and working on your computer:

- A Java Development Kit (JDK) version 6 or newer
- Maven 3.x
- IntelliJ IDEA or Eclipse IDE for Java EE Developers

Creating UF Tasks Project

Your UF Tasks project will follow the standard Maven project layout. So let's create it using our archetype.

Creating UF Tasks Project

In a command line, run the archetype to create your first app. [TODO Change archetype version after release]

```
$ mvn archetype:generate -DarchetypeGroupId=org.uberfire -DarchetypeArtifactId=uberfire-proj
```

Note: If you get an error including "The desired archetype does not exist...", you may need to change the version "0.7.0.Beta2" to "0.7.0-FINAL".

Maven will ask for your groupId, let's use:

```
org.uberfire
```

For the artifactId value:

```
uftasks
```

For version

```
1.0-SNAPSHOT
```

For package

```
org.uberfire
```

Use default value for package, for capitalizedRootArtifactId use:

```
UFTasks
```

Confirm your changes and then you should see the message:

```
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 3:32.347s
[INFO] Finished at: Wed Jan 14 20:30:27 BRST 2015
[INFO] Final Memory: 29M/959M
[INFO] ------------------------------------------------------------
`
```

Building your App

To build your app, go to directory uftasks, build the project in maven and wait for the build finish.

```
$ cd uftasks
$ mvn clean install
```

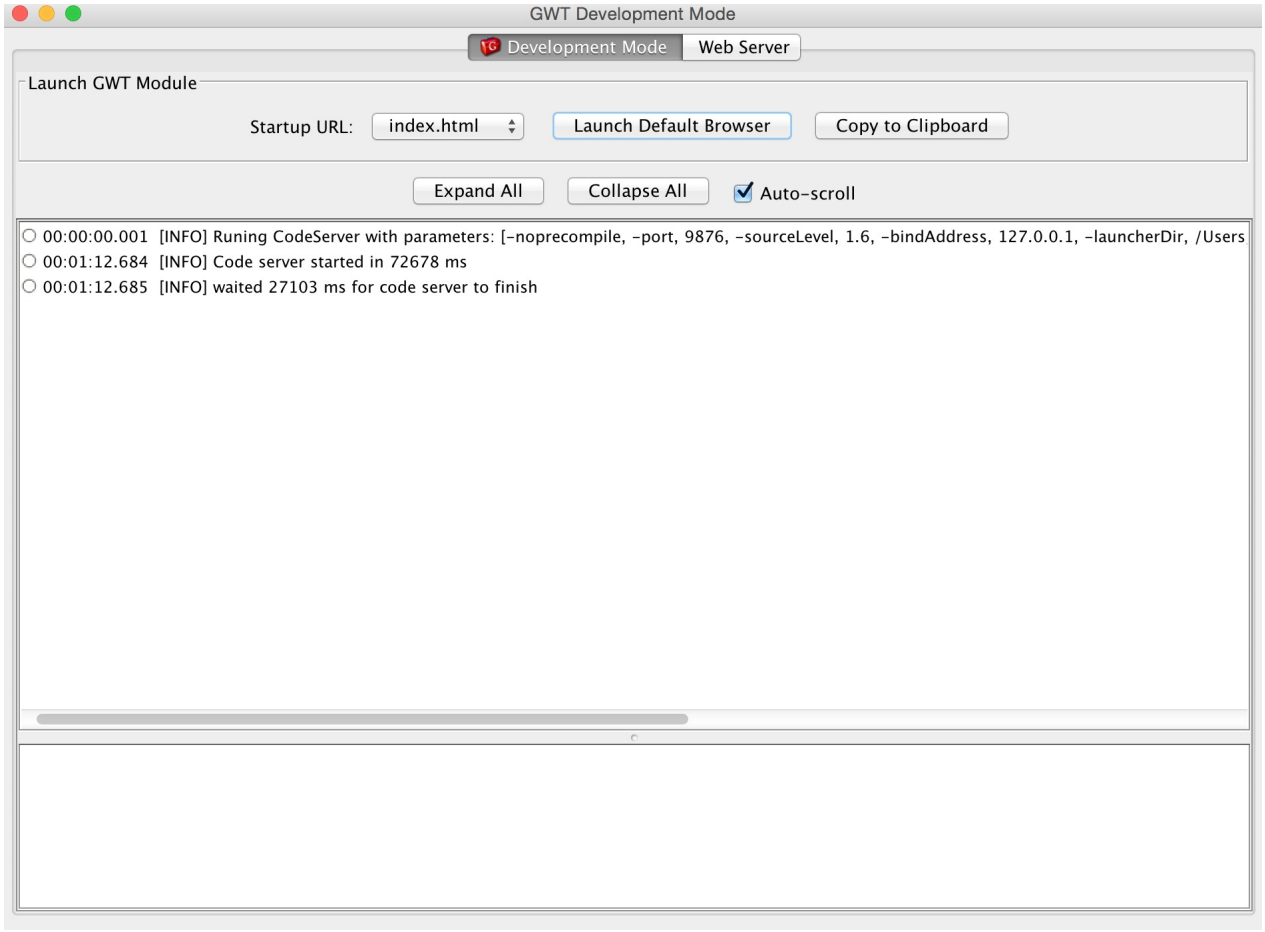
You should see the maven build success message again.

See it work!

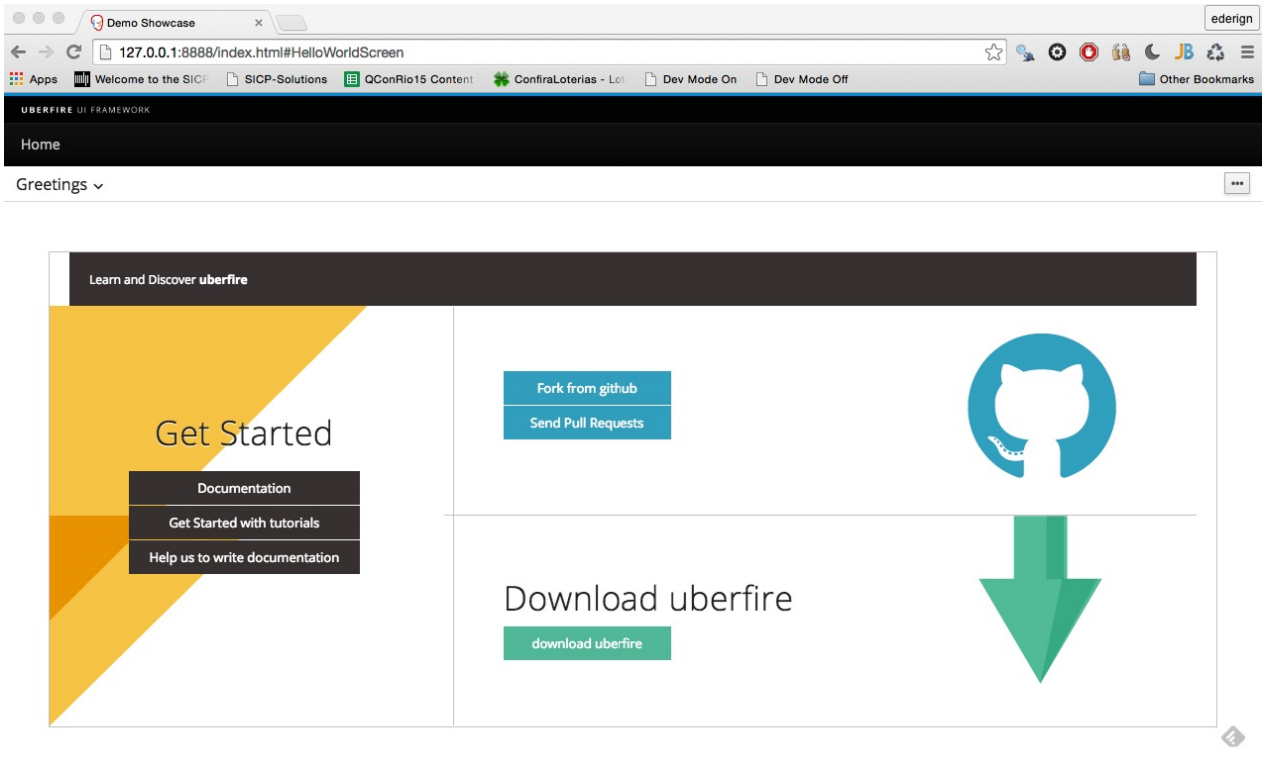
How about running our project?

```
cd uftasks-showcase/uftasks-webapp
mvn clean gwt:run
```

Wait for GWT console build your app:



Click on "Launch Default Browser" to open your Uberfire App. Log in as admin (Username) with admin (as Password). You should see our Hello World.



When you see that, you are good to go.

Layout of Uberfire Archetype

Your UberFire project will follow the standard Maven project layout. Most open source Java projects follow this layout these days, so this should look familiar.

Here’s a rundown of the specific files and directories you’ll find in every UberFire project generated from Uberfire Archetype. Don’t get hung up on the details yet. We’ll get to each of these in turn.

Archetype Structure

- **bom**: "bill of materials" of the archetype. It defines the versions of all the artifacts that will be created in the library
- **parent-with-dependencies**: declares all dependency versions of the archetype.
- **component**: an example of uberfire component project.
- **showcase**: uberfire showcase directory, that contains the web app and distribution-wars.

Setup Dev Enviroment

Before you start building our project, it is important to set up your favorite development environment.

To run your web app, Uberfire uses Errai's EmbeddedWildFlyLauncher, that provides an embedded WildFly instance within Dev Mode to allow users to debug server- and client-side code simultaneously in a single JVM.

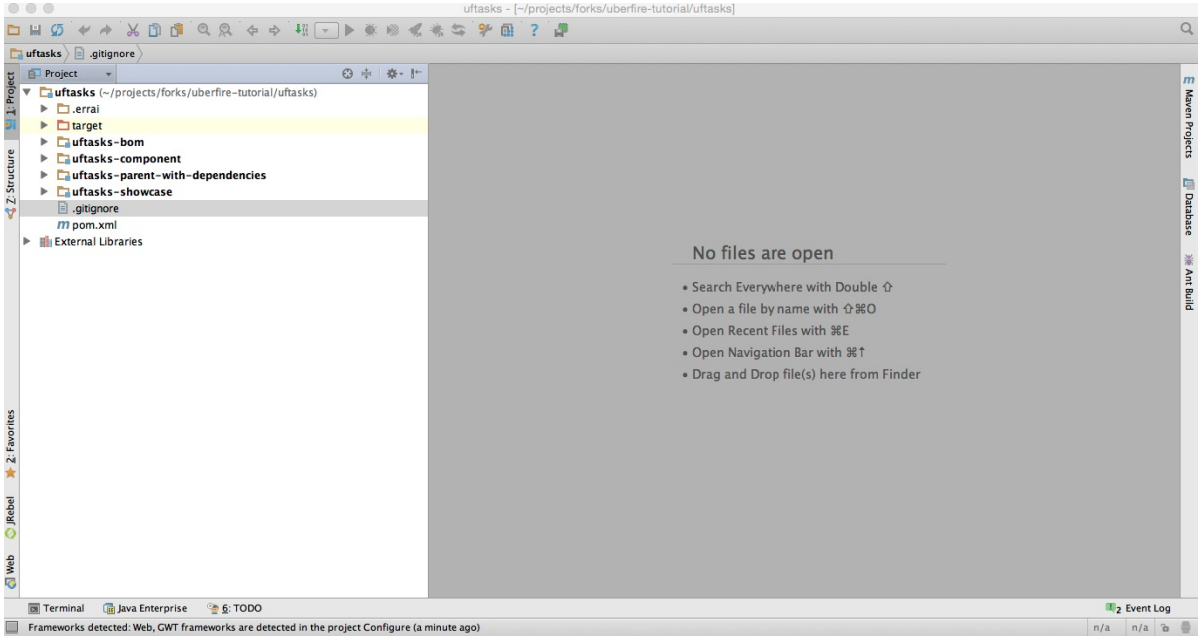
This section will detail instructions for using it in Eclipse and Intellij IDEA Ultimate Edition:

Intellij IDEA

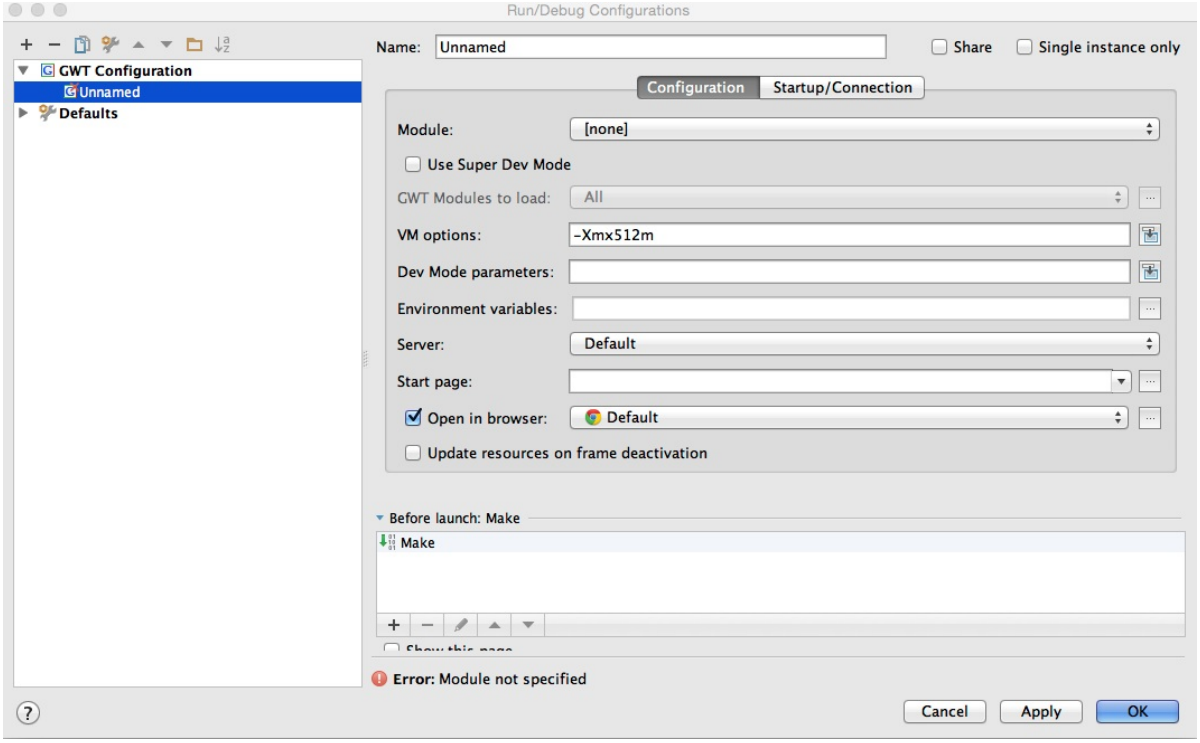
The Ultimate Edition for Intellij IDEA comes with a built-in GWT plugin that allows you to run and debug GWT apps specifically. We can configure the plugin to use the embedded WildFly launcher for debugging ease, in order to debug both server and client-side code in one debug session. This section describes how to set up a GWT run/debug configuration within Intellij IDEA.

(This instructions was tested on IntelliJ IDEA 14.1.3 - Ultimate)

1. Click on open, and point to pom.xml file located on uftasks dir. (Wait for import and indexing)



2. Access Run menu, select "Edit Configurations", press the + button under GWT Configurations to create a new GWT configuration.



3. Fill out the following parameters in the corresponding boxes:

Name: UFTasks

Module: uftasks-webapp

Use Super Dev Mode: yes (check it)

GWT Modules to load: org.uberfire.FastCompiledUFTasksShowcase

(if your Gwt Module are not displayed on this combobox, go to FAQ section and look for fix maven version on IDEA).

VM options: -Xmx2048m -XX:MaxPermSize=512M -Derrai.jboss.home=/Users/ederign/projects/uftasks/uftasks-showcase/uftasks-webapp/target/wildfly-8.1.0.Final/

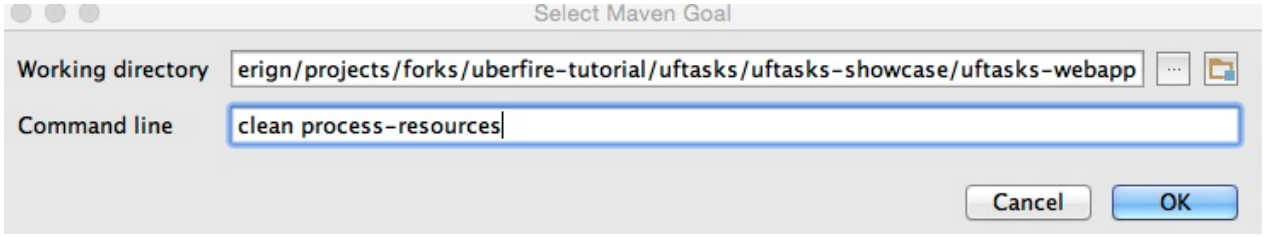
Where errai.jboss.home points to your WildFly installation directory. This can either be your local WildFly installation directory, or in the target/ directory of your app. For the UFTasks app, errai.jboss.home points to the WildFly installation within the target/ directory, which is redownloaded and installed as part of the build.

Dev Mode Parameters: -server org.jboss.errai.cdi.server.gwt.EmbeddedWildFlyLauncher

With JavaScript debugger: optionally (you'll need install an IntelliJ plugin in your

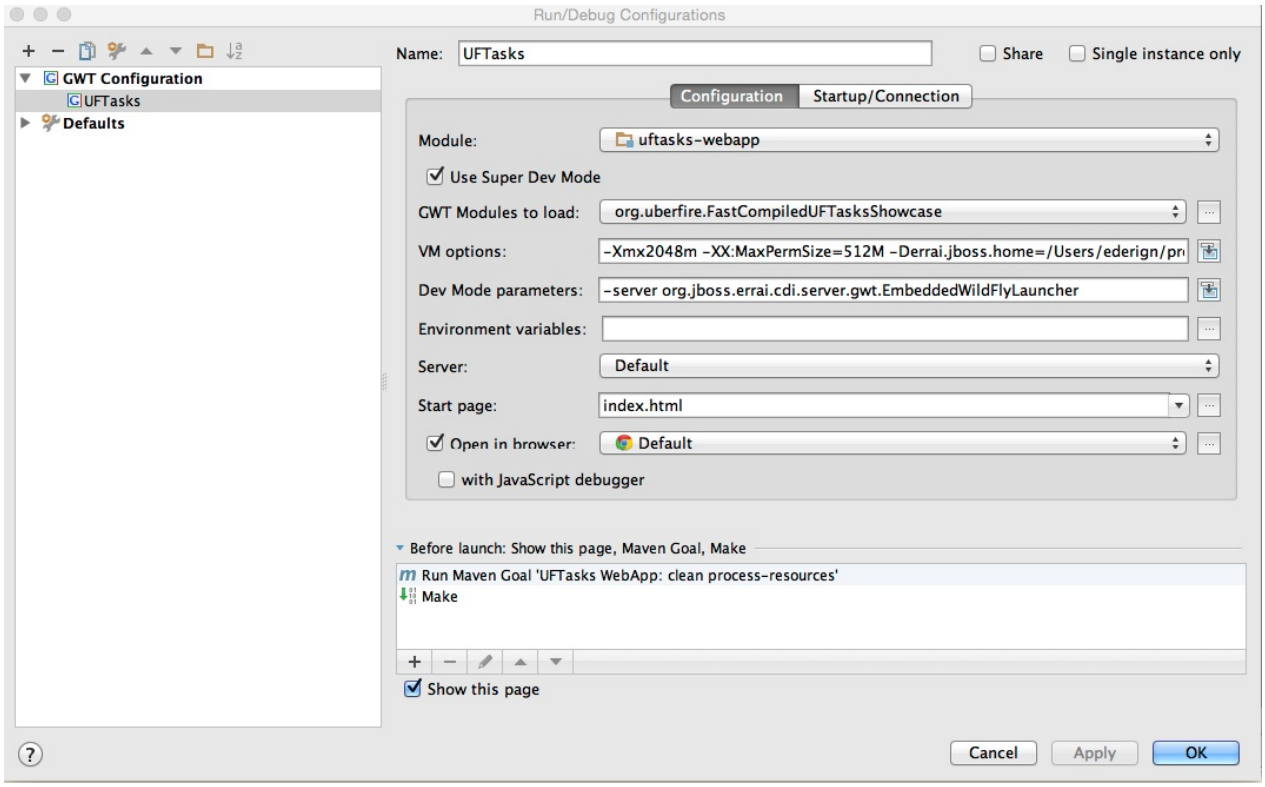
browser in order to have a better debug experience)

Before Launch: Add a new “Run Maven Goal” configuration **before** your make. Press + button, "Run Maven Goal"

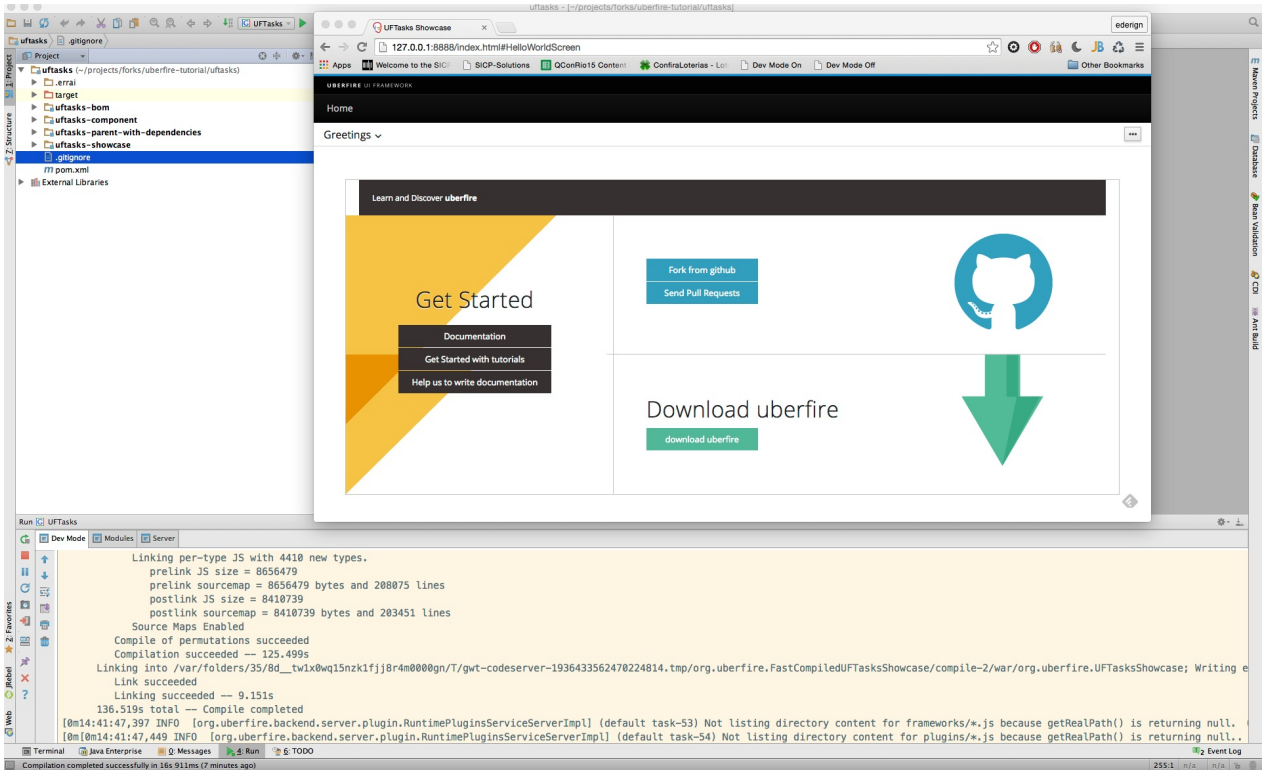


Point “Working directory” to your webapp folder, Set “clean process-resources" in line parameter, press OK, and move this new configuration *before* Make.

This is what your config should look like:



To run or debug your app, select this configuration in the top right corner of Intellij IDEA and click the Run or Debug buttons next to it. Your app should start up in Dev Mode within Intellij automatically and you should be able to use Intellij’s own debugger to debug your code.

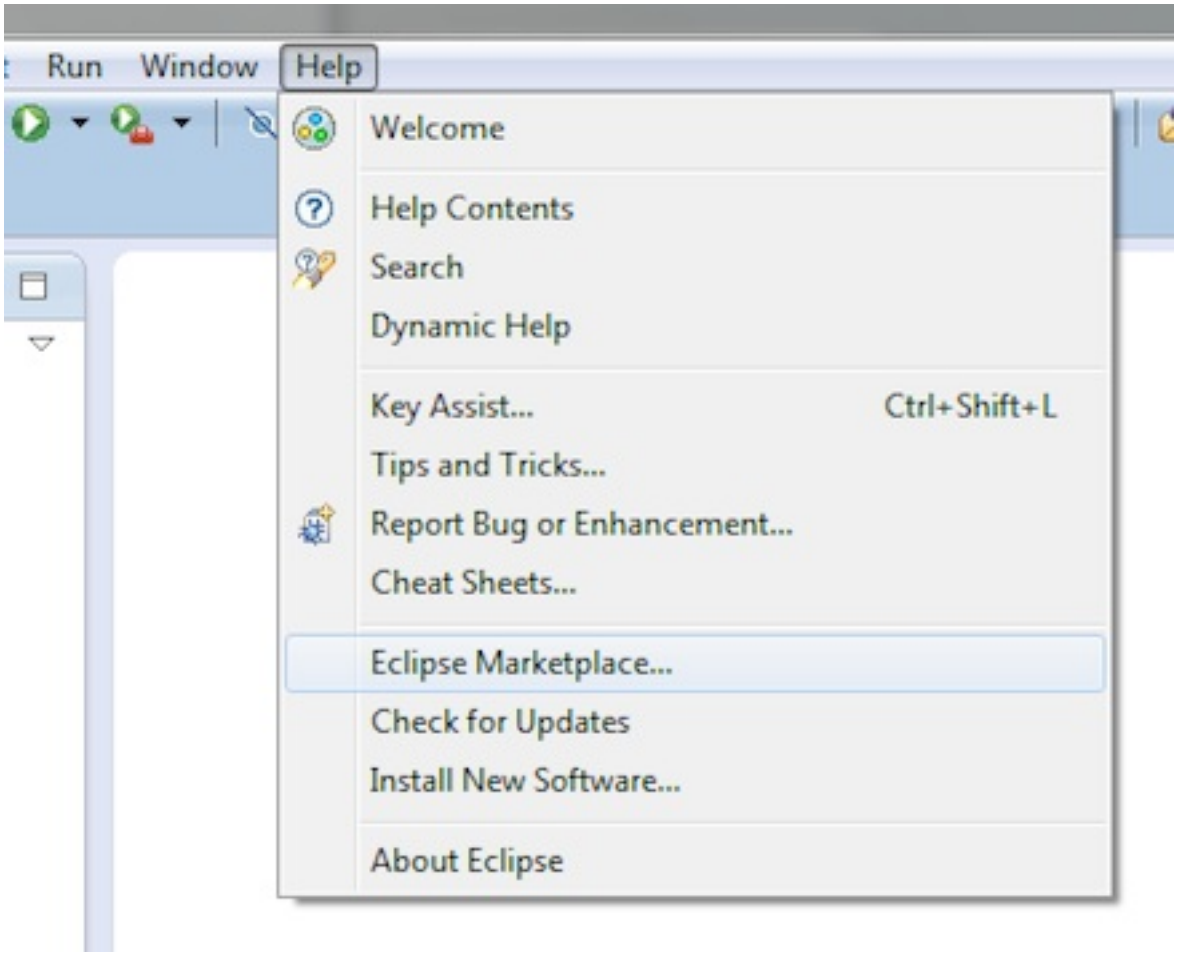

UFTasks Showcase
127.0.0.1:8888/index.html#HelloWorldScreen
UBERFIRE UI FRAMEWORK
Home
Greetings
Learn and Discover uberfire
Get Started
Documentation
Get Started with tutorials
Help us to write documentation
Fork from github
Send Pull Requests
Download uberfire
download uberfire
Linking per-type JS with 4410 new types.
prelink JS size = 8656479
prelink sourcemap = 8656479 bytes and 208075 lines
postlink JS size = 8410739
postlink sourcemap = 8410739 bytes and 203451 lines
Source Maps Enabled
Compile of permutations succeeded
Compilation succeeded -- 125.499s
Linking into /var/folders/35/8d__tw1x0wq15nzk1fjj8r4m0000gn/T/gwt-codeserver-1936433562470224814.tmp/org.uberfire.FastCompiledUFTasksShowcase/compile-2/war/org.uberfire.UFTasksShowcase; Writing e
Link succeeded
Linking succeeded -- 9.151s
136.519s total -- Compile completed
[0m14:41:47,397 INFO [org.uberfire.backend.server.plugin.RuntimePluginsServiceServerImpl] (default task-53) Not listing directory content for frameworks/*.js because getRealPath() is returning null.
[0m[0m14:41:47,449 INFO [org.uberfire.backend.server.plugin.RuntimePluginsServiceServerImpl] (default task-54) Not listing directory content for plugins/*.js because getRealPath() is returning null..
Compilation completed successfully in 16s 911ms (7 minutes ago)

Eclipse

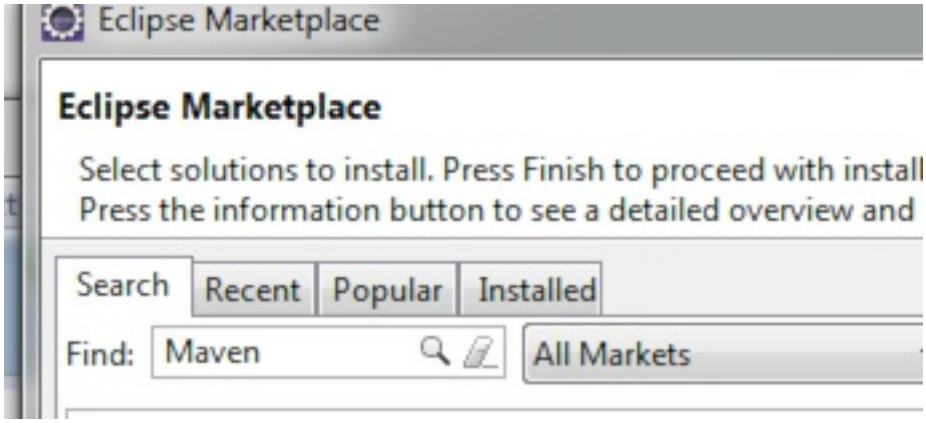
This section will walk you through using Maven tooling for running and debugging your app within Eclipse. If you have not already installed m2e in Eclipse, you will want to do so now.

To install the Maven tooling, use the following steps:

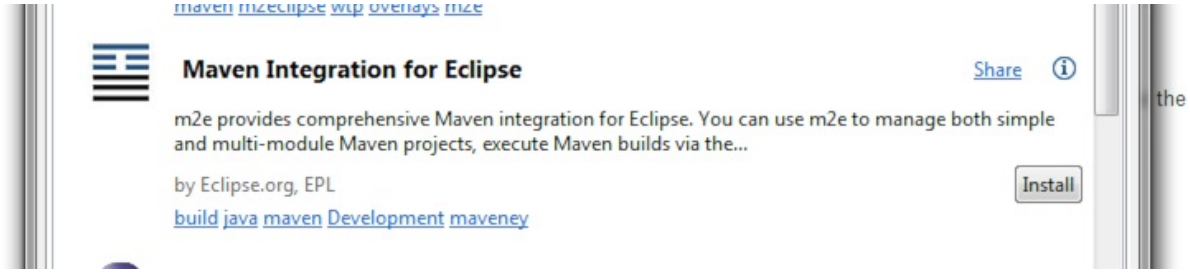
1. Go to the Eclipse Marketplace under the Help menu in Eclipse.



2. In the Find dialog enter the phrase Maven and hit enter.



3. Find the Maven Integration for Eclipse plugin and click the Install button for that entry.



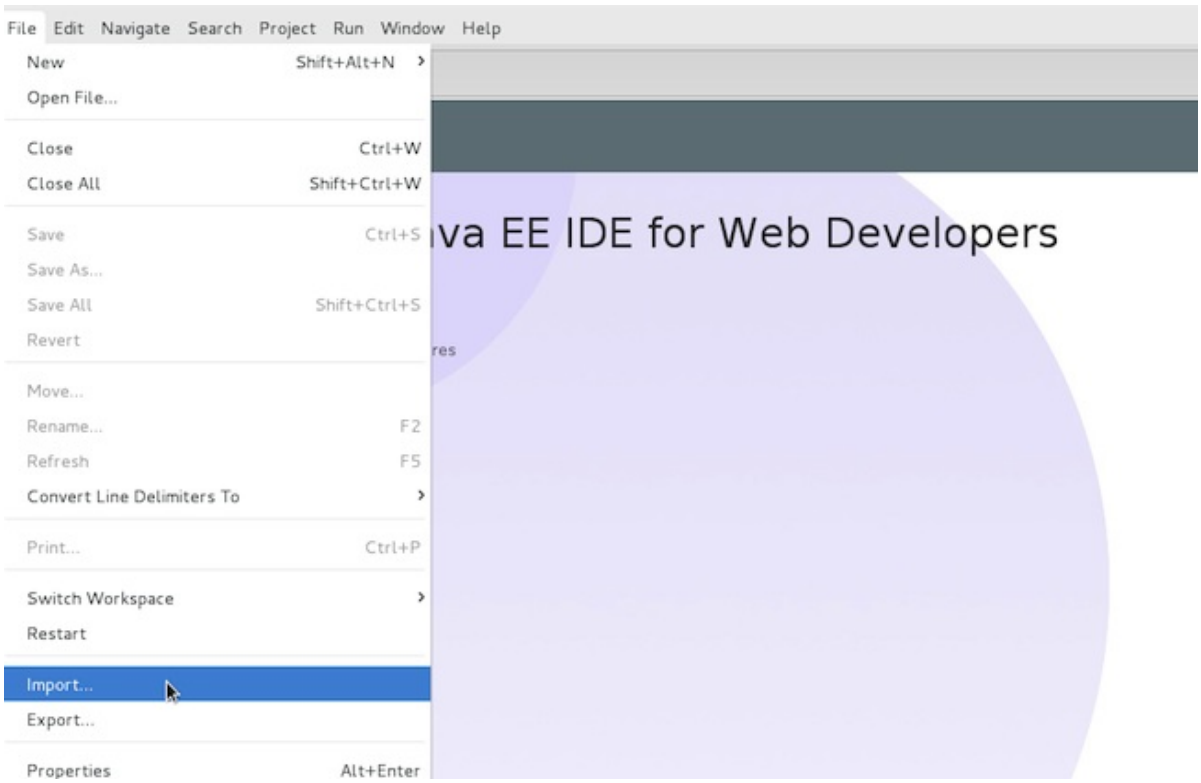
4. Accept the defaults by clicking Next , and then accept the User License Agreement to begin the installation.

Import UFtasks Project

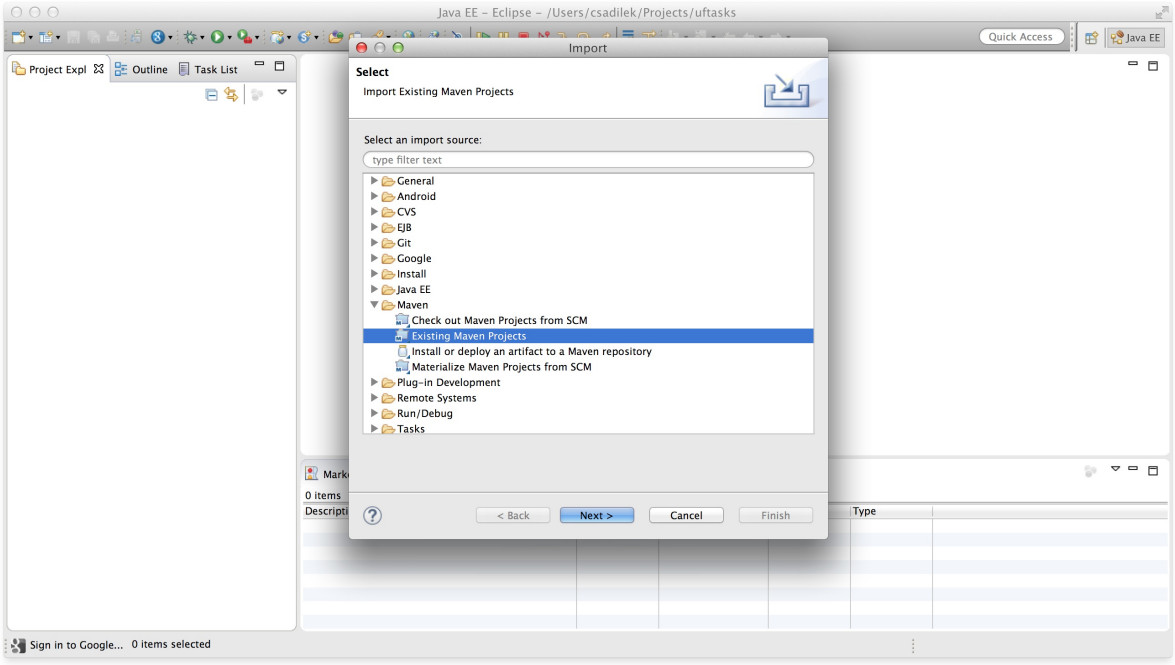
Once you have completed the installation of the prerequisites from the previous section, you will now be able to go ahead and import the Maven project you created in the first section of this guide.

Follow these steps to get the project setup:

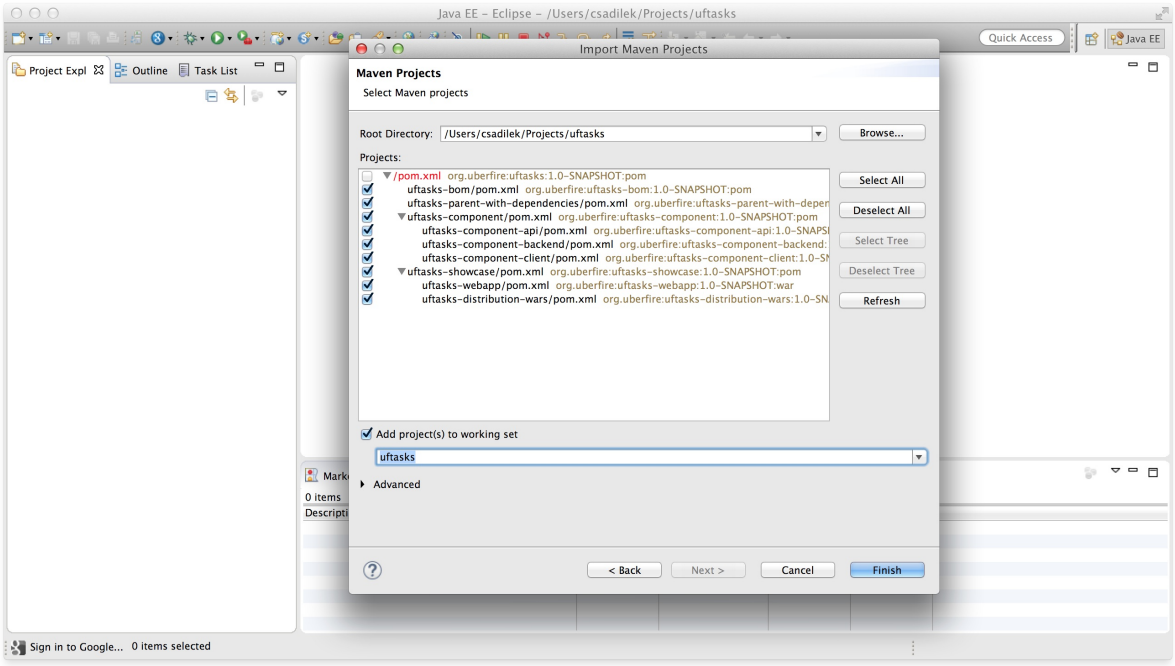
1. From the File menu, select Import…



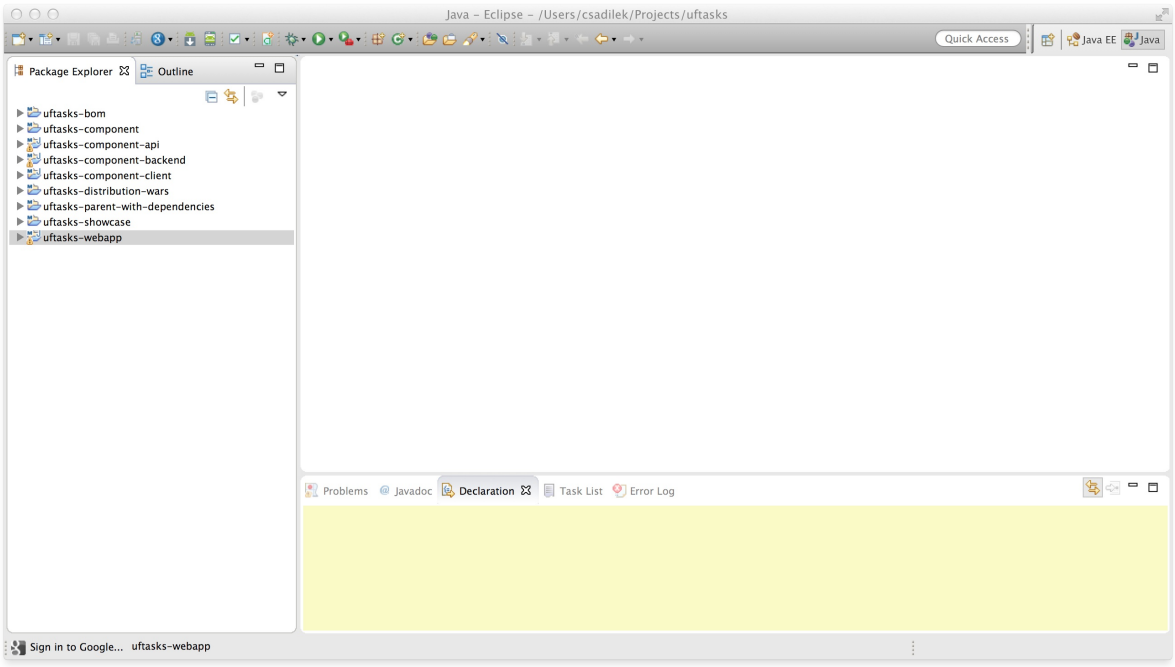
2. You will be presented with the Import dialog box. From here you want to select Maven → Existing Maven Projects



3. From the Import Maven Projects dialog, you will need to select the directory location of the project you created in the first section of this guide. In the Root Directory field of the dialog, enter the path to the project, or click Browse… to select it from the file chooser dialog.

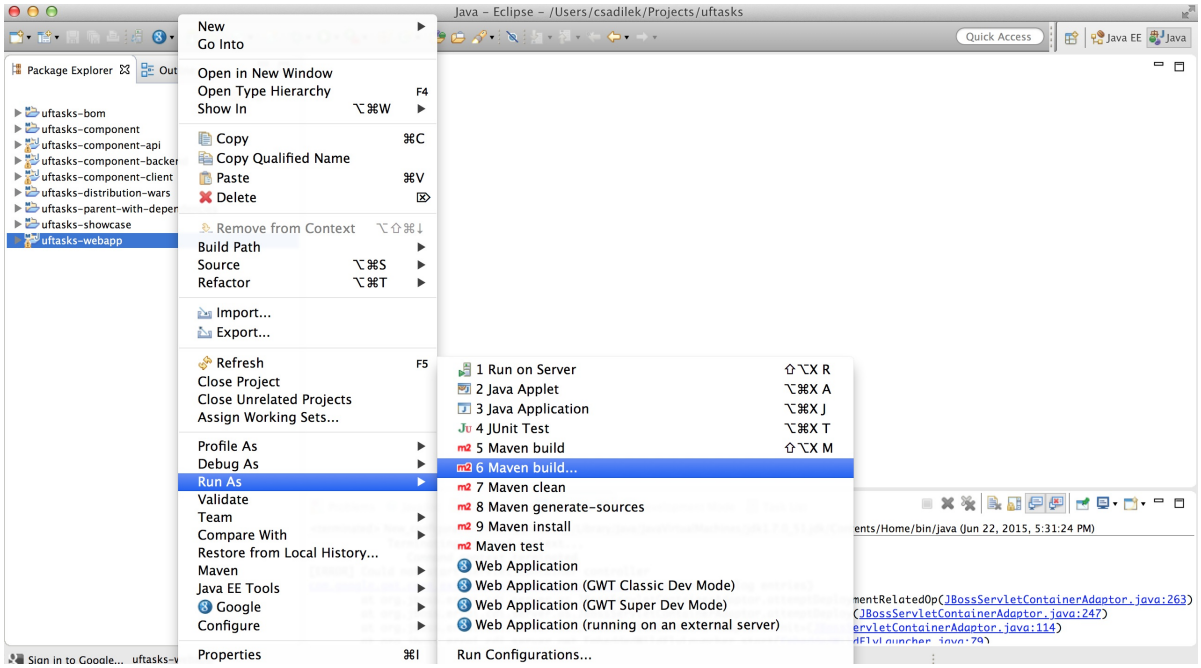


4. Click Finish to begin the import process.
5. When the import process has finished, you should see your project imported within the Eclipse Project Explorer.

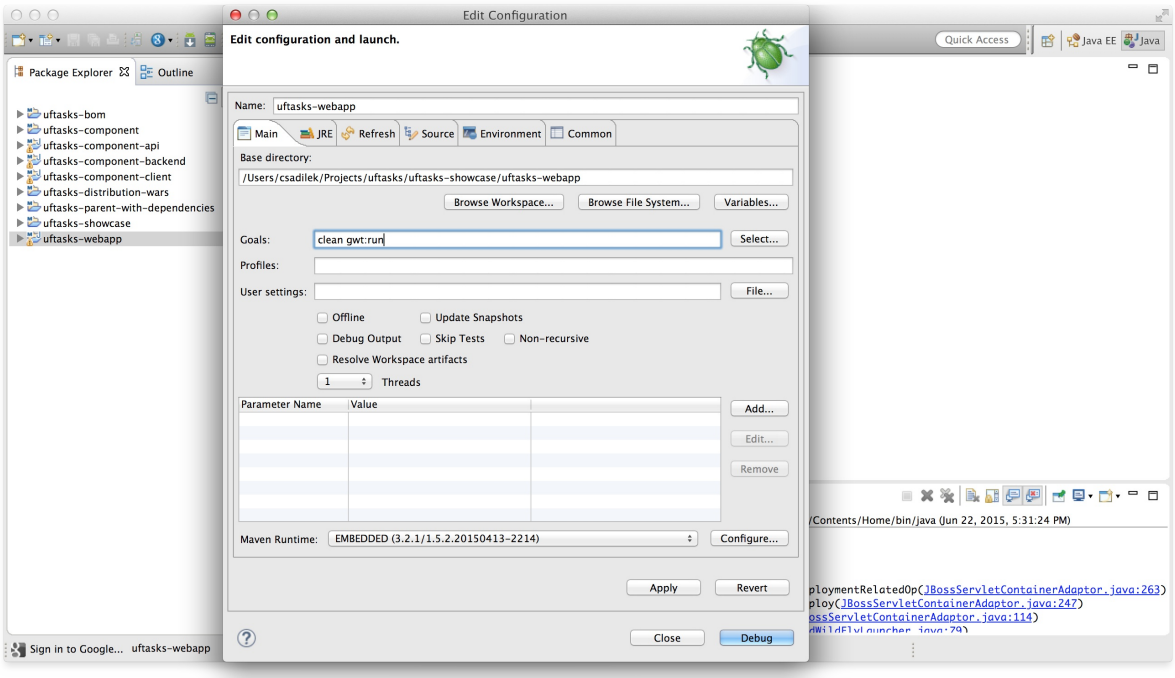


Running UFTasks with Eclipse

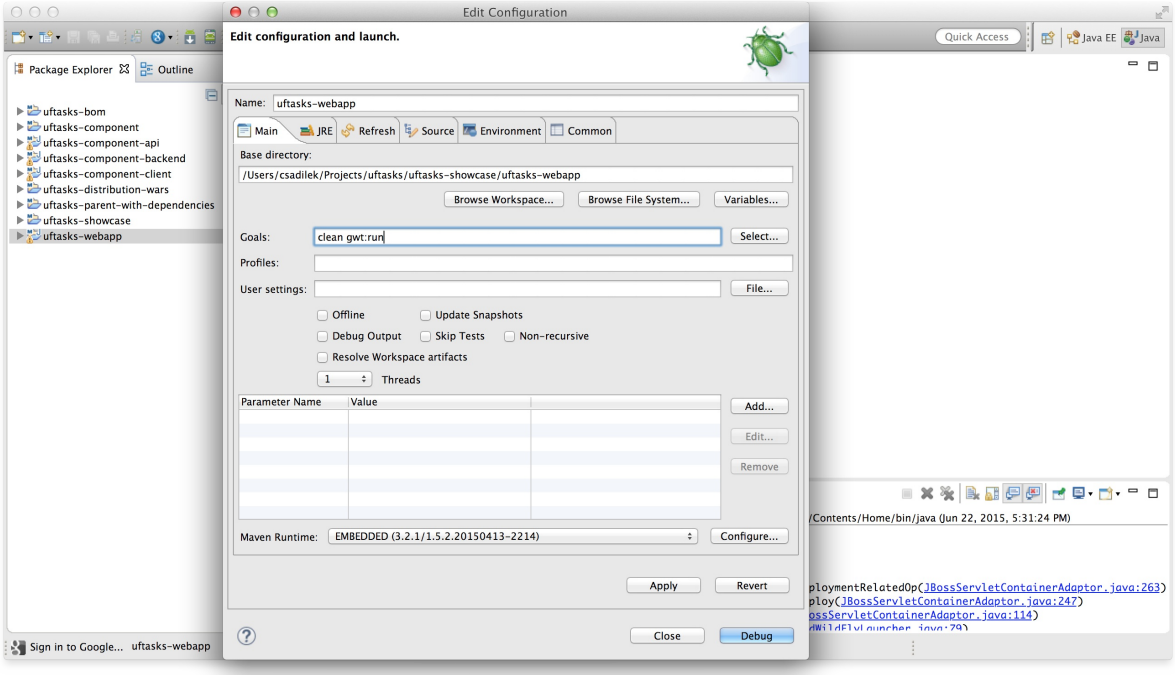
1. Next you will need to setup a Maven Run Profile for UF tasks. To do so select Run As… > Maven Build…



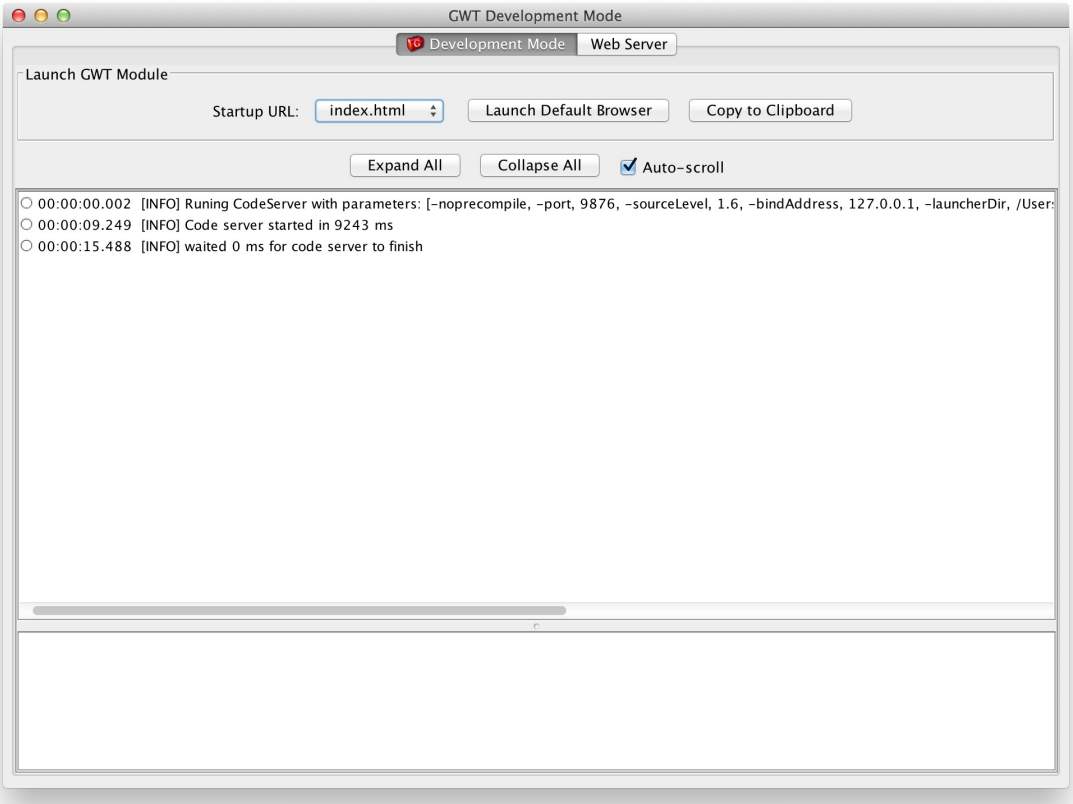
2. Edit the configuration, and your goals should be: clean gwt:run



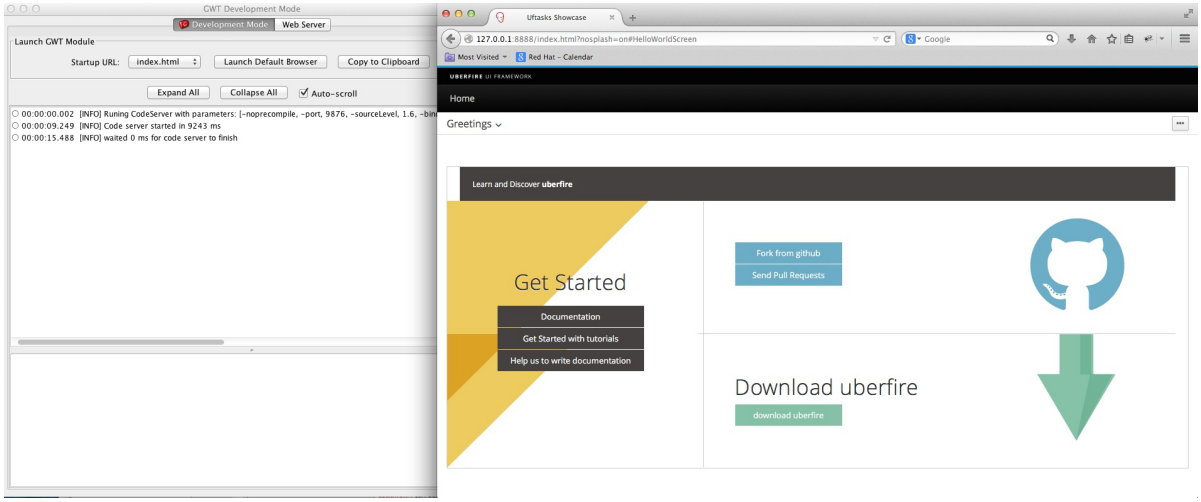
3. Now, click on Run or Debug Button:



4. Wait for GWT build and run, and then click on Launch default Browser button:

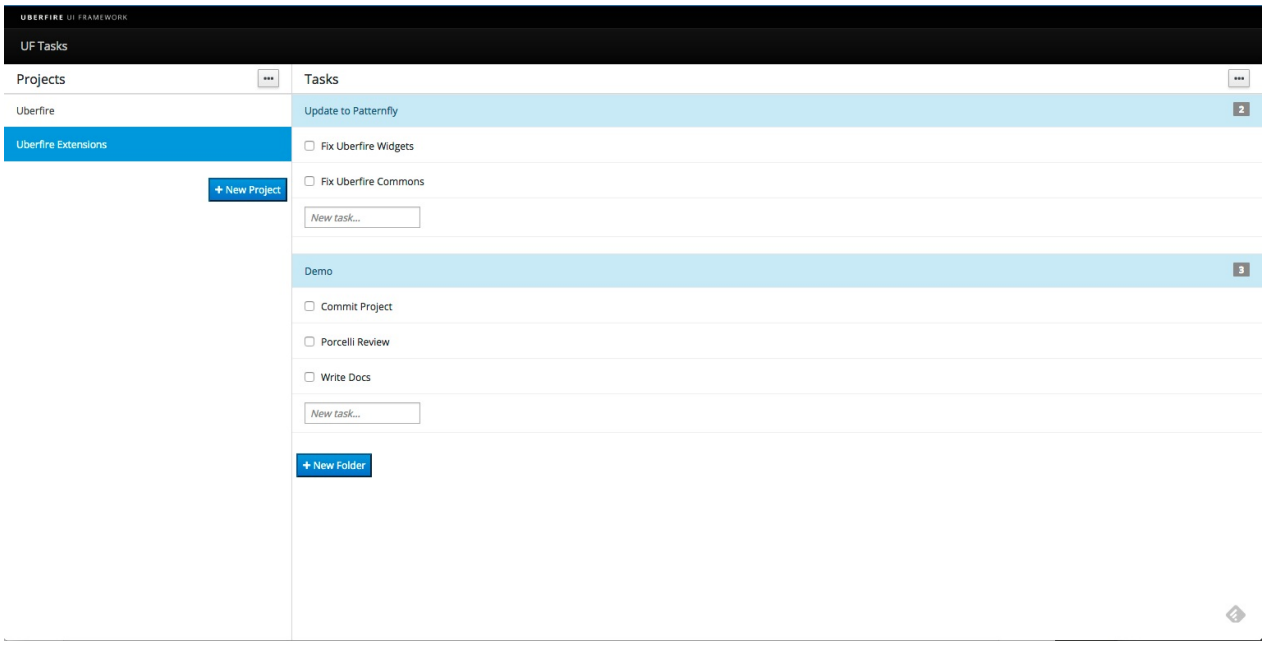


5. See the result in your browser:



UF Tasks

In order to understand how Uberfire works, let's create a simple task manager that will look like this:



On the left side, there is "Projects" section, where user can create Projects.

Each project has a group of folders (displayed in the main window), and each folder has a group of tasks.

A task can be created in the text field "New Task" and can me mark as *Done* with the checkbox in the left of task description.

Uberfire Design Guidelines

Before write some real code, it's important to get in contact with the pattern MVP, because this is how our tutorial was developed.

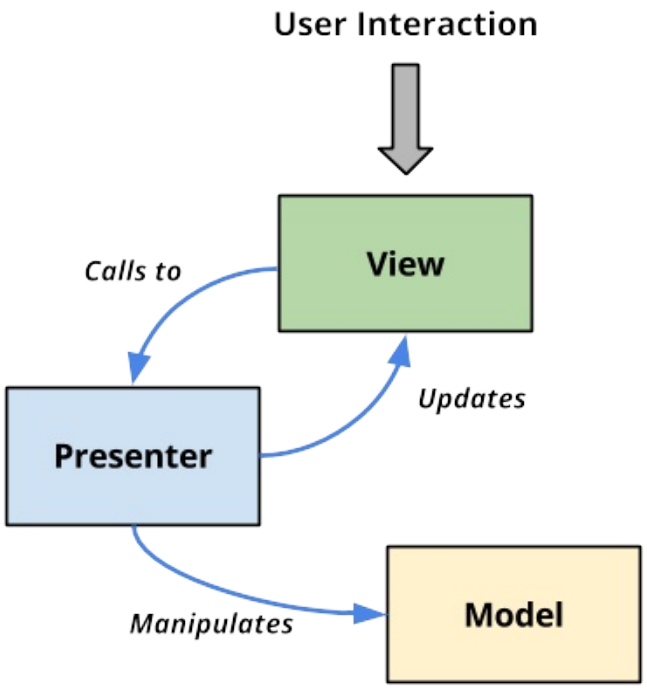
MVP Pattern

MVP (Model-View-Presenter) like any pattern is open to different interpretation, so here is what MVP was used in this tutorial:

- Model is POJO;

- View is a passive interface that that displays data, but this data cannot be the model. So it's limited to primitive types and regular platform objects (ie. String). Every user action should be routed to the presenter;

- Presenter is where all business logic should live; it acts upon the model and the view.



Model-View-Presenter

Cleaning the Archetype

Uberfire Archetype contains some useful code samples that are not necessary in our app. Let's clean up.

Please delete these files: SharedSample.java, HelloWorldScreen.java, HelloWorldScreen.ui.xml, MainPerspective.java.

Creating project structure

UberFire interfaces are made of some fundamental building blocks: Widgets, Layout Panels, Screens, Workbench Panels, Menu Bars, Tool Bars, and Perspectives. Layout

Panels can contain Widgets and other Layout Panels; Perspectives contain Workbench Panels, an optional Menu Bar, and an optional Tool Bar.

Perspectives split up the screen into multiple resizeable regions, and end users can drag and drop Panels between these regions to customize their workspace.

For now, we will create two Uberfire Screens (Project and Tasks) and one perspective to hold these screens.

Creating Projects Screen

Following the MVP pattern, each Uberfire Screen will be a Presenter plus a View. Our views will be built using UI Binder, so in that case we will have also a ui.xml file associated with each screen.

(Feel free to choose another View technology i.e. Errai UI).

Inside the package org.uberfire.client.screens, create this new source file:

- ProjectsPresenter.java

```
package org.uberfire.client.screens;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import org.uberfire.client.annotations.WorkbenchPartTitle;
import org.uberfire.client.annotations.WorkbenchPartView;
import org.uberfire.client.annotations.WorkbenchScreen;
import org.uberfire.client.mvp.UberView;

@ApplicationScoped
@WorkbenchScreen(identifier = "ProjectsPresenter")
public class ProjectsPresenter {

    public interface View extends UberView<ProjectsPresenter> {
    }

    @Inject
    private View view;

    @WorkbenchPartTitle
    public String getTitle() {
        return "Projects";
    }

    @WorkbenchPartView
    public UberView<ProjectsPresenter> getView() {
        return view;
```

```
    }

}
```

The presenter itself is a CDI bean with one injected field (the view). But whether or not we're familiar with CDI, we're seeing a bunch of Uberfire annotations for the first time. Let's examine some of them:

@WorkbenchScreen Tells UberFire that the class defines a Screen in the application. Each screen has an identifier.

@WorkbenchPartTitle Denotes the method that returns the Screen's title. Every Screen must have a @WorkbenchPartTitle method.

@WorkbenchPartView Denotes the method that returns the Panel's view. The view can be any class that extends GWT's Widget class or implements GWT's IsWidget interface. In this example, we're returning a CDI bean that implements UberView, which is the specific view, for this presenter (following MVP pattern). Every Screen must have a @WorkbenchPartView method, extend Widget, or implement IsWidget. Let's define our view:

- ProjectsView.java

```
package org.uberfire.client.screens;

import javax.annotation.PostConstruct;
import javax.enterprise.context.Dependent;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Widget;
import org.gwtbootstrap3.client.ui.Button;
import org.gwtbootstrap3.client.ui.LinkedGroup;
import org.gwtbootstrap3.client.ui.LinkedGroupItem;

@Dependent
public class ProjectsView extends Composite implements ProjectsPresenter.View {

    interface Binder
            extends
            UiBinder<Widget, ProjectsView> {

    }
```

```
    private static Binder uiBinder = GWT.create( Binder.class );

    private ProjectsPresenter presenter;


    @PostConstruct
    public void setup() {
        initWidget( uiBinder.createAndBindUi( this ) );
    }

    @Override
    public void init( ProjectsPresenter presenter ) {
        this.presenter = presenter;
    }

}
```

- ProjectsView.ui.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui"
             xmlns:b="urn:import:org.gwtbootstrap3.client.ui">

  <ui:with field='res' type='org.uberfire.client.resources.AppResource'/>
  <g:FlowPanel>
    <b:Label text="Project View"/>
  </g:FlowPanel>

</ui:UiBinder>
```

For now, this view has only a label telling "Project View".

Creating Tasks Screen

Our second screen is the Task Screen. Let's create it. (inside org.uberfire.client.screens package)

- TasksPresenter.java

```
package org.uberfire.client.screens;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import org.uberfire.client.annotations.WorkbenchPartTitle;
import org.uberfire.client.annotations.WorkbenchPartView;
import org.uberfire.client.annotations.WorkbenchScreen;
```

```
import org.uberfire.client.mvp.UberView;

@ApplicationScoped
@WorkbenchScreen(identifier = "TasksPresenter")
public class TasksPresenter {

    public interface View extends UberView<TasksPresenter> {
    }

    @Inject
    private View view;

    @WorkbenchPartTitle
    public String getTitle() {
        return "Tasks";
    }


    @WorkbenchPartView
    public UberView<TasksPresenter> getView() {
        return view;
    }
}
```

- TasksView.java

```
package org.uberfire.client.screens;

import javax.annotation.PostConstruct;
import javax.enterprise.context.Dependent;

import com.google.gwt.core.client.GWT;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Widget;

@Dependent
public class TasksView extends Composite implements TasksPresenter.View {

    private TasksPresenter presenter;

    interface Binder
            extends
            UiBinder<Widget, TasksView> {

    }

    private static Binder uiBinder = GWT.create( Binder.class );

    @Override
    public void init( final TasksPresenter presenter ) {
        this.presenter = presenter;
    }

    @PostConstruct
```

```
    public void setup() {
        initWidget( uiBinder.createAndBindUi( this ) );

    }
}
```

- TasksView.ui.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui"
             xmlns:b="urn:import:org.gwtbootstrap3.client.ui">


  <ui:with field='res' type='org.uberfire.client.resources.AppResource'/>
  <ui:style>
    .button {
      margin-top: 5px;
      margin-left: 5px;
    }
  </ui:style>
  <g:FlowPanel>
    <b:Label text="Project View"/>
  </g:FlowPanel>
</ui:UiBinder>
```

Creating Tasks Perspective

Now we have a Screen, but nowhere to put it. Remember, the UberFire workbench UI is arranged as Workbench → Perspective → Workbench Panel → Screen. Perspectives dictate the position and size of Workbench Panels. (besides the explicit positioning approach, we also can define perspectives using Errai UI templates).

We need to define a Perspective. (inside org.uberfire.client.perspectives)

- TasksPerspective.java

```
package org.uberfire.client.perspectives;

import javax.enterprise.context.ApplicationScoped;

import org.uberfire.client.annotations.Perspective;
import org.uberfire.client.annotations.WorkbenchPerspective;
import org.uberfire.client.workbench.panels.impl.MultiListWorkbenchPanelPresenter;
import org.uberfire.client.workbench.panels.impl.SimpleWorkbenchPanelPresenter;
import org.uberfire.mvp.impl.DefaultPlaceRequest;
import org.uberfire.workbench.model.CompassPosition;
```

```
import org.uberfire.workbench.model.PanelDefinition;
import org.uberfire.workbench.model.PerspectiveDefinition;
import org.uberfire.workbench.model.impl.PanelDefinitionImpl;
import org.uberfire.workbench.model.impl.PartDefinitionImpl;
import org.uberfire.workbench.model.impl.PerspectiveDefinitionImpl;

@ApplicationScoped
@WorkbenchPerspective(identifier = "TasksPerspective", isDefault = true)
public class TasksPerspective {

    @Perspective
    public PerspectiveDefinition buildPerspective() {
        final PerspectiveDefinitionImpl perspective = new PerspectiveDefinitionImpl( MultiLi
        perspective.setName( "TasksPerspective" );

        final PanelDefinition west = new PanelDefinitionImpl( SimpleWorkbenchPanelPresenter.
        west.addPart( new PartDefinitionImpl( new DefaultPlaceRequest( "ProjectsPresenter" )
        west.setWidth( 350 );
        perspective.getRoot().insertChild( CompassPosition.WEST, west );

        perspective.getRoot().addPart( new PartDefinitionImpl( new DefaultPlaceRequest( "Tas

        return perspective;
    }
}
`
```

Once again, we're encountering some new annotations:

@WorkbenchPerspective Tells UberFire that the class defines a perspective.

@Perspective Tells UberFire that this method returns the PerspectiveDefinition that governs the perspective's layout and default contents. Every @WorkbenchPerspective class needs a method annotated with @Perspective.

In this definition, we'll add a new panel on the left-hand side (WEST) and populate it with ProjectsPresenter by default. The perspective root panel (main window) will be populated with TasksPresenter.

Time to see it work!

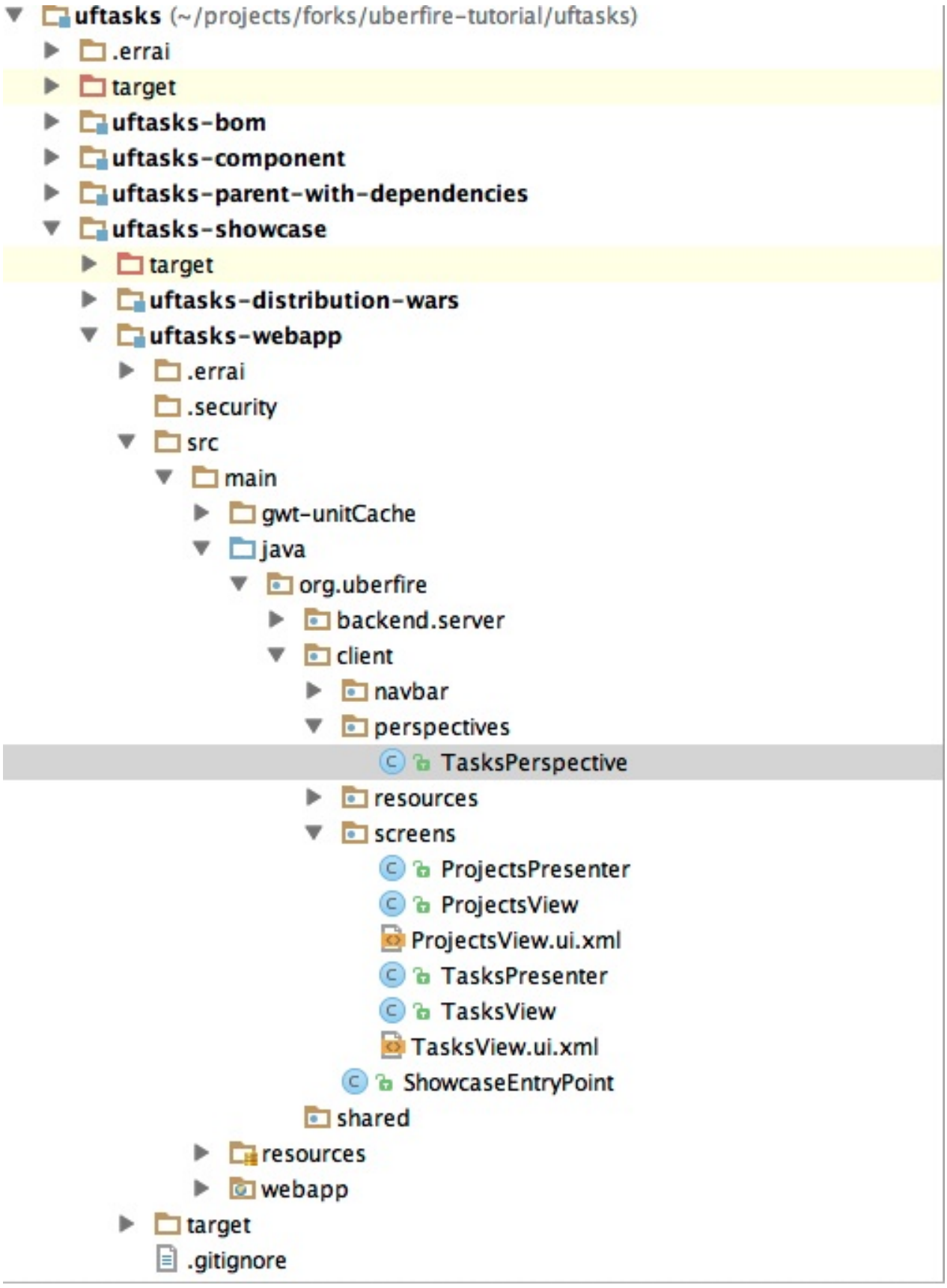
We've come a long way since we started with that empty directory. Let's reward all the hard work by starting our app and seeing it do something!

If you are using a IDE, stop the server, build and restart or if you are using command line interface:

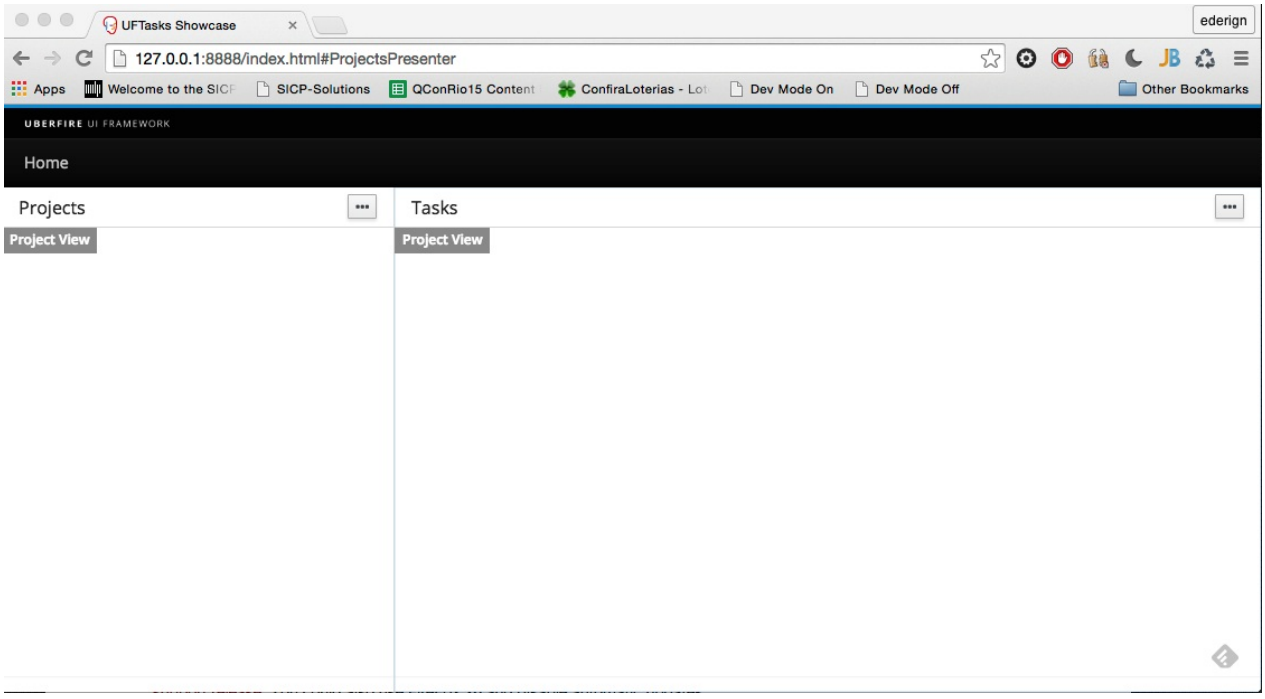
```
$ mvn compile
$ mvn gwt:run
```

Eventually, the GWT Development Mode GUI will pop up. Wait for the "Calculating…" button to change to "Launch in Default Browser," then press that button.

That should be the result of our work:



And our app running:



Not Working?

At this point, your project should be more-or-less identical to the Tutorial project at the checkpoint-1 tag. If your project isn't working at this point, grab that one and compare yours with it.

```
$ git clone https://github.com/uberfire/uberfire-tutorial.git
$ cd uberfire-tutorial
$ git checkout checkpoint-1
```

Projects Screen

Now we have the basic infrastructure required to our project, it's time to put some functionality in the Project Screen. Let's begin by the ProjectsView.

ProjectsView

Our view has two components: a Bootstrap3 LinkedGroup to list our projects and a button to create new ones.

Here's what ProjectView.ui.xml look like:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
```

```
             xmlns:g="urn:import:com.google.gwt.user.client.ui"
             xmlns:b="urn:import:org.gwtbootstrap3.client.ui">

  <ui:with field='res' type='org.uberfire.client.resources.AppResource'/>
  <ui:style>
    .button {
      float: right;
      margin-right: 5px;
    }
  </ui:style>
  <g:FlowPanel>
    <b:LinkedGroup ui:field="projects"/>
    <b:Button styleName="{style.button}" type="PRIMARY" icon="PLUS" text="New Project" ui:fi
  </g:FlowPanel>

</ui:UiBinder>
`
```

And the owner class for the above template might look like this:

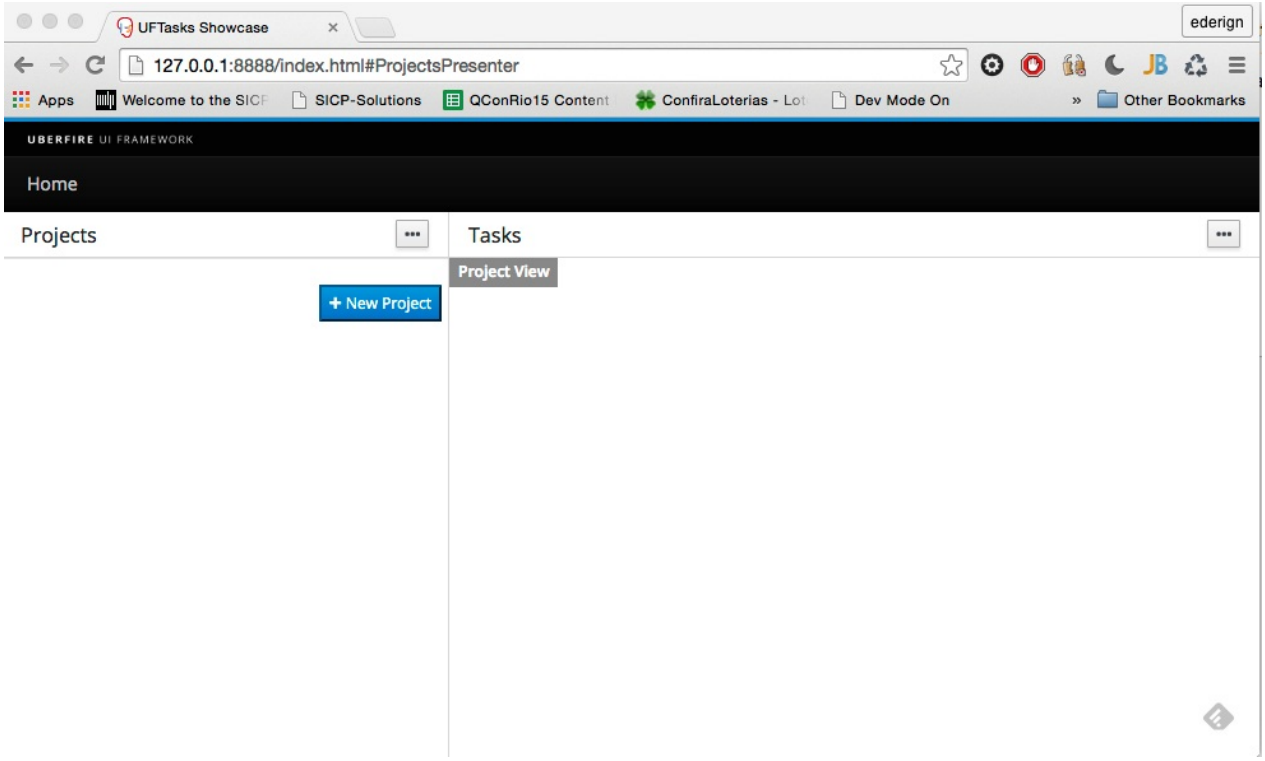
```
package org.uberfire.client.screens;

import javax.annotation.PostConstruct;
import javax.enterprise.context.Dependent;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Widget;
import org.gwtbootstrap3.client.ui.Badge;
import org.gwtbootstrap3.client.ui.Button;
import org.gwtbootstrap3.client.ui.LinkedGroup;
import org.gwtbootstrap3.client.ui.LinkedGroupItem;

@Dependent
public class ProjectsView extends Composite implements ProjectsPresenter.View {

    interface Binder
            extends
            UiBinder<Widget, ProjectsView> {

    }

    private static Binder uiBinder = GWT.create( Binder.class );

    private ProjectsPresenter presenter;

    @UiField
    Button newProject;

    @UiField
```

```
    LinkedGroup projects;

    @PostConstruct
    public void setup() {
        initWidget( uiBinder.createAndBindUi( this ) );
        newProject.addClickHandler( new ClickHandler() {
            @Override
            public void onClick( ClickEvent event ) {
                presenter.newProject();
            }
        } );
    }

    @Override
    public void init( ProjectsPresenter presenter ) {
        this.presenter = presenter;
    }

    @Override
    public void clearProjects() {
        projects.clear();
    }

    @Override
    public void addProject( final String projectName,
                            final boolean active) {
        final LinkedGroupItem projectItem = createProjectItems( projectName, active );
        projects.add( projectItem );
    }

    private LinkedGroupItem createProjectItems( final String projectName,
                                                boolean active ) {
        final LinkedGroupItem projectItem = GWT.create( LinkedGroupItem.class );
        projectItem.setText( projectName );
        projectItem.setActive( active );
        projectItem.addClickHandler( new ClickHandler() {
            @Override
            public void onClick( ClickEvent event ) {
                presenter.selectProject( projectName );
            }
        } );
        return projectItem;
    }

}
```

Two @UiFields to bind the template with the view Java class, an click handler to "New Project" button and a method to addAProject. This method should receive as parameter the projectName and a boolean representing if the project is active in the screen.

Time to see our view working.

Refresh the browser, let GWT Super Dev mode make his magic, and click on New Project button.



New Project Screen

The next step of our project is to provide a real implementation to new project button. In order to achieve this, our view needs to tell the presenter that the button is clicked. In order to achieve that, change the implementation of new project click handler on ProjectsView.java:

```
@PostConstruct
  public void setup() {
      initWidget( uiBinder.createAndBindUi( this ) );
      newProject.addClickHandler( new ClickHandler() {
          @Override
          public void onClick( ClickEvent event ) {
              presenter.newProject();
          }
      } );
  }
```

And create this method on ProjectsPresenter.java

```
  public void newProject() {
      //TODO
  }
```

This method will call another presenter, in order to display the popup (modal) and ask user for new project name. To achieve this, let's create these classes on

org.uberfire.client.screens.popup package:

NewProjectPresenter.java

```
package org.uberfire.client.screens.popup;

import javax.annotation.PostConstruct;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

import org.uberfire.client.mvp.UberView;
import org.uberfire.client.screens.ProjectsPresenter;

@Dependent
public class NewProjectPresenter {

    private ProjectsPresenter projectsPresenter;

    public interface View extends UberView<NewProjectPresenter> {
        void show();
        void hide();
    }

    @Inject
    private View view;

    @PostConstruct
    public void setup(){
        view.init( this );
    }

    public void show( ProjectsPresenter projectsPresenter ){
        this.projectsPresenter = projectsPresenter;
        view.show();
    }

    public void newProject( String projectName ) {
        projectsPresenter.createNewProject( projectName );
        view.hide();
    }

    public void close() {
        view.hide();
    }

}
```

The method show(projectsPresenter) will open the modal on view. The method newProject(projectName) will create a new project on projectsPresenter and hide the view.

NewProjectView.java

```
package org.uberfire.client.screens.popup;

import javax.annotation.PostConstruct;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Widget;
import org.gwtbootstrap3.client.ui.Button;
import org.gwtbootstrap3.client.ui.Modal;
import org.gwtbootstrap3.client.ui.TextBox;

public class NewProjectView extends Composite
        implements NewProjectPresenter.View {

    interface Binder
            extends
            UiBinder<Widget, NewProjectView> {

    }

    private static Binder uiBinder = GWT.create( Binder.class );

    @UiField
    Modal popup;

    @UiField
    TextBox projectName;

    @UiField
    Button addGroup;

    @UiField
    Button cancel;

    private NewProjectPresenter presenter;


    @Override
    public void init( NewProjectPresenter presenter ) {
        this.presenter = presenter;
    }

    @PostConstruct
    public void setup() {
        initWidget( uiBinder.createAndBindUi( this ) );
        cancel.addClickHandler( new ClickHandler() {
            @Override
            public void onClick( ClickEvent event ) {
                presenter.close();
            }
        } );
        addGroup.addClickHandler( new ClickHandler() {
            @Override
            public void onClick( ClickEvent event ) {
```

```
                presenter.newProject( projectName.getText() );
            }
        } );
    }

    @Override
    public void show() {
        popup.show();
    }

    @Override
    public void hide() {
        popup.hide();
        projectName.setText( "" );
    }

}
```

NewProjectView.ui.xml

```
<!--
  ~ Copyright 2012 JBoss Inc
  ~
  ~ Licensed under the Apache License, Version 2.0 (the "License");
  ~ you may not use this file except in compliance with the License.
  ~ You may obtain a copy of the License at
  ~
  ~       http://www.apache.org/licenses/LICENSE-2.0
  ~
  ~ Unless required by applicable law or agreed to in writing, software
  ~ distributed under the License is distributed on an "AS IS" BASIS,
  ~ WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  ~ See the License for the specific language governing permissions and
  ~ limitations under the License.
  -->

<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui"
             xmlns:b="urn:import:org.gwtbootstrap3.client.ui">

  <g:FlowPanel>
    <b:Modal ui:field="popup">
      <b:ModalBody>
        <b:Form>
          <b:FieldSet>
                 <b:Legend>New Project</b:Legend>
            <b:FormGroup>
              <b:FormLabel for="name">Name</b:FormLabel>
              <b:TextBox placeholder="Project name" ui:field="projectName"/>
            </b:FormGroup>
          </b:FieldSet>
        </b:Form>
      </b:ModalBody>
      <b:ModalFooter>
```

```
        <b:Button type="DANGER" ui:field="cancel">Cancel</b:Button>
        <b:Button type="PRIMARY" ui:field="addGroup">OK</b:Button>
      </b:ModalFooter>
    </b:Modal>
  </g:FlowPanel>
</ui:UiBinder>
```

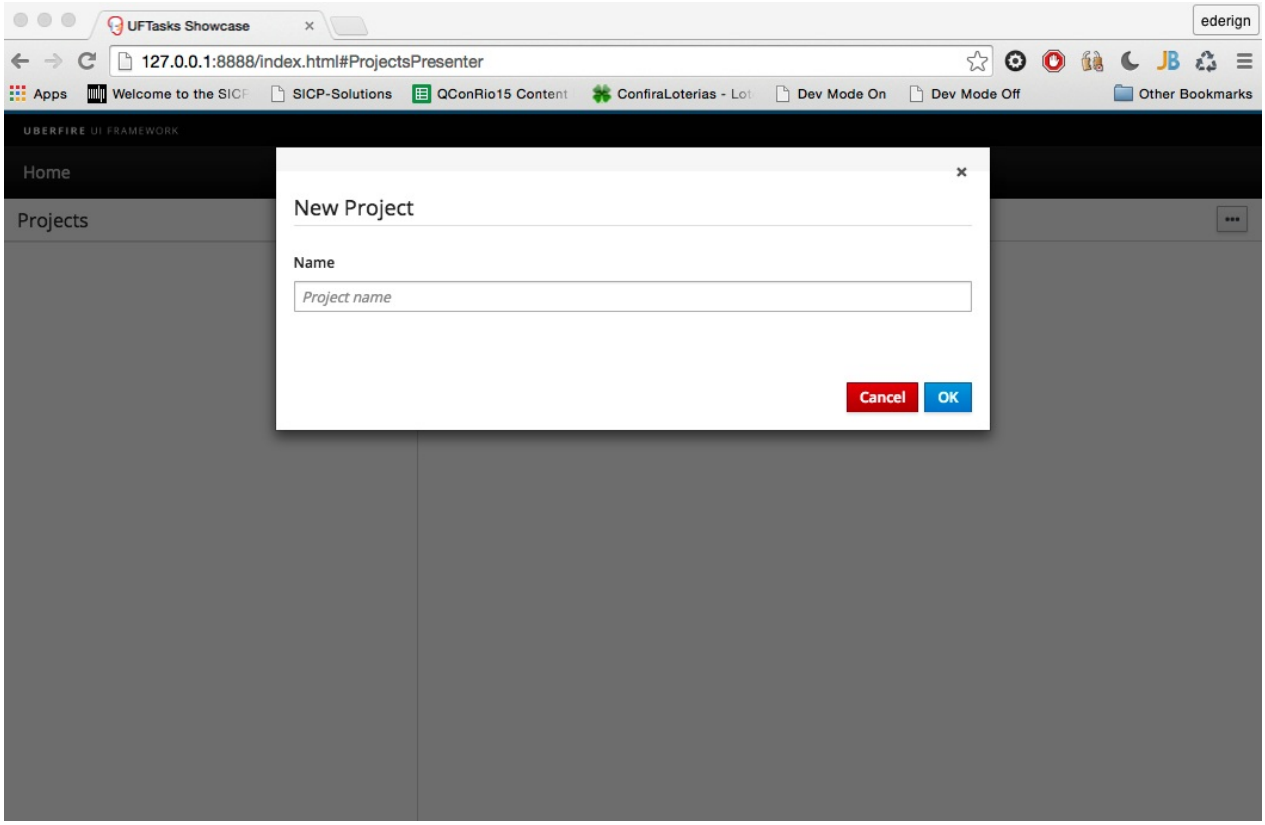
We have also to change **ProjectPresenter.java** in order to open the popup and receive the name of new project created. And this snippet to our class.

```
    @Inject
    private NewProjectPresenter newProjectPresenter;

    public void newProject() {
        newProjectPresenter.show( this );
    }

    public void createNewProject( String projectName ) {
        Window.alert( "project created" + projectName );
    }
```

Time to see it work!

Let’s see all the hard work by starting our app and seeing it opening the popup and displaying the project created message!

Refresh your browser and click on new project button.



Adding projects to project's list

Our next task is to finally add a project to projects list. First of all, let's create the Project model class. In package org.uberfire.share.model, create this class:

Project.java

```
package org.uberfire.shared.model;

public class Project {

    private final String name;
    private boolean selected;

    public Project( String name ) {
        this.name = name;
        this.selected = false;
    }

    public String getName() {
        return name;
    }

    public boolean isSelected() {
        return selected;
    }

    public void setSelected( boolean selected ) {
        this.selected = selected;
    }
```

```
}
```

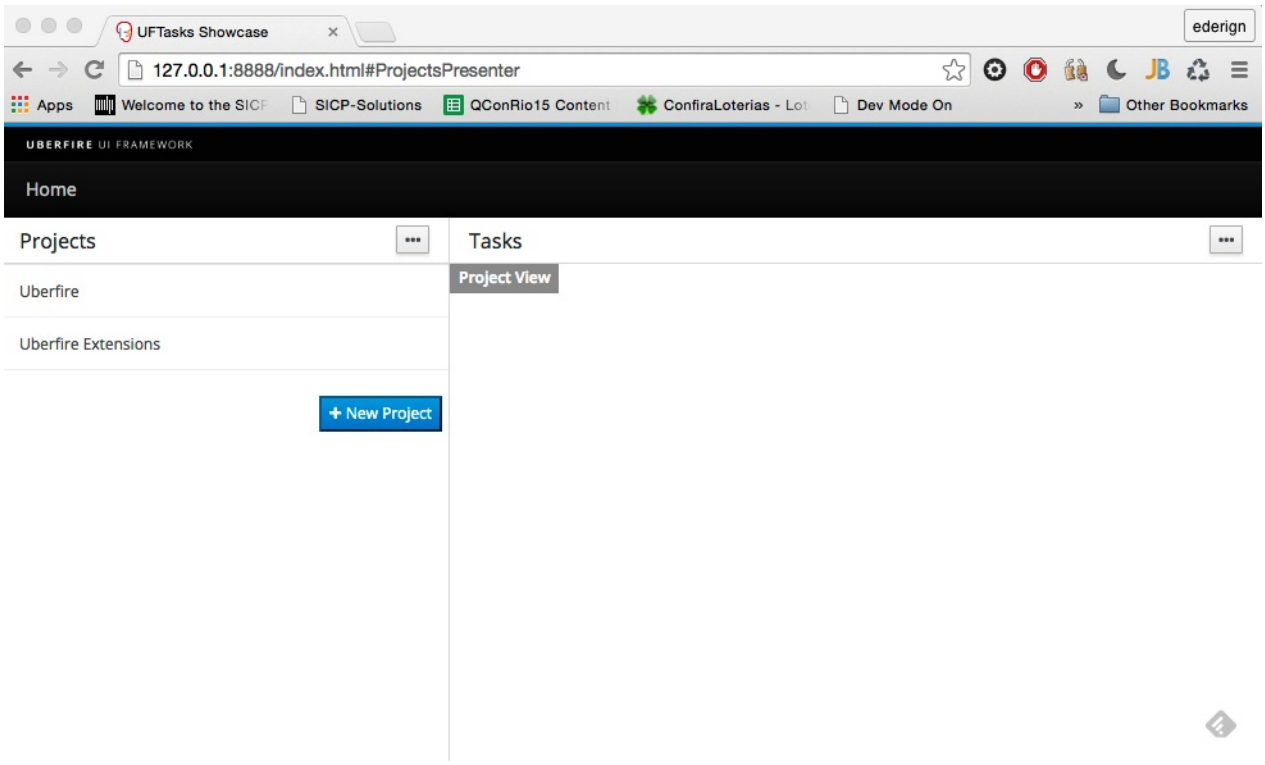
Next, when the method createNewProject(project) is called, let's create a new model for that project and update the view. Add the following code to ProjectsPresenter.java:

ProjectsPresenter.java

```
private List<Project> projects = new ArrayList<Project>();

    public void createNewProject( String projectName ) {
        projects.add( new Project( projectName ) );
        updateView();
    }

    private void updateView() {
        view.clearProjects();
        for ( Project project : projects ) {
            view.addProject( project.getName(), project.isSelected() );
        }
    }
```

Time to see it work!

Refresh the browser, and add two projects to our list.



Creating tasks list

It's time to create our task list. When our user creates or select a project in project's list, we should display the folders and tasks associated with that project.

In order to do that, let's change our **ProjectsView.java** to notify the presenter that a project is selected. Please change createProjectItems method:

```
private LinkedGroupItem createProjectItems( final String projectName,
                                            boolean active ) {
        final LinkedGroupItem projectItem = GWT.create( LinkedGroupItem.class );
        projectItem.setText( projectName );
        projectItem.setActive( active );
        projectItem.addClickHandler( new ClickHandler() {
            @Override
            public void onClick( ClickEvent event ) {
                presenter.selectProject( projectName );
            }
        } );
        return projectItem;
    }
```

Create inside org.uberfire.shared.events, a model class called ProjectSelectedEvent, to allow communication between our screens:

ProjectSelectedEvent.java

```
package org.uberfire.shared.events;

public class ProjectSelectedEvent {

    private final String name;

    public ProjectSelectedEvent( String name ) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

And also create these methods on **ProjectsPresenter.java**:

```
    @Inject
```

```
    private Event<ProjectSelectedEvent> projectSelectedEvent;

    public void selectProject( String projectName ) {
        setActiveProject( projectName );
        projectSelectedEvent.fire( new ProjectSelectedEvent( projectName ) );
    }

    private void setActiveProject( String projectName ) {
        for ( Project project : projects ) {
            if ( projectName.equalsIgnoreCase( project.getName() ) ) {
                project.setSelected( true );
            } else {
                project.setSelected( false );
            }
        }
        updateView();
    }
```

Let's detail the two important calls that happens inside selectProject(projectName) method:

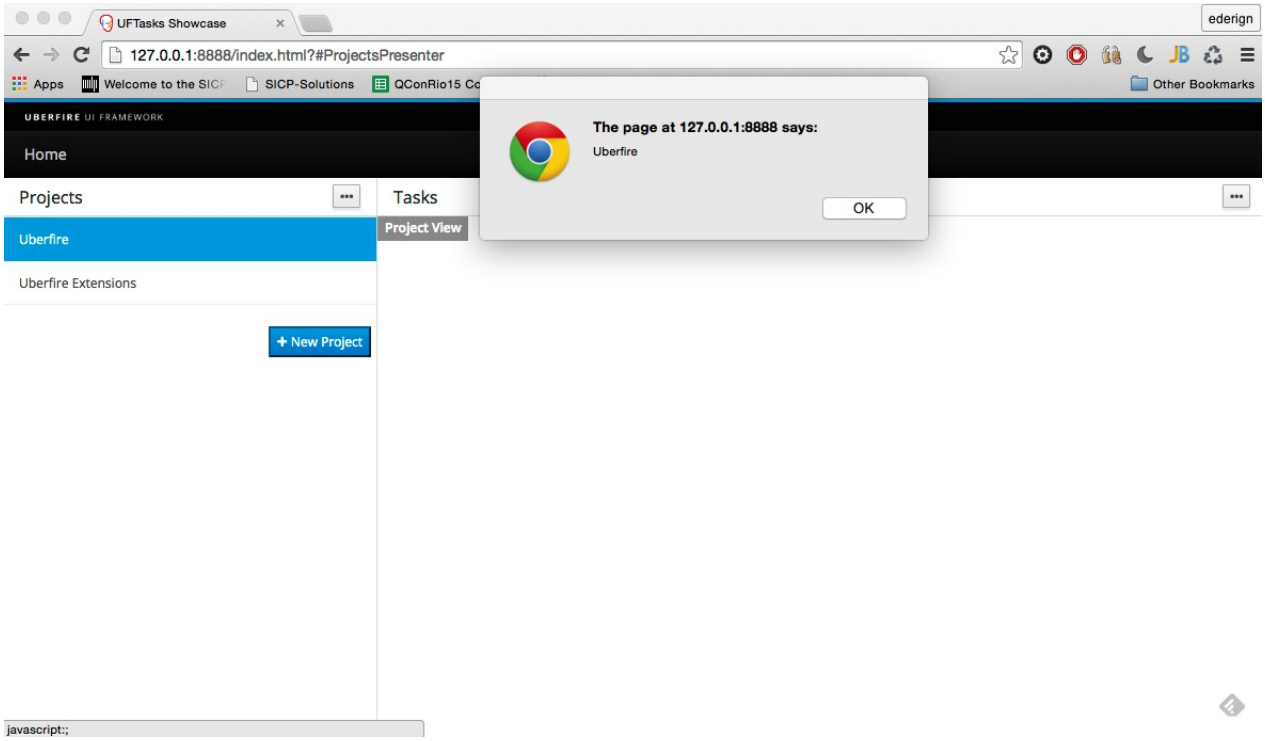
- setActiveProject(projectName): mark a specific project as active and update the view;
- projectSelectEvent.fire(...): fires a cdi event telling TasksPresenter.java that a project was selected

Now, let's listen this cdi event on **TaskPresenter.java**. Add the following method:

```
    public void projectSelected( @Observes ProjectSelectedEvent projectSelectedEvent ) {
        Window.alert( projectSelectedEvent.getName() );
    }
```

Time to see it work!

Refresh the browser, create two projects and click in one of them.



New folder

Our next step is to create new folder button. A folder is an aggregator of tasks. This button is only displayed after a project is selected. Let's create the view:

TasksView.ui.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui"
             xmlns:b="urn:import:org.gwtbootstrap3.client.ui">


  <ui:with field='res' type='org.uberfire.client.resources.AppResource'/>
  <ui:style>
    .button {
      margin-top: 5px;
      margin-left: 5px;
    }
  </ui:style>
  <g:FlowPanel>
    <g:FlowPanel ui:field="tasks"/>
    <b:Button styleName="{style.button}" type="PRIMARY" icon="PLUS" text="New Folder" ui:fie
  </g:FlowPanel>
</ui:UiBinder>
```

Update your **TaskPresenter.java** with the following code:

```
package org.uberfire.client.screens;

import java.util.ArrayList;
import java.util.List;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;
import javax.inject.Inject;

import org.uberfire.client.annotations.WorkbenchPartTitle;
import org.uberfire.client.annotations.WorkbenchPartView;
import org.uberfire.client.annotations.WorkbenchScreen;
import org.uberfire.client.mvp.UberView;
import org.uberfire.client.screens.popup.NewFolderPresenter;
import org.uberfire.shared.events.ProjectSelectedEvent;

@ApplicationScoped
@WorkbenchScreen(identifier = "TasksPresenter")
public class TasksPresenter {

    public interface View extends UberView<TasksPresenter> {

        void activateNewFolder();

        void clearTasks();

        void newFolder( String name,
                        Integer size,
                        List<String> tasks );
    }

    @Inject
    private View view;

    private String currentSelectedProject;

    @Inject
    private NewFolderPresenter newFolderPresenter;

    @WorkbenchPartTitle
    public String getTitle() {
        return "Tasks";
    }

    @WorkbenchPartView
    public UberView<TasksPresenter> getView() {
        return view;
    }

    public void projectSelected( @Observes ProjectSelectedEvent projectSelectedEvent ) {
        this.currentSelectedProject = projectSelectedEvent.getName();
        selectFolder();
    }

    private void selectFolder() {
        view.activateNewFolder();
        updateView( null );
    }
```

```
    public void showNewFolder() {
        newFolderPresenter.show( this );
    }

    private void updateView( String folderName ) {
        view.clearTasks();
        if(folderName!=null){
            view.newFolder( folderName, 0, new ArrayList<String>() );
        }
    }

    public void newFolder( String folderName ) {
        updateView( folderName );
    }

}
```

Pay attention to showNewFolder() method. This opens a popup to ask for folder name. The popup structure is like NewProject* structure.Let's create it inside package org.uberfire.client.screens.popup:

NewFolderPresenter.java

```
package org.uberfire.client.screens.popup;

import javax.annotation.PostConstruct;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

import org.uberfire.client.mvp.UberView;
import org.uberfire.client.screens.TasksPresenter;

@Dependent
public class NewFolderPresenter {

    public interface View extends UberView<NewFolderPresenter> {
        void show();
        void hide();
    }

    @Inject
    private View view;

    private TasksPresenter tasksPresenter;

    @PostConstruct
    public void setup() {
        view.init( this );
    }

    public void show( TasksPresenter tasksPresenter ) {
        this.tasksPresenter = tasksPresenter;
        view.show();
    }
```

```
    public void newFolder( String folderName ) {
        tasksPresenter.newFolder( folderName );
        view.hide();
    }

    public void close() {
        view.hide();
    }

}
```

NewFolderView.java

```
package org.uberfire.client.screens.popup;

import javax.annotation.PostConstruct;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Widget;
import org.gwtbootstrap3.client.ui.Button;
import org.gwtbootstrap3.client.ui.Modal;
import org.gwtbootstrap3.client.ui.TextBox;

public class NewFolderView extends Composite
        implements NewFolderPresenter.View {

    interface Binder
            extends
            UiBinder<Widget, NewFolderView> {
    }

    private static Binder uiBinder = GWT.create( Binder.class );

    @UiField
    Modal popup;

    @UiField
    TextBox folderName;

    @UiField
    Button addFolder;

    @UiField
    Button cancel;

    private NewFolderPresenter presenter;

    @Override
    public void init( NewFolderPresenter presenter ) {
```

```
        this.presenter = presenter;
    }

    @PostConstruct
    public void setup() {
        initWidget( uiBinder.createAndBindUi( this ) );
        cancel.addClickHandler( new ClickHandler() {
            @Override
            public void onClick( ClickEvent event ) {
                presenter.close();
            }
        } );
        addFolder.addClickHandler( new ClickHandler() {
            @Override
            public void onClick( ClickEvent event ) {
                presenter.newFolder( folderName.getText() );
            }
        } );
    }

    @Override
    public void show() {
        popup.show();
    }

    @Override
    public void hide() {
        popup.hide();
        folderName.setText( "" );
    }

}
```

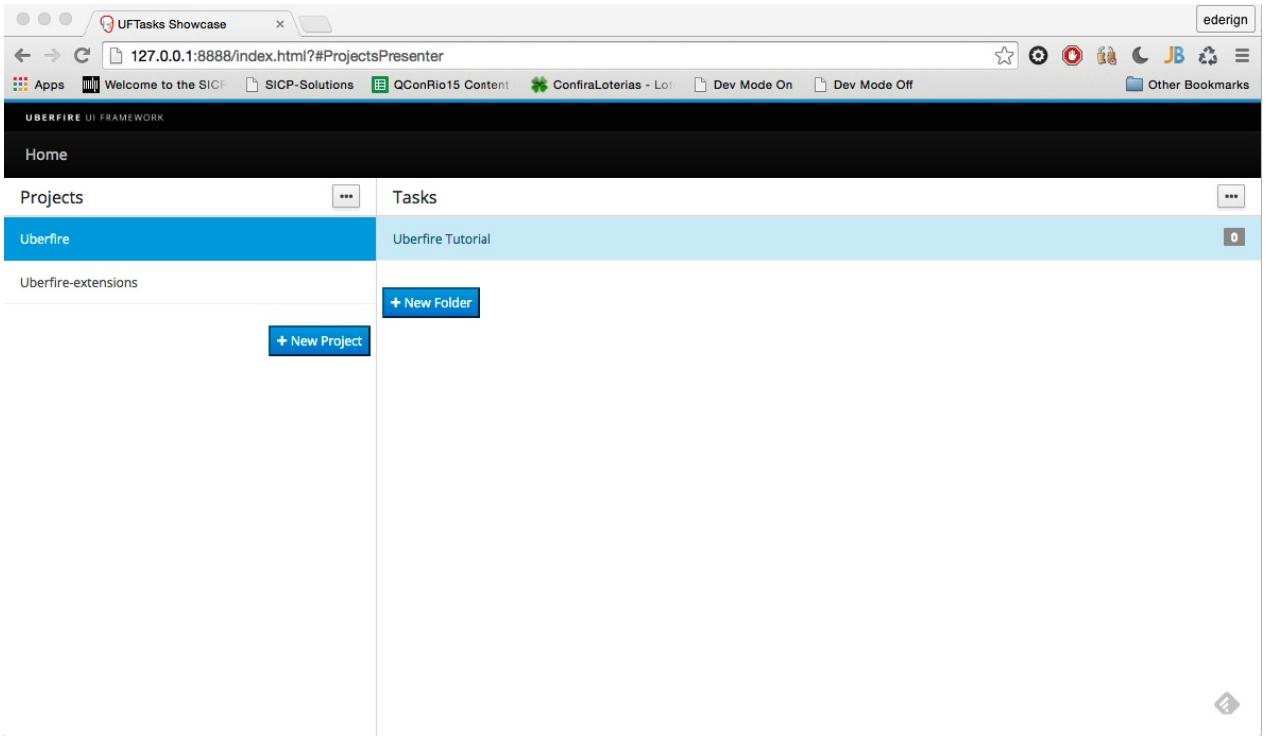
NewFolderView.ui.xml

```
<!--
  ~ Copyright 2012 JBoss Inc
  ~
  ~ Licensed under the Apache License, Version 2.0 (the "License");
  ~ you may not use this file except in compliance with the License.
  ~ You may obtain a copy of the License at
  ~
  ~       http://www.apache.org/licenses/LICENSE-2.0
  ~
  ~ Unless required by applicable law or agreed to in writing, software
  ~ distributed under the License is distributed on an "AS IS" BASIS,
  ~ WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  ~ See the License for the specific language governing permissions and
  ~ limitations under the License.
  -->

<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui"
             xmlns:b="urn:import:org.gwtbootstrap3.client.ui">
```

```
    <g:FlowPanel>
      <b:Modal ui:field="popup">
        <b:ModalBody>
          <b:Form>
            <b:FieldSet>
                  <b:Legend>New Folder</b:Legend>
              <b:FormGroup>
                <b:FormLabel for="name">Name</b:FormLabel>
                <b:TextBox placeholder="Folder name" ui:field="folderName"/>
              </b:FormGroup>
            </b:FieldSet>
          </b:Form>
        </b:ModalBody>
        <b:ModalFooter>
          <b:Button type="DANGER" ui:field="cancel">Cancel</b:Button>
          <b:Button type="PRIMARY" ui:field="addFolder">OK</b:Button>
        </b:ModalFooter>
      </b:Modal>
    </g:FlowPanel>
  </ui:UiBinder>
```

Time to see it work!

Refresh the browser, create two projects and click in one of them. Create a new folder and the result will be something like:



Not Working?

At this point, your project should be more-or-less identical to the Tutorial project at the checkpoint-2 tag. If your project isn't working at this point, grab that one and compare

yours with it.

```
$ git clone https://github.com/uberfire/uberfire-tutorial.git
$ cd uberfire-tutorial
$ git checkout checkpoint-2
```

Time to add some tasks

It's time to add some tasks to our project. At this point, we believe that you already have knowledge in basic Uberfire concepts. So we will comment quickly the changes necessary to implement the tasks support to our project. Let's begin creating a model class (package org.uberfire.shared.model):

Folder.java

```
package org.uberfire.shared.model;

import java.util.ArrayList;
import java.util.List;

public class Folder {

    private final String name;

    private List<String> tasks = new ArrayList<String>();

    public Folder( String name ) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public List<String> getTasks() {
        return tasks;
    }

    public void addTask( String task ) {
        tasks.add( task );
    }

    public void removeTask( String taskText ) {
        tasks.remove( taskText );
    }
}
```

Now update the **TasksPresenter.java** in order to support the creation of tasks.

```
package org.uberfire.client.screens;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;
import javax.inject.Inject;

import org.uberfire.client.annotations.WorkbenchPartTitle;
import org.uberfire.client.annotations.WorkbenchPartView;
import org.uberfire.client.annotations.WorkbenchScreen;
import org.uberfire.client.mvp.UberView;
import org.uberfire.client.screens.popup.NewFolderPresenter;
import org.uberfire.shared.events.ProjectSelectedEvent;
import org.uberfire.shared.model.Folder;

@ApplicationScoped
@WorkbenchScreen(identifier = "TasksPresenter")
public class TasksPresenter {

    public interface View extends UberView<TasksPresenter> {

        void activateNewFolder();

        void clearTasks();

        void newFolder( String name,
                        Integer size,
                        List<String> strings );
    }

    @Inject
    private View view;

    @Inject
    private NewFolderPresenter newFolderPresenter;

    private String currentSelectedProject;

    private Map<String, List<Folder>> foldersPerProject = new HashMap<String, List<Folder>>(

    @WorkbenchPartTitle
    public String getTitle() {
        return "Tasks";
    }

    @WorkbenchPartView
    public UberView<TasksPresenter> getView() {
        return view;
    }

    public void projectSelected( @Observes ProjectSelectedEvent projectSelectedEvent ) {
        this.currentSelectedProject = projectSelectedEvent.getName();
        selectFolder();
    }
```

```
    private void selectFolder() {
        view.activateNewFolder();
        updateView();
    }

    public void showNewFolder() {
        newFolderPresenter.show( this );
    }

    public void createTask( String folderName,
                            String task ) {

        Folder folder = getFolder( folderName );
        if ( folder != null ) {
            folder.addTask( task );
        }
        updateView();
    }

    private Folder getFolder( String folderName ) {
        for ( final Folder folder : getFolders() ) {
            if ( folder.getName().equalsIgnoreCase( folderName ) ) {
                return folder;
            }
        }
        return null;
    }

    public void doneTask( String folderName,
                          String taskText ) {
        Folder folder = getFolder( folderName );
        if ( folder != null ) {
            folder.removeTask( taskText );
        }
        updateView();
    }

    private List<Folder> getFolders() {
        List<Folder> folders = foldersPerProject.get( currentSelectedProject );
        if ( folders == null ) {
            folders = new ArrayList<Folder>();
        }
        return folders;
    }

    private void updateView() {
        view.clearTasks();
        for ( final Folder folder : getFolders() ) {
            view.newFolder( folder.getName(), folder.getTasks().size(), folder.getTasks() );
        }
    }

    public void newFolder( String folderName ) {
        List<Folder> folders = getFolders();
        folders.add( new Folder( folderName ) );
        foldersPerProject.put( currentSelectedProject, folders );
        updateView();
    }
}
```

Let's highlight some important pieces of code

- Map< String, List < Folder > > foldersPerProject: keeps a in memory map of folders for each project selected;
- createTask(folderName,task): add a task for a specific folder. This is triggered by addTasks text box;
- doneTask(folderName,taskText): after a task is marked as done, remove it from folder;
- newFolder(folderName): create a new folder and add it in persistence structure (folderPerProject).

It's update the **TasksView.java** in order to support the creation of tasks.

```
package org.uberfire.client.screens;

import java.util.List;
import javax.annotation.PostConstruct;
import javax.enterprise.context.Dependent;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyDownEvent;
import com.google.gwt.event.dom.client.KeyDownHandler;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Widget;
import org.gwtbootstrap3.client.ui.Badge;
import org.gwtbootstrap3.client.ui.Button;
import org.gwtbootstrap3.client.ui.InlineCheckBox;
import org.gwtbootstrap3.client.ui.InputGroup;
import org.gwtbootstrap3.client.ui.ListGroup;
import org.gwtbootstrap3.client.ui.ListGroupItem;
import org.gwtbootstrap3.client.ui.TextBox;
import org.gwtbootstrap3.client.ui.constants.ListGroupItemType;

@Dependent
public class TasksView extends Composite implements TasksPresenter.View {

    @UiField
    Button newFolder;

    @UiField
    FlowPanel tasks;

    private TasksPresenter presenter;
```

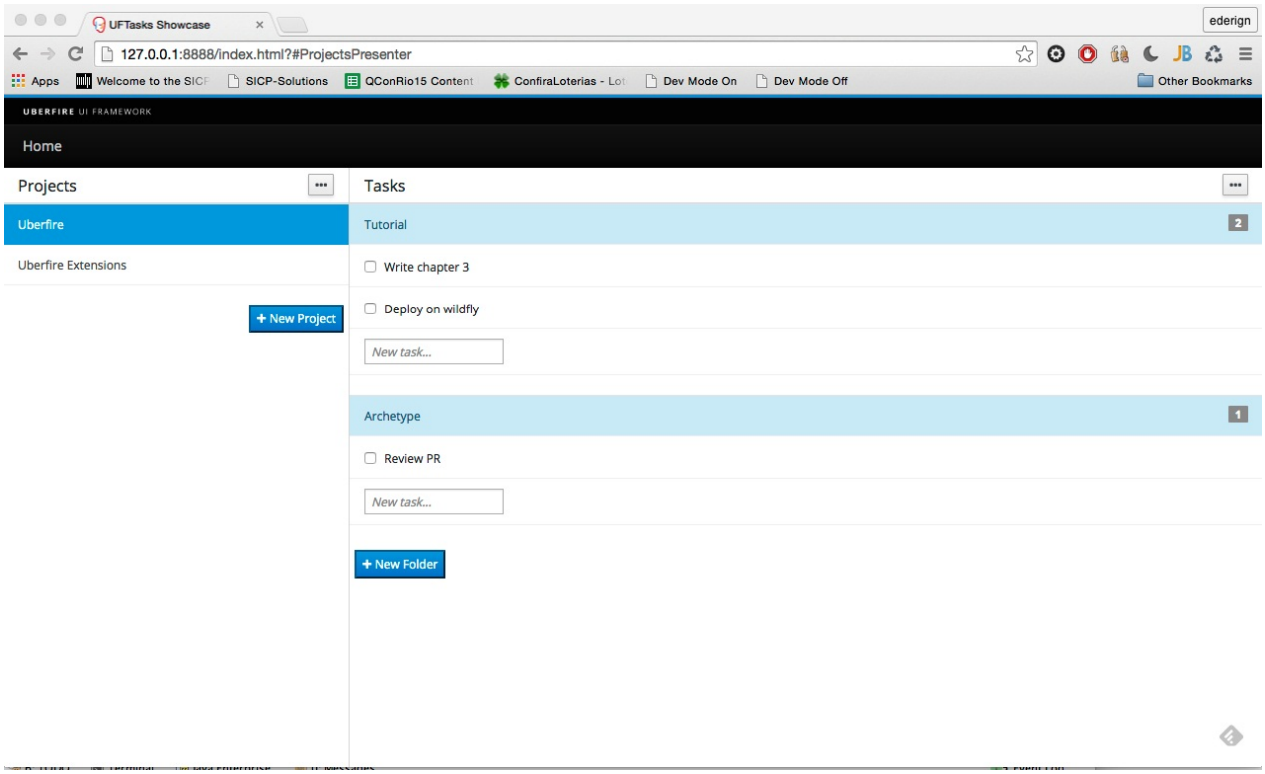
```
interface Binder
        extends
        UiBinder<Widget, TasksView> {

}

private static Binder uiBinder = GWT.create( Binder.class );

@Override
public void init( final TasksPresenter presenter ) {
    this.presenter = presenter;
    this.newFolder.setVisible( false );
}

@PostConstruct
public void setup() {
    initWidget( uiBinder.createAndBindUi( this ) );

}

@Override
public void activateNewFolder( ) {
    newFolder.setVisible( true );
    newFolder.addClickHandler( new ClickHandler() {
        @Override
        public void onClick( ClickEvent event ) {
            presenter.showNewFolder();
        }
    } );
}

@Override
public void clearTasks() {
    tasks.clear();
}

@Override
public void newFolder( String folderName,
                       Integer numberOfTasks,
                       List<String> tasksList ) {

    ListGroup folder = GWT.create( ListGroup.class );
    folder.add( generateFolderTitle( folderName, numberOfTasks ) );

    for ( String task : tasksList ) {
        folder.add( generateTask( folderName, task ) );
    }
    folder.add( generateNewTask( folderName ) );
    tasks.add( folder );
}

private ListGroupItem generateNewTask( String folderName ) {
    ListGroupItem newTask = GWT.create( ListGroupItem.class );
    InputGroup inputGroup = GWT.create( InputGroup.class );
    inputGroup.add( createTextBox( folderName ) );
    newTask.add( inputGroup );
    return newTask;
}
```

```
    private TextBox createTextBox( final String folderName ) {
        final TextBox taskText = GWT.create( TextBox.class );
        taskText.addKeyDownHandler( new KeyDownHandler() {
            @Override
            public void onKeyDown( KeyDownEvent event ) {
                if ( event.getNativeKeyCode() == KeyCodes.KEY_ENTER ) {
                    presenter.createTask( folderName, taskText.getText() );
                }
            }
        } );
        taskText.setWidth( "400" );
        taskText.setPlaceholder( "New task..." );
        return taskText;
    }

    private ListGroupItem generateFolderTitle( String name,
                                               Integer numberOfTasks ) {
        ListGroupItem folderTitle = GWT.create( ListGroupItem.class );
        folderTitle.setText( name );
        folderTitle.setType( ListGroupItemType.INFO );
        Badge number = GWT.create( Badge.class );
        number.setText( String.valueOf( numberOfTasks ) );
        folderTitle.add( number );
        return folderTitle;
    }

    private ListGroupItem generateTask( String folderName,
                                        String taskText ) {
        ListGroupItem tasks = GWT.create( ListGroupItem.class );
        tasks.add( createTaskCheckbox( folderName, taskText ) );
        return tasks;
    }

    private InlineCheckBox createTaskCheckbox( final String folderName,
                                               final String taskText ) {
        InlineCheckBox checkBox = GWT.create( InlineCheckBox.class );
        checkBox.addClickHandler( new ClickHandler() {
            @Override
            public void onClick( ClickEvent event ) {
                presenter.doneTask( folderName, taskText );
            }
        } );
        checkBox.setText( taskText );
        return checkBox;
    }

}
```

Pay attention to the method newFolder(folderName, numberOfTasks, tasksList), because here is where we create the components for task folder: a folderTitle, a list of tasks and newTask textbox.

Time to see it work!

Refresh the browser, create two projects and click in one of them. Create a new folder and add some tasks for it.



Not Working?

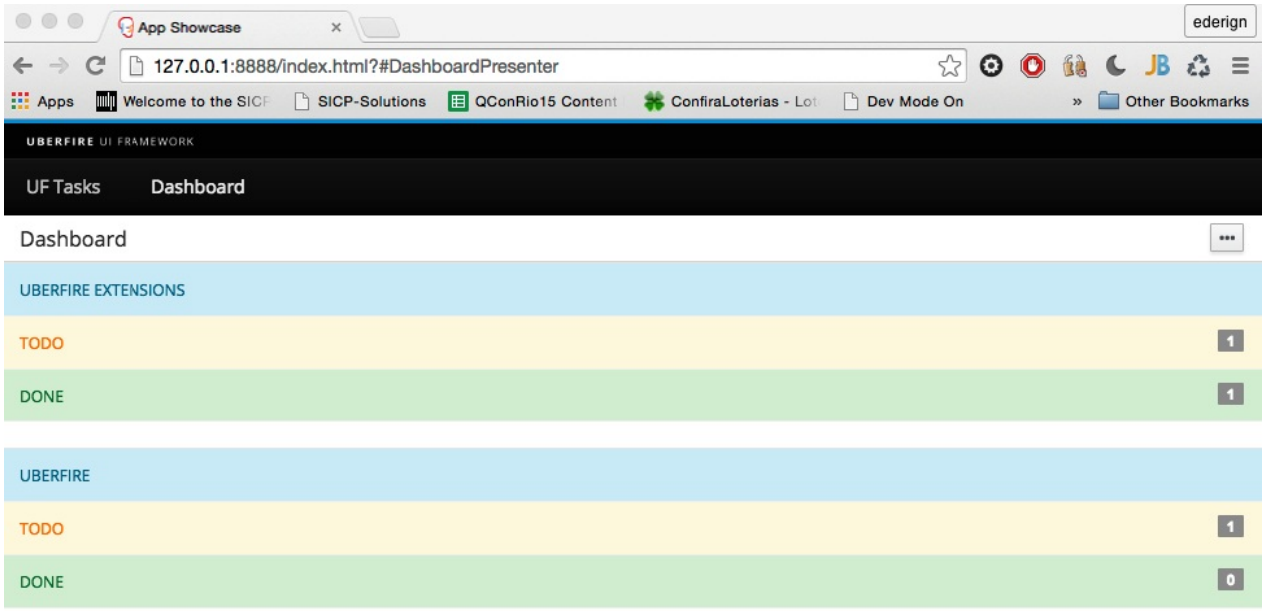
At this point, your project should be more-or-less identical to the Tutorial project at the checkpoint-3 tag. If your project isn't working at this point, grab that one and compare yours with it.

```
$ git clone https://github.com/uberfire/uberfire-tutorial.git
$ cd uberfire-tutorial
$ git checkout checkpoint-3
```

Let's improve your app

In order to get in touch with some important Uberfire concepts like adding more Perspectives, using PlaceManagers, etc. . So let's improve your tasks app and build a basic dashboard on it.

This dashboard will count the number of tasks created and done by project and will look like this:



Because we are writing new perspectives, screens and Uberfire needs to generate some code, stop the server and let's back to work.

Dashboard Perspective

Create this class on package org.uberfire.client.perspectives:

```
package org.uberfire.client.perspectives;

import javax.enterprise.context.ApplicationScoped;

import org.uberfire.client.annotations.Perspective;
import org.uberfire.client.annotations.WorkbenchPerspective;
import org.uberfire.client.workbench.panels.impl.MultiListWorkbenchPanelPresenter;
import org.uberfire.client.workbench.panels.impl.SimpleWorkbenchPanelPresenter;
import org.uberfire.mvp.impl.DefaultPlaceRequest;
import org.uberfire.workbench.model.CompassPosition;
import org.uberfire.workbench.model.PanelDefinition;
import org.uberfire.workbench.model.PerspectiveDefinition;
import org.uberfire.workbench.model.impl.PanelDefinitionImpl;
import org.uberfire.workbench.model.impl.PartDefinitionImpl;
import org.uberfire.workbench.model.impl.PerspectiveDefinitionImpl;

@ApplicationScoped
@WorkbenchPerspective(identifier = "DashboardPerspective")
public class DashboardPerspective {

    @Perspective
    public PerspectiveDefinition buildPerspective() {
        final PerspectiveDefinitionImpl perspective = new PerspectiveDefinitionImpl( MultiLi
```

```
        perspective.setName( "DashboardPerspective" );

        perspective.getRoot().addPart( new PartDefinitionImpl( new DefaultPlaceRequest( "Das

        return perspective;
    }
}
```

This class has one presenter (DashboardPresenter) and note that it is a little bit different from TaskPerspective, because it is not a default perspective. You can have only one default perspective that will be automatic loaded on Uberfire startup.

DashBoardPresenter

On org.uberfire.client.screens package:

```
package org.uberfire.client.screens;

import java.util.HashMap;
import java.util.Map;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;
import javax.inject.Inject;

import org.uberfire.client.annotations.WorkbenchPartTitle;
import org.uberfire.client.annotations.WorkbenchPartView;
import org.uberfire.client.annotations.WorkbenchScreen;
import org.uberfire.client.mvp.UberView;
import org.uberfire.lifecycle.OnOpen;
import org.uberfire.shared.events.TaskCreated;
import org.uberfire.shared.events.TaskDone;

@ApplicationScoped
@WorkbenchScreen(identifier = "DashboardPresenter")
public class DashboardPresenter {

    public interface View extends UberView<DashboardPresenter> {

        void addProject( String project,
                         String tasksCreated,
                         String tasksDone );

        void clear();
    }

    @Inject
    private View view;

    private Map<String, ProjectTasksCounter> projectTasksCounter = new HashMap<String, Proje

    @WorkbenchPartTitle
```

```
    public String getTitle() {
        return "Dashboard";
    }

    @WorkbenchPartView
    public UberView<DashboardPresenter> getView() {
        return view;
    }

    @OnOpen
    public void onOpen() {
        updateView();
    }

    private void updateView() {
        view.clear();
        for ( String project : projectTasksCounter.keySet() ) {
            ProjectTasksCounter projectTasksCounter = this.projectTasksCounter.get( project
            view.addProject( project, projectTasksCounter.getTasksCreated(), projectTasksCou
        }
    }

    public void taskCreated( @Observes TaskCreated taskCreated ) {
        ProjectTasksCounter projectTasksCounter = getProjectTasksCounter( taskCreated.getPro
        projectTasksCounter.taskCreated();
    }

    public void taskDone( @Observes TaskDone taskDone ) {
        ProjectTasksCounter projectTasksCounter = getProjectTasksCounter( taskDone.getProjec
        projectTasksCounter.taskDone();
    }

    public ProjectTasksCounter getProjectTasksCounter( String projectName ) {
        ProjectTasksCounter projectTasksCounter = this.projectTasksCounter.get( projectName
        if ( projectTasksCounter == null ) {
            projectTasksCounter = new ProjectTasksCounter();
            this.projectTasksCounter.put( projectName, projectTasksCounter );
        }
        return projectTasksCounter;
    }

    private class ProjectTasksCounter {

        int tasksDone;
        int tasksCreated;

        public void taskDone() {
            tasksDone++;
            tasksCreated--;
        }

        public void taskCreated() {
            tasksCreated++;
        }

        public String getTasksDone() {
            return String.valueOf( tasksDone );
        }
```

```
        public String getTasksCreated() {
            return String.valueOf( tasksCreated );
        }
    }

}
```

This presenter has a Map and a utility class to keep track of tasks created and done. But how we keep track of the tasks changes? Let's pay a close attention to these methods:

```
  public void taskCreated( @Observes TaskCreated taskCreated ) {
        ProjectTasksCounter projectTasksCounter = getProjectTasksCounter( taskCreated.getPro
        projectTasksCounter.taskCreated();
    }

    public void taskDone( @Observes TaskDone taskDone ) {
        ProjectTasksCounter projectTasksCounter = getProjectTasksCounter( taskDone.getProjec
        projectTasksCounter.taskDone();
    }
```

There is two observer methods listening tasks events. Let's create this event classes and fire them?

On org.uberfire.shared.events package:

```
package org.uberfire.shared.events;

public class TaskCreated {

    private final String project;
    private final String folder;
    private final String task;

    public TaskCreated( String project,
                        String folder,
                        String task ) {

        this.project = project;
        this.folder = folder;
        this.task = task;
    }

    public String getProject() {
        return project;
    }
}
```

And a event class for TaskDone:

```
package org.uberfire.shared.events;

public class TaskDone {

    private final String project;
    private final String folder;
    private final String task;

    public TaskDone( String project,
                     String folder,
                     String task ) {

        this.project = project;
        this.folder = folder;
        this.task = task;
    }

    public String getProject() {
        return project;
    }
}
```

Remember that the TaskPresenter create and mark as done tasks? So let's change these methods to fire a CDI event:

```
    @Inject
    private Event<TaskCreated> taskCreatedEvent;

    @Inject
    private Event<TaskDone> taskDoneEvent;

    public void doneTask( String folderName,
                          String taskText ) {
        Folder folder = getFolder( folderName );
        if ( folder != null ) {
            folder.removeTask( taskText );
        }
        taskDoneEvent.fire( new TaskDone(currentSelectedProject,folderName, taskText) );
        updateView();
    }

    public void createTask( String folderName,
                            String task ) {

        Folder folder = getFolder( folderName );
        if ( folder != null ) {
            folder.addTask( task );
        }
        taskCreatedEvent.fire( new TaskCreated(currentSelectedProject,folderName, task) );
        updateView();
    }
```

DashBoardView

Now it's time to create our view classes (org.uberfire.client.screens package):

DashboardView.java

```
package org.uberfire.client.screens;

import javax.annotation.PostConstruct;
import javax.enterprise.context.Dependent;

import com.google.gwt.core.client.GWT;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Widget;
import org.gwtbootstrap3.client.ui.Badge;
import org.gwtbootstrap3.client.ui.ListGroup;
import org.gwtbootstrap3.client.ui.ListGroupItem;
import org.gwtbootstrap3.client.ui.constants.ListGroupItemType;

@Dependent
public class DashboardView extends Composite implements DashboardPresenter.View {

    interface Binder
            extends
            UiBinder<Widget, DashboardView> {

    }

    private static Binder uiBinder = GWT.create( Binder.class );

    private DashboardPresenter presenter;

    @UiField
    FlowPanel projects;

    @PostConstruct
    public void setup() {
        initWidget( uiBinder.createAndBindUi( this ) );
    }

    @Override
    public void init( DashboardPresenter presenter ) {
        this.presenter = presenter;
    }

    @Override
    public void addProject( String project,
                            String tasksCreated,
                            String tasksDone ) {
```

```
        ListGroup projectGroup = GWT.create( ListGroup.class );

        projectGroup.add( createListGroupItem( ListGroupItemType.INFO, project.toUpperCase()
        projectGroup.add( createListGroupItem( ListGroupItemType.WARNING, "TODO", String.val
        projectGroup.add( createListGroupItem( ListGroupItemType.SUCCESS, "DONE", String.val

        projects.add( projectGroup );
    }

    @Override
    public void clear() {
        projects.clear();
    }

    private ListGroupItem createListGroupItem( ListGroupItemType type,
                                               String text,
                                               String number ) {
        ListGroupItem item = GWT.create( ListGroupItem.class );
        item.setType( type );
        item.setText( text );
        if ( number != null ) {
            Badge numberBadge = GWT.create( Badge.class );
            numberBadge.setText( number );
            item.add( numberBadge );
        }
        return item;
    }

}
```

DashboardView.ui.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui"
    >

  <ui:with field='res' type='org.uberfire.client.resources.AppResource'/>

  <g:FlowPanel ui:field="projects"/>


</ui:UiBinder>
```

Create Perspective Menu

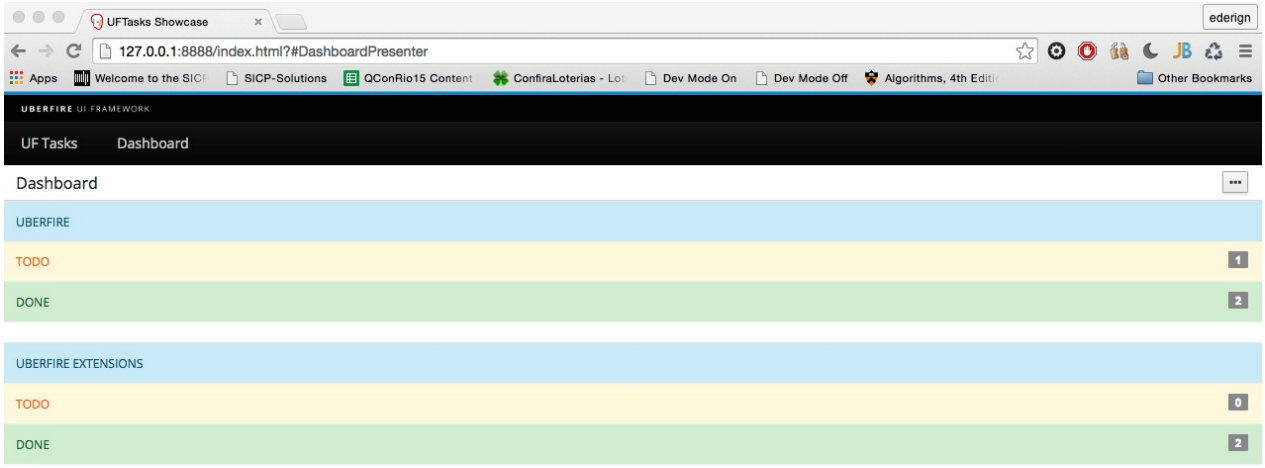
Now it's time to put this work on our app menu. Open ShowcaseEntryPoint.java and update method setupMenu(event):

```
private void setupMenu( @Observes final ApplicationReadyEvent event ) {
        final Menus menus =
                newTopLevelMenu( "UF Tasks" )
                        .respondsWith( new Command() {
                            @Override
                            public void execute() {
                                placeManager.goTo( new DefaultPlaceRequest( "TasksPerspectiv
                            }
                        } )
                        .endMenu().
        newTopLevelMenu( "Dashboard" )
                .respondsWith( new Command() {
                    @Override
                    public void execute() {
                        placeManager.goTo( new DefaultPlaceRequest( "DashboardPerspective" )
                    }
                } )
                .endMenu()
                        .build();

        menubar.addMenus( menus );
    }
```

Pay attention to placeManager.goTo(new DefaultPlaceRequest("DashboardPerspective")) call. This method is a Workbench-centric abstraction over the browser's history mechanism. Allows the application to initiate navigation to any displayable thing: a WorkbenchPerspective, a WorkbenchScreen, a WorkbenchPopup, a WorkbenchEditor, a WorkbenchPart within a screen or editor, or the editor associated with a VFS file.

Time to see it work!

Start the server, refresh the browser, create two projects and click in one of them. Create a new folder and add some tasks for it and mark some of them as done. Click on Dashboard menu:



Not Working?

At this point, your project should be more-or-less identical to the Tutorial project at the checkpoint-4 tag. If your project isn't working at this point, grab that one and compare yours with it.

```
$ git clone https://github.com/uberfire/uberfire-tutorial.git
$ cd uberfire-tutorial
$ git checkout checkpoint-4
```

Next

We hope you like this tutorial. If you find any issues or want to contribute on it, feel free to send a pull request for Uberfire Docs and Uberfire Tutorial projects.

Now, let's deploy this app in Tomcat and Wildfly.

Deploy on Tomcat

This guide will help you install an UberFire demo application on your own computer. This will help let you try out an UberFire application in Tomcat, and prove the UberFire apps work with your setup.

Building our App

This demo will be based on UF tasks app from previous section. If you didn't work on this App, fell download it and build it from source code:

```
git clone https://github.com/uberfire/uberfire-tutorial.git
cd uftasks
mvn clean install
```

Get an app server

UFTasks was generated from Uberfire Archetype. So inside uftasks-showcase, there is a directory called uftasks-distributions-wars that has inside target directory, WAR files for JBoss EAP 6.4, Tomcar 7.0 and Wildfly 8.1. Let's install this app on Tomcat 7.0

Get an app server

If you don't already have Tomcat 7.0 installed on your computer, you can download and instalk it. Installing is as easy as downloading and unzipping.

Start the app server

Now start the app server using a command line terminal. Use the cd command to change the working directory of your terminal to the place where you unzipped the application server, then execute one of the following commands, based on your operating system and choice of app server:

*nix, Mac OS X	Windows

bin/startup.sh	bin\startup.bat

Then visit the URL http://localhost:8080/ and you should see a webpage confirming that the app server is running.

Deploy the WAR

Rename the UFtasks tomcat WAR file to uftasks.war and copy it into the auto-deployment directory for your app server. In example on Unix/Linux/Mac:

```
mv uftasks-showcase-1.0-SNAPSHOT-tomcat7.0.war /~/bin/apache-tomcat-7.0.57/webapps/uftasks.w
```

See it work!

Now visit http://localhost:8080/uftasks/ and sign in with username admin, password admin.

Deploy on Wildfly

This guide will help you install an UberFire demo application on your own computer. This will help let you try out an UberFire application in Wildfly, and prove the UberFire apps work with your setup.

Building our App

This demo will be based on UF tasks app from previous section. If you didn't work on this App, fell download it and build it from source code:

```
git clone https://github.com/uberfire/uberfire-tutorial.git
cd uftasks
mvn clean install
```

Get an app server

UFTasks was generated from Uberfire Archetype. So inside uftasks-showcase, there is a directory called uftasks-distributions-wars that has inside target directory, WAR files for JBoss EAP 6.4, Tomcar 7.0 and Wildfly 8.1. Let's install this app on Tomcat 7.0

Get an app server

If you don't already have Wildfly 8.1 installed on your computer, you can download and install it. Installing is as easy as downloading and unzipping.

Start the app server

Now start the app server using a command line terminal. Use the cd command to change the working directory of your terminal to the place where you unzipped the application server, then execute one of the following commands, based on your operating system and choice of app server:

*nix, Mac OS X	Windows

bin/standalone.sh	bin\standalone.bat

Then visit the URL http://localhost:8080/ and you should see a webpage confirming that the app server is running.

Deploy the WAR

Rename the UFtasks wildfly WAR file to uftasks.war and copy it into the auto-deployment directory for your app server. In example on Unix/Linux/Mac:

```
mv uftasks-showcase-1.0-SNAPSHOT-wildfly8.1.war /~/bin/wildfly-8.1.0.Final/standalone/deploy
```

See it work!

Now visit http://localhost:8080/uftasks/ and sign in with username admin, password admin.

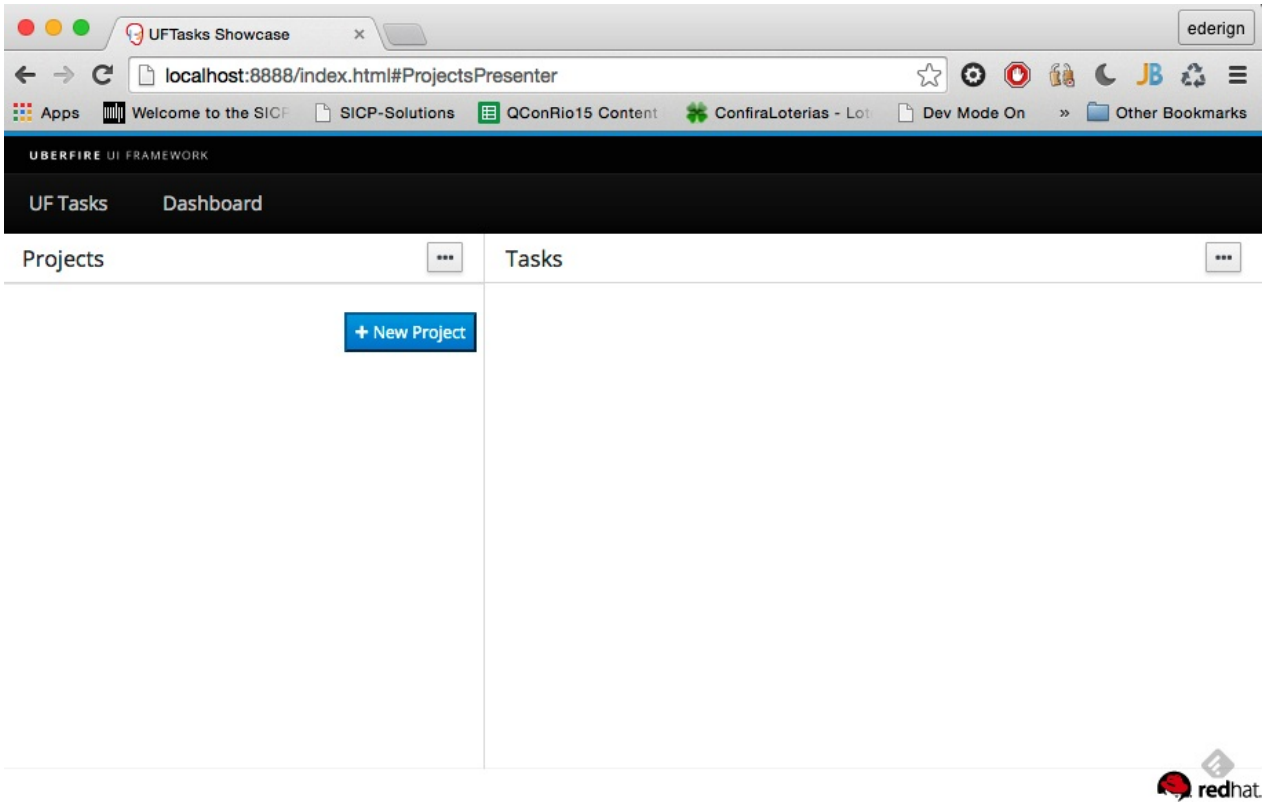
Uberfire Model

Uberfire has several modeling concepts. This section will detail these important points of the framework.

Workbench

UberFire workbenches are made of some fundamental building blocks: Widgets, Layout Panels, Screens, Workbench Panels, Menu Bars, Tool Bars, and Perspectives.

Layout Panels can contain Widgets and other Layout Panels; Perspectives contain Workbench Panels, an optional Menu Bar, and an optional Tool Bar. Perspectives split up the screen into multiple resizable regions, and end users can drag and drop Panels between these regions to customize their workspace.

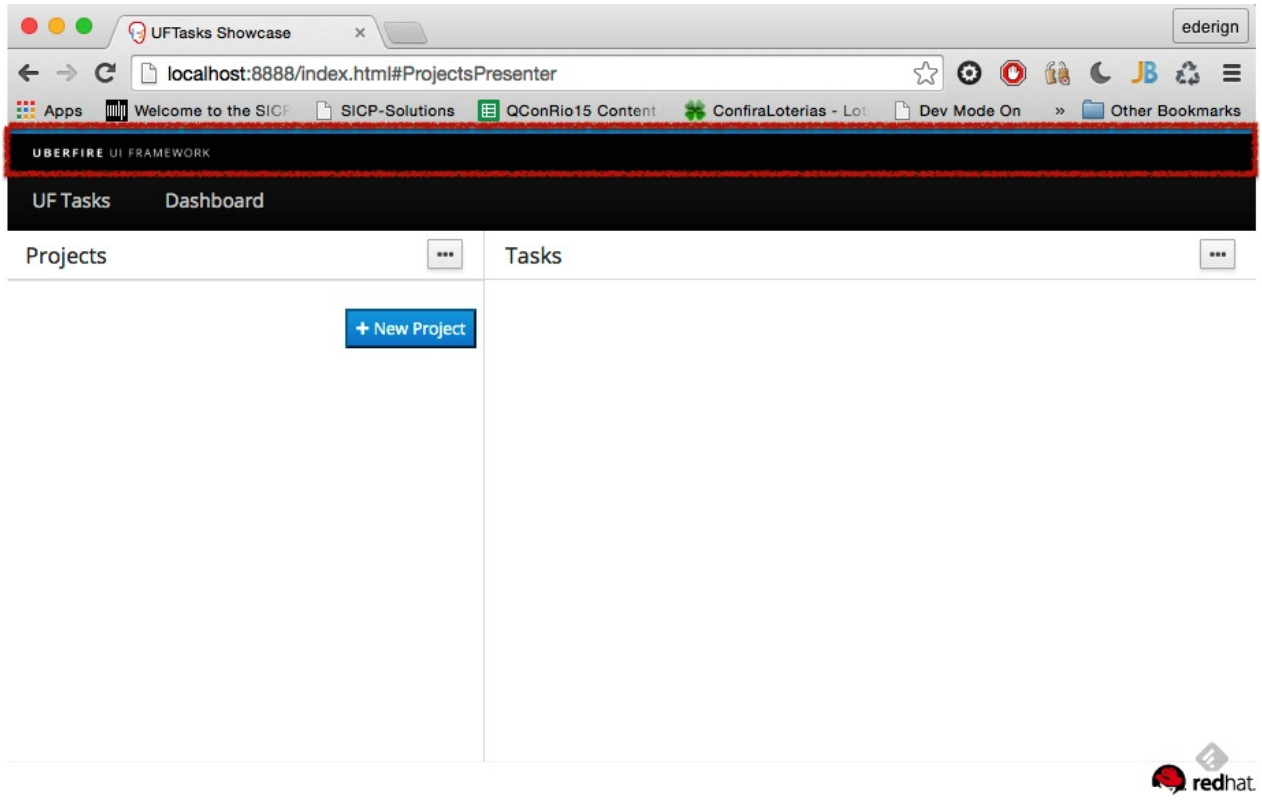


Let's detail each of these concepts.

Header

Headers in Uberfire are automatically discovered via CDI and added to the top of the Workbench screen. They stick to the top of the viewport even when the main content area is scrolled up and down or the browser window is resized.

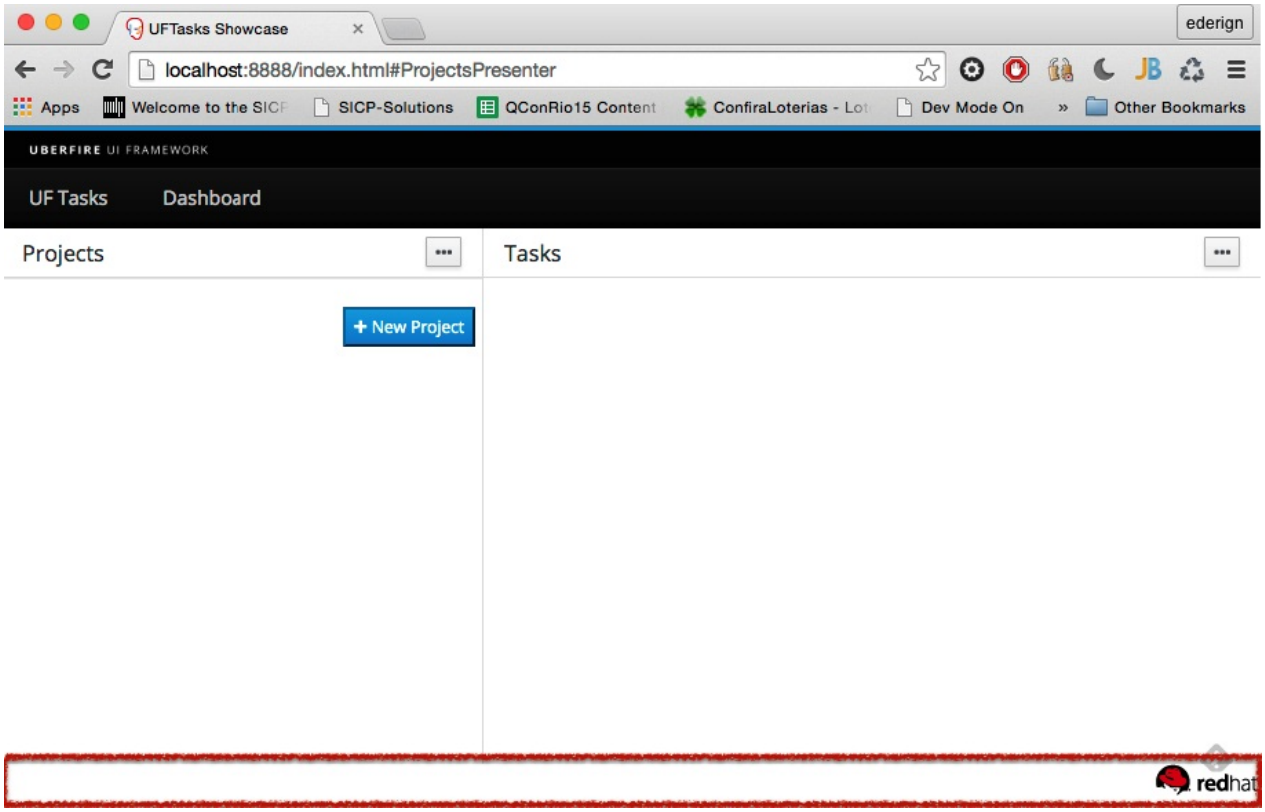
In order to create a header you have to create a CDI bean that implement org.uberfire.client.workbench.Header interface.



Footer

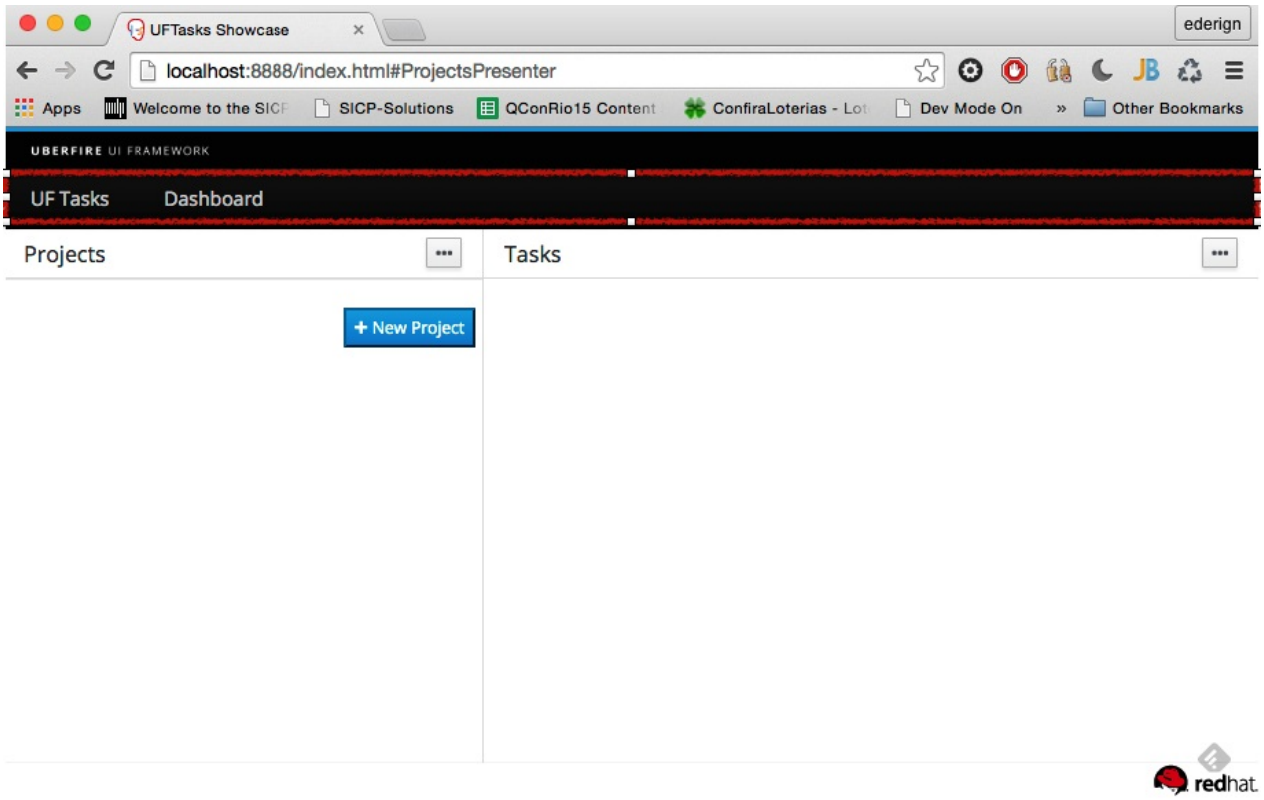
Footers in Uberfire are automatically discovered via CDI and added to the bottom of the Workbench screen. They stick to the bottom of the viewport even when the main content area is scrolled up and down or the browser window is resized.

In order to create a footer you have to create a CDI bean that implement org.uberfire.client.workbench.Footer interface.



Menu

Menu bars typically live at the top of the screen, and UberFire's perspective layout system gives you a way to place widgets in just that spot.



In order to create a menu bar in the top of the app, create a class that implements Header and add menus to it via WorkbenchMenuBarPresenter CDI bean. As an example, the Menu of UFTasks app:

ShowcaseEntryPoint.java

```
@Inject
private WorkbenchMenuBar menubar;

private void setupMenu( @Observes final ApplicationReadyEvent event ) {
    final Menus menus =
            newTopLevelMenu( "UF Tasks" )
                    .respondsWith( new Command() {
                        @Override
                        public void execute() {
                            placeManager.goTo( new DefaultPlaceRequest( "TasksPerspectiv
                        }
                    } )
                    .endMenu().
                    newTopLevelMenu( "Dashboard" )
                    .respondsWith( new Command() {
```

```
                            @Override
                            public void execute() {
                                placeManager.goTo( new DefaultPlaceRequest( "DashboardPerspe
                            }
                        } )
                        .endMenu()
                        .build();

        menubar.addMenus( menus );
    }
```

AppNavBar.java

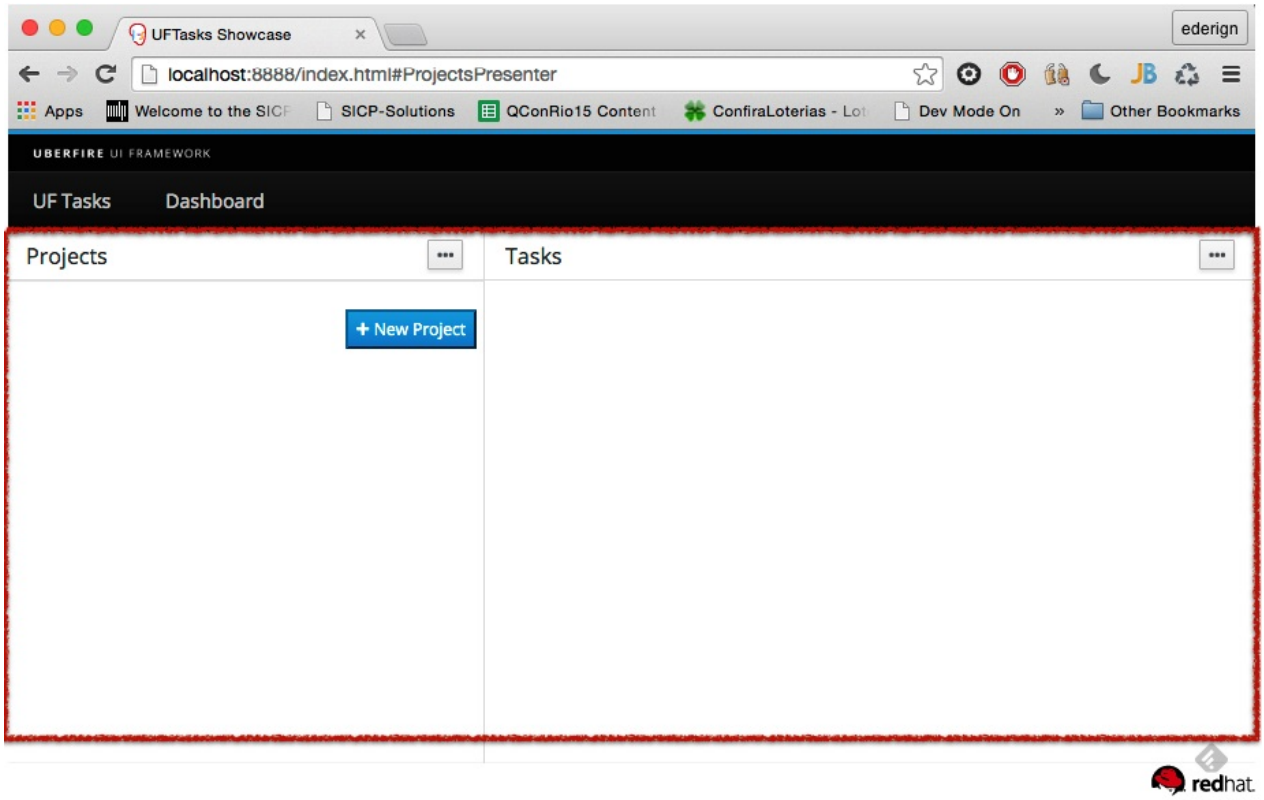
```
@ApplicationScoped
public class AppNavBar
        extends Composite implements Header {

    @Inject
    private WorkbenchMenuBarPresenter menuBarPresenter;

    @Override
    public Widget asWidget() {
        return menuBarPresenter.getView().asWidget();
    }

    @Override
    public String getId() {
        return "AppNavBar";
    }

    @Override
    public int getOrder() {
        return MAX_VALUE;
    }
}
```

Perspective

An Uberfire Workbench contains one or more perspectives. The UberFire workbench UI is arranged as:

Workbench → Perspective → Workbench Panel → Component

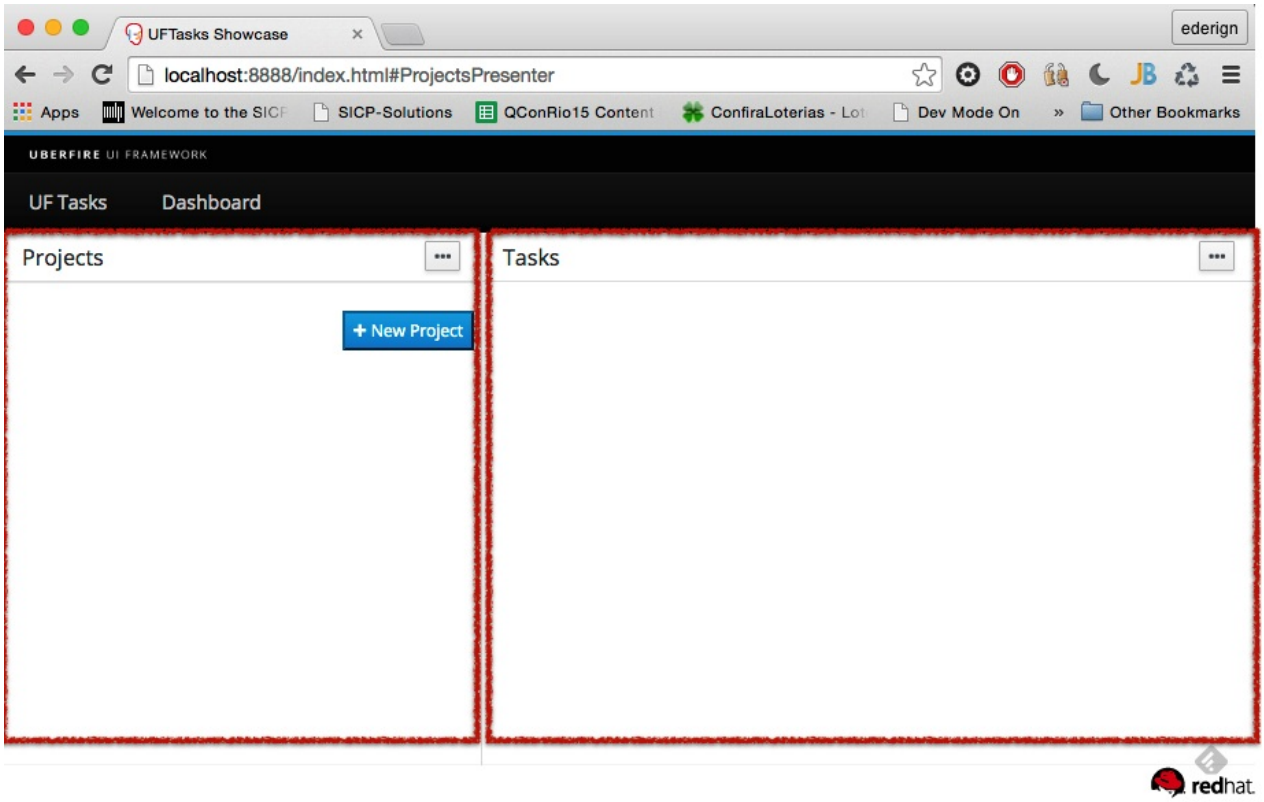
Perspectives split up the screen into multiple resizable regions, and end users can drag and drop Panels between these regions to customize their workspace. Perspectives dictate the position and size of Workbench Panels.



Each Perspective has a PerspectiveDefinition. A PerspectiveDefinition is a meta-data defining a Perspective.

Panel

Each Uberfire Perspective contains multiple Panels.



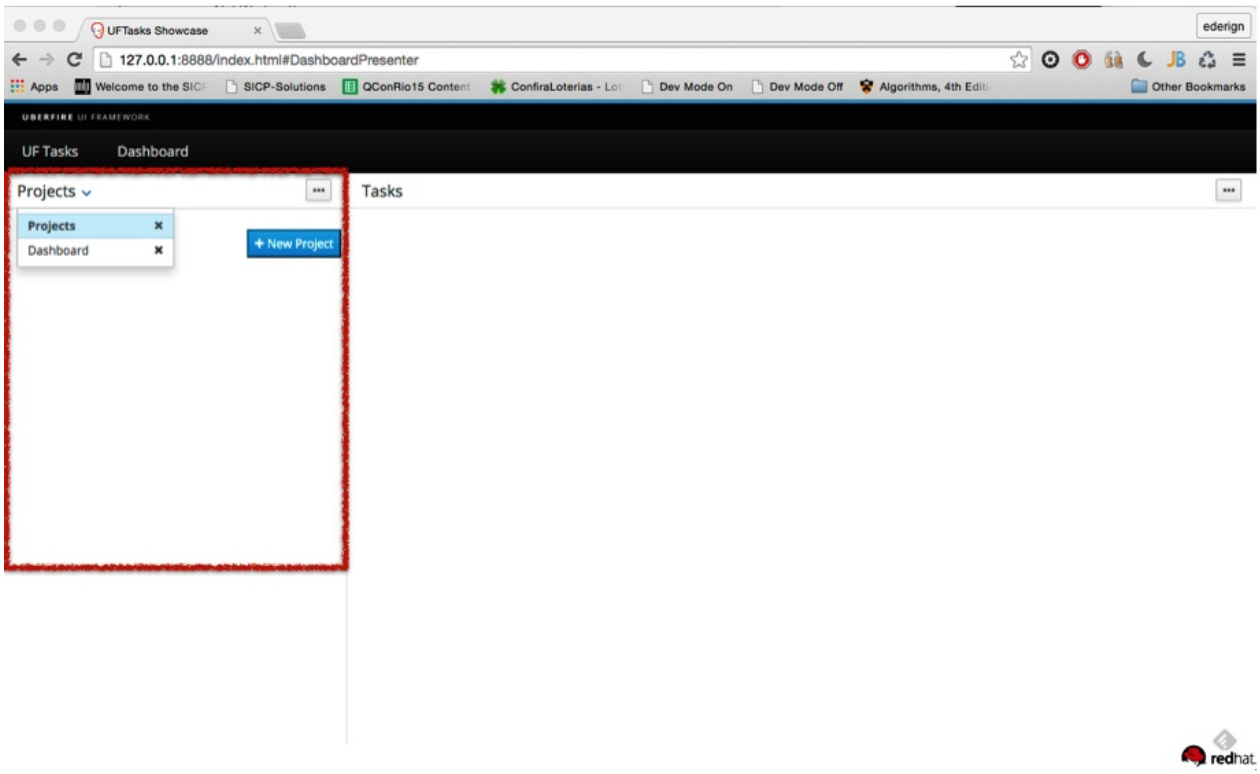
A Panel describes a physical region within a Workbench Perspective. Panels have a set physical size that they occupy, which is divided up between any panel decorations (a tab bar or dropdown list is common), one or more Parts (generally Editors or Screens), one of which can be visible at a time, and also child Panel Definitions, all of which are visible simultaneously.

Part

A part specifies content that should be put in a panel main display area when it is materialized. The content add is specified by a PartDefinition, at the core of which is a PlaceRequest that identifies a WorkbenchActivity (an UberfireComponent like screen, editor, popups or splashscreen).

In example, an panel that contains two parts:

```
final PanelDefinition west = new PanelDefinitionImpl( SimpleWorkbenchPanelPresenter.class.ge
west.addPart( new PartDefinitionImpl( new DefaultPlaceRequest( "ProjectsPresenter" ) ) );
west.addPart( new PartDefinitionImpl( new DefaultPlaceRequest( "DashboardPresenter" ) ) );
```



Component

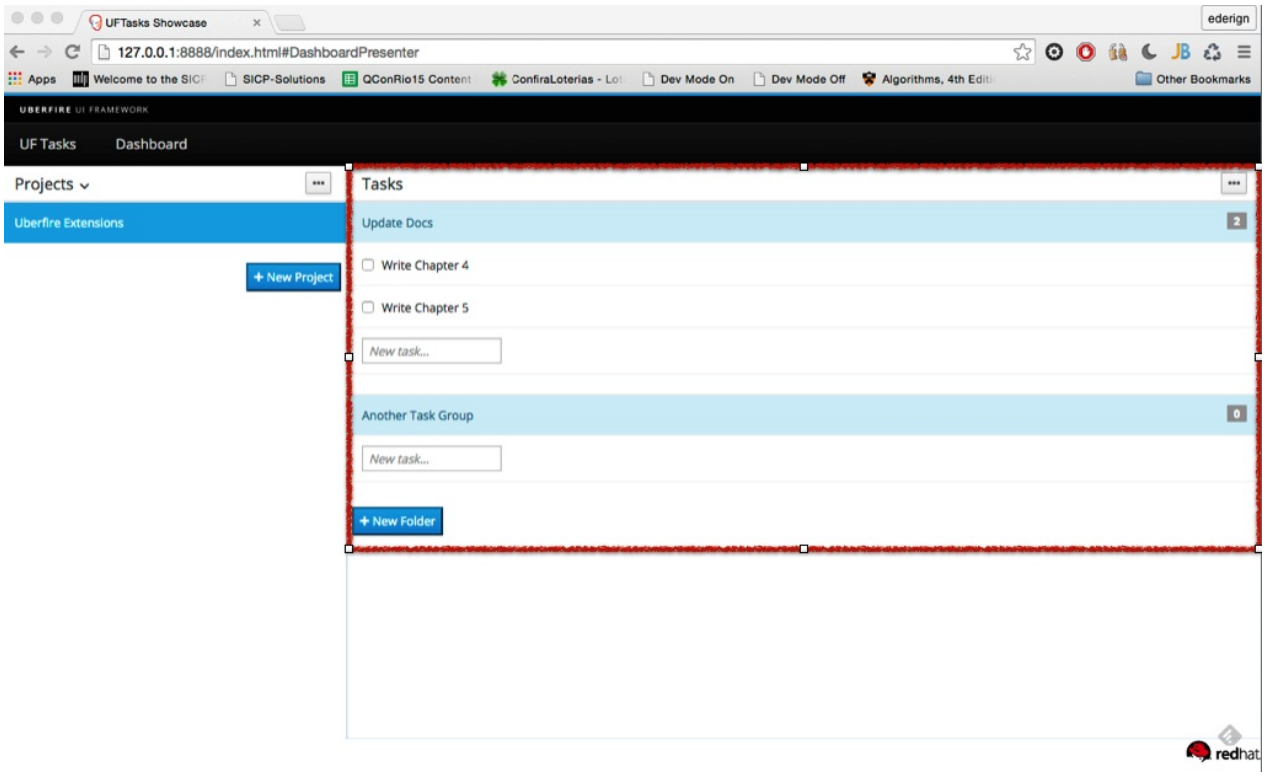
A Uberfire Component is defined by a WorkbenchActivity. This activity is interface between UberFire framework behaviour and application-defined behaviour

In the model-view-presenter (MVP) sense, an Activity is essentially an application-provided Presenter: it has a view (its widget) and it defines a set of operations that can affect that view.

Applications can implement an Activity interface directly, they can subclass one of the abstract Activity implementations that come with the framework, or they may rely on UberFire's annotation processors to generate Activity implementations from annotated Java objects.

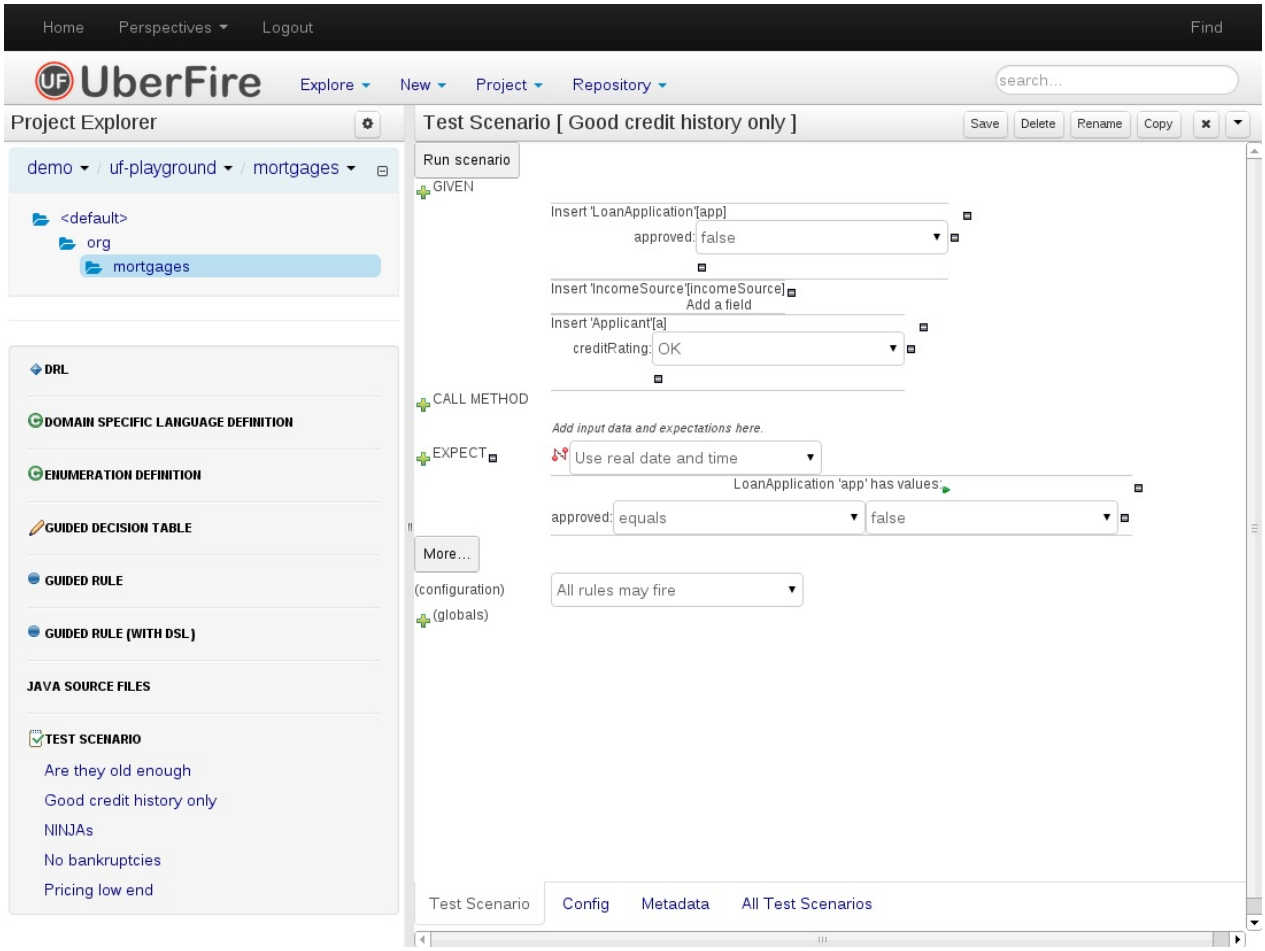
Screen

A Screen is a part that provides the application-defined behavior and application-defined view associated with a particular Uberfire Place.



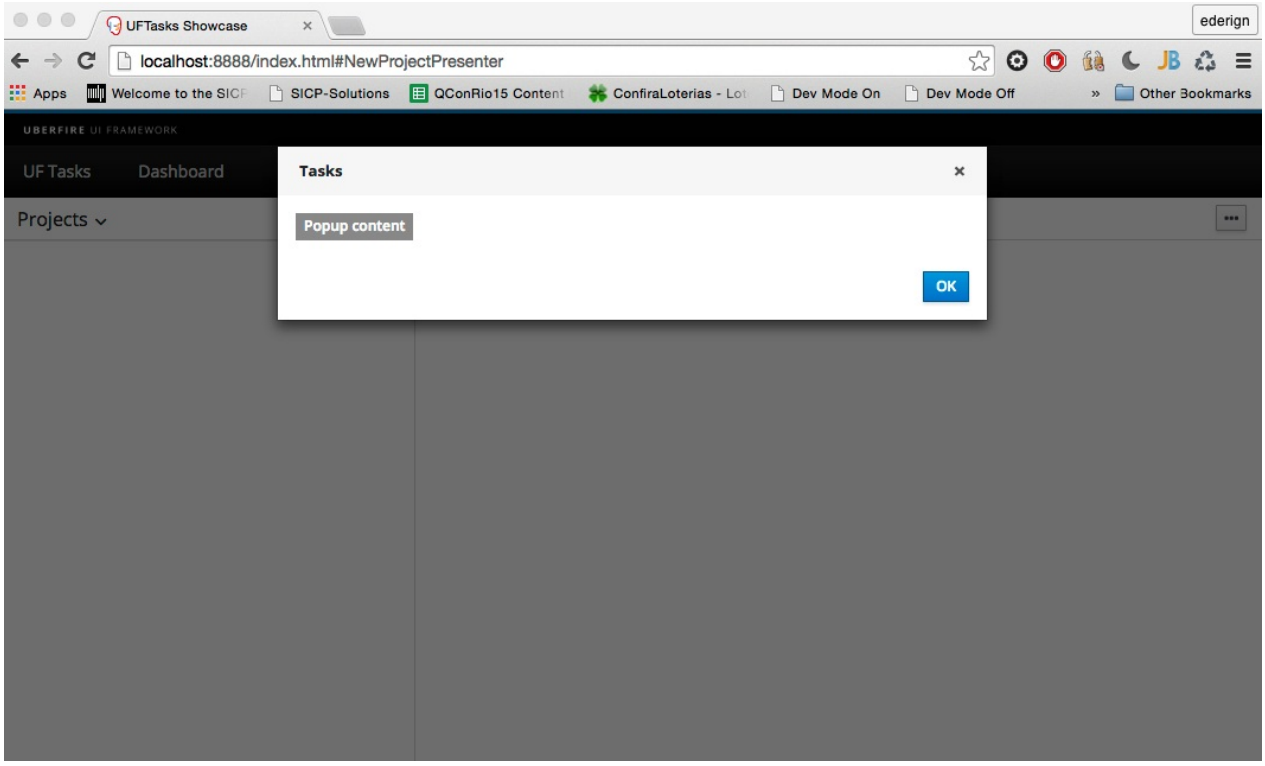
Editor

An Editor is a WorkbenchPart that perform some editor function to a specific file-type. The content is associated with a VFS path and usually represents some sort of document that can be edited and saved.



PopUp

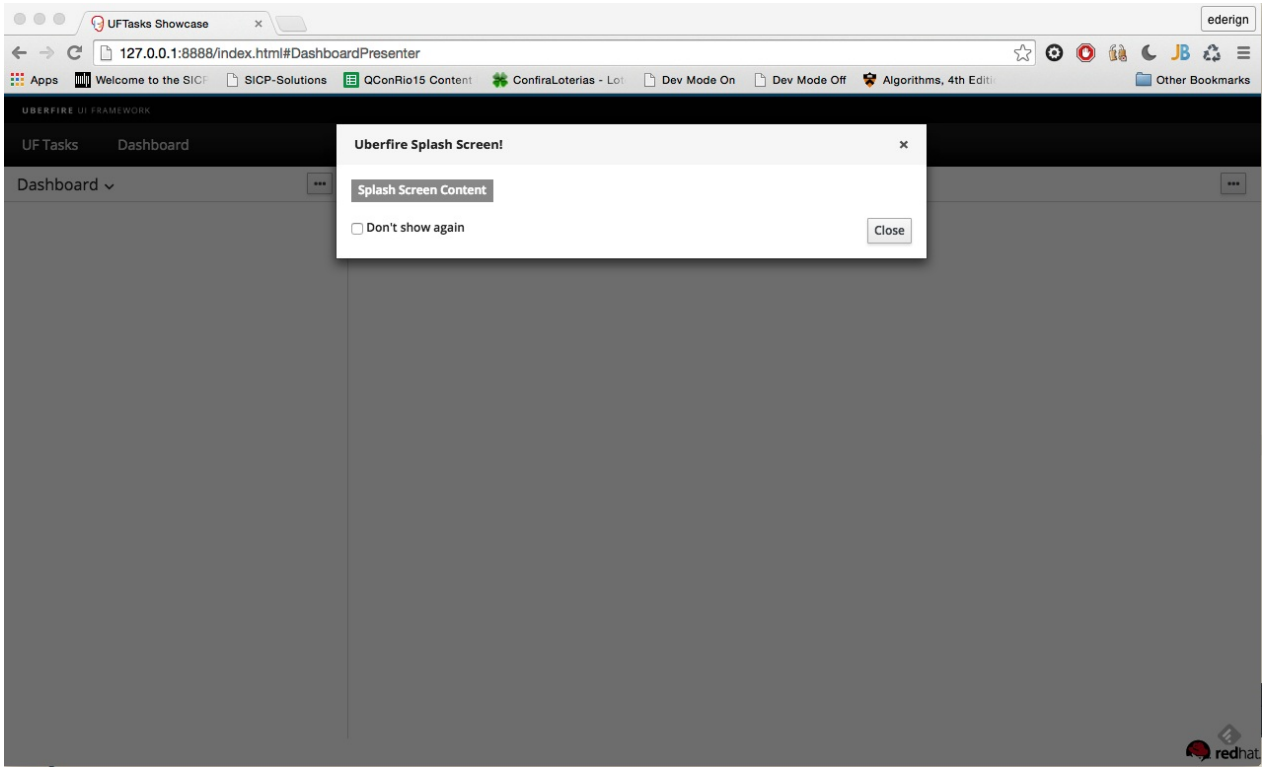
A PopUp is a part that provides a popup behavior to a view associated with a particular Uberfire Place.



SplashScreen

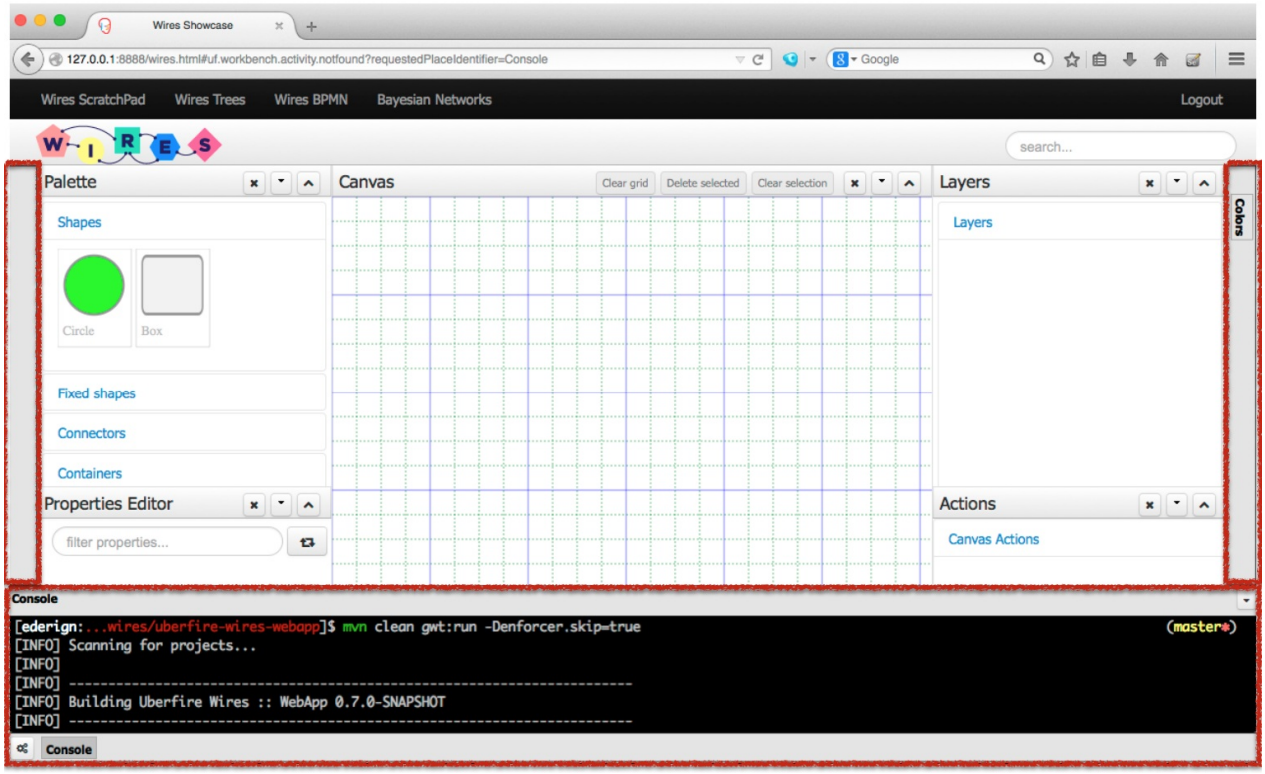
A SplashScreen is a part that provides a splash behavior to a view associated with a particular Uberfire Place.

Each SplashScreen is associated with some interception points (parts where this splash screen will be displayed).



Docks

Docks in Uberfire are special areas where you can place components (usually screens) in a dock style component. A dock item can be placed in South, West or East dock. When users select a dock item, the dock panel are expanded displaying the dock item content (like Console dock in the example).



VFS

UberFire offers a virtual file system (VFS) API that's accessible from both client-side (runs in the web browser) and server-side (runs in a Java app server) code.

The VFS has pluggable backends, and UberFire includes both a plain filesystem backend and a Git backend. This means you can write code that runs in the web browser that interacts with the contents of a filesystem directory or a Git repository: exploring, reading files, checking in changes, reviewing history, even branching and merging

Plugins

UberFire was conceived to be a high modularized framework, so basically it's possible to select whatever you want for your application.

UberFire modules are named plugins, so everything can be considered a plugin (from a simple Screen, to VFS or Security). With such approach UF allows Apps to select what they want and also create your own.

PlugIn are classified as compile time (maven module) and runtime. Most of components available are compile time, but UF also enable extensions on runtime, basically for UI components (that basically uses UFJS to extend it)

Uberfire Extensions

Uberfire extensions is a group of awesome features that that complement the Uberfire ecosystem. They are based on two github repos: Uberfire Extensions and Kie Uberfire Extensions.

They are detailed in further sections.

Uberfire Architecture

Uberfire Architecture Overview

Uberfire App Archicture

Uberfire Backend + Client

Uberfire Clustering

Client side only

Uberfire Components

Workbench

Components

Perspective

Type

Components

Panel

Components

Part

Components

Security

VFS

VFS

Uberfire Javascript API's

Uberfire has a group of JavaScript APIs that allows the extensibility of the workbench. These APIs are automatic loaded inside the workbench if they are placed in directory "plugins" of your webapp or can be executed via regular Javascript call.

This section will describe with examples the power of this API.

JS Perspectives

The register Perspective API allows dynamic creation of perspectives.

```
$registerPerspective({
    id: "Home",
    is_default: true,
    panel_type: "org.uberfire.client.workbench.panels.impl.MultiListWorkbenchPanelPresenter"
    view: {
        parts: [
            {
                place: "welcome",
                min_height: 100,
                parameters: {}
            }
        ],
        panels: [
            {
                width: 250,
                min_width: 200,
                position: "west",
                panel_type: "org.uberfire.client.workbench.panels.impl.MultiListWorkbenchPan
                parts: [
                    {
                        place: "YouTubeVideos",
                        parameters: {}
                    }
                ]
            },
            {
                position: "east",
                panel_type: "org.uberfire.client.workbench.panels.impl.MultiListWorkbenchPan
                parts: [
                    {
                        place: "TodoListScreen",
                        parameters: {}
                    }
                ]
            },
            {
                height: 400,
                position: "south",
```

```
                panel_type: "org.uberfire.client.workbench.panels.impl.MultiTabWorkbenchPane
                parts: [
                    {
                        place: "YouTubeScreen",
                        parameters: {}
                    }
                ]
            }
        ]
    }
});
```

JS Editor

The Editor API allows dynamic creation of editors and associate them with a file type.

```
$registerEditor({
    "id": "sample editor",
    "type": "editor",
    "templateUrl": "editor.html",
    "resourceType": "org.uberfire.client.workbench.type.AnyResourceType",
    "on_concurrent_update":function(){
        alert('on_concurrent_update callback')
        $vfs_readAllString(document.getElementById('filename').innerHTML, function(a) {
            document.getElementById('editor').value= a;
        });
    },
    "on_startup": function (uri) {
        $vfs_readAllString(uri, function(a) {
            alert('sample on_startup callback')
        });
    },
    "on_open":function(uri){
        $vfs_readAllString(uri, function(a) {
            document.getElementById('editor').value=a;
        });
        document.getElementById('filename').innerHTML = uri;
    }
});
```

Note that in addition to make an editor pluggable, there is some workbench events that can be hooked via JS callback after a Uberfire/workbench native event. On these editors we have two events registered: on_startup and on_open. But there are also more options of Uberfire lifecycle events:

- on_concurrent_update;
- on_concurrent_delete;
- on_concurrent_rename;

- on_concurrent_copy;
- on_rename;
- on_delete;
- on_copy;
- on_update;
- on_open;
- on_close;
- on_focus;
- on_lost_focus;
- on_may_close;
- on_startup;
- on_shutdown;

An editor is displayed via an html template, like this small example:

```
<div id="sampleEditor">
    <p>Sample JS editor (generated by editor-sample.js)</p>
    <textarea id="editor"></textarea>

    <p>Current file:</p><span id="filename"></span>
    <button id="save" type="button" onclick="$vfs_write(document.getElementById('filename').
    <br>

    <p>This button change the file content, and uberfire send a callback to the editor:</p>
    <button id="reset" type="button" onclick="$vfs_write(document.getElementById('filename')
</div>
```

JS PlaceManager

Uberfire PlaceManager API allows "go to" a specific workbench component. "Go to" means asks for the workbench display the component associated with some target.

```
$goToPlace("componentIdentifier");
```

Register Plugin API

The Register plugin API creates dynamic plugins (that will be transformed in workbench screens) via a JS API.

```
$registerPlugin({
```

```
    id: "my_angular_js",
    type: "angularjs",
    templateUrl: "angular.sample.html",
    title: function () {
        return "angular " + Math.floor(Math.random() * 10);
    },
    on_close: function () {
        alert("this is a pure JS alert!");
    }
});
```

Each plugin also has an html template.

```
File: angular.sample.html
<div ng-controller="TodoCtrl">
    <span>{{remaining()}} of {{todos.length}} remaining</span>
    [ <a href="" ng-click="archive()">archive</a> ]
    <ul class="unstyled">
        <li ng-repeat="todo in todos">
            <input type="checkbox" ng-model="todo.done">
            <span class="done-{{todo.done}}">{{todo.text}}</span>
        </li>
    </ul>
    <form ng-submit="addTodo()">
        <input type="text" ng-model="todoText" size="30" placeholder="add new todo here">
        <input class="btn-primary" type="submit" value="add">
    </form>
    <form ng-submit="goto()">
        <input type="text" ng-model="placeText" size="30" placeholder="place to go">
        <input class="btn-primary" type="submit" value="goTo">
    </form>
</div>
```

A plugin can be hooked to workbench events via a series of JS callbacks:

- on_concurrent_update;
- on_concurrent_delete;
- on_concurrent_rename;
- on_concurrent_copy;
- on_rename;
- on_delete;
- on_copy;
- on_update;
- on_open;
- on_close;
- on_focus;

- on_lost_focus;
- on_may_close;
- on_startup;
- on_shutdown;

JS Splash Screens

Splash screens can be also created via a JS API.

```
$registerSplashScreen({
    id: "home.splash",
    templateUrl: "home.splash.html",
    body_height: 325,
    title: function () {
        return "Cool Home Splash " + Math.floor(Math.random() * 10);
    },
    display_next_time: true,
    interception_points: ["Home"]
});
```

JS Virtual File System (VFS)

With this API, you can write and read a file saved in the file system with an asynchronous call.

```
$vfs_readAllString(uri,  function(a) {
           //callback logic
        });

    $vfs_write(uri,content,  function(a) {
        //callback logic
     })
```

Angular Integration

App Styling

Bootstrap

Patternfly

Uberfire Extensions

Uberfire extensions is a group of awesome features that that complement the Uberfire ecosystem. They are based on two github repos: Uberfire Extensions and Kie Uberfire Extensions.

Wires

Widgets

Ace Editor

MarkDown

Property Editor

Security

Metadata

Automatic VFS Index/Search

Self Service App

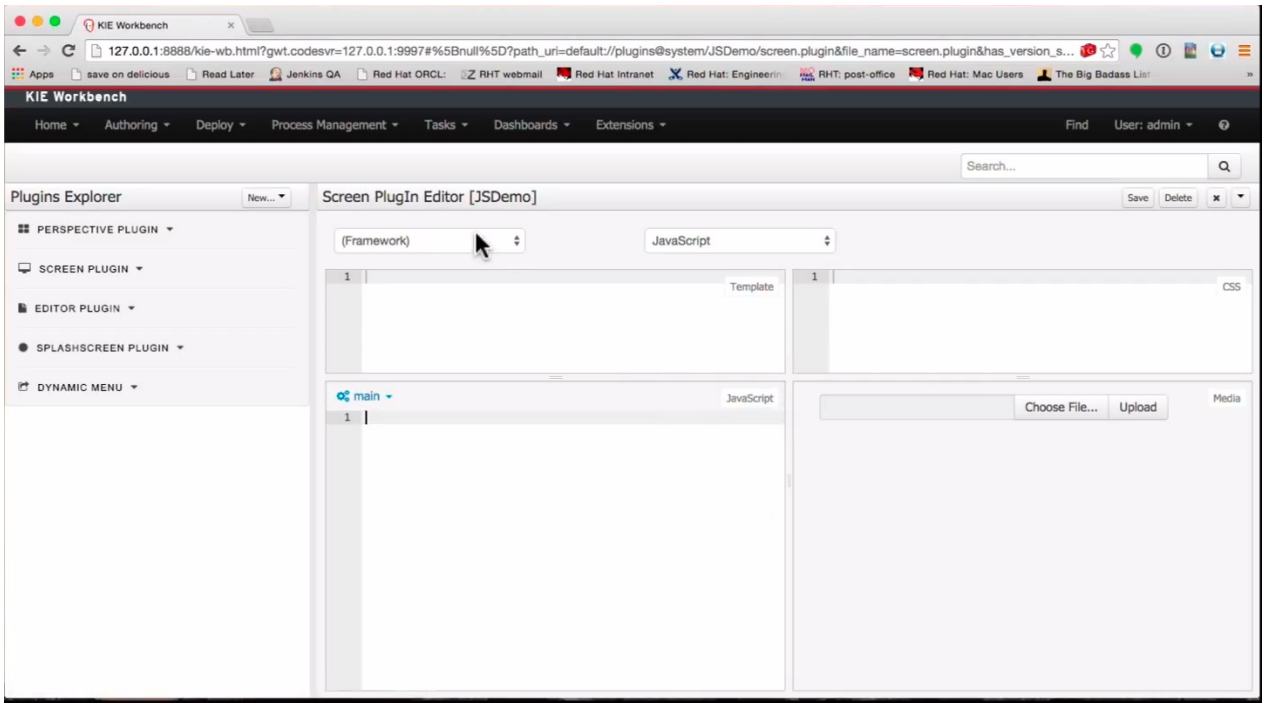
Uberfire RAD Extensions

Uberfire RAD extensions or configurable workbench is group of features divided in four main features: Plugin Management, Perspective Editor, Apps and JS APIs.

Plugin Management

Uberfire can be extensible by JS and HTML in web app, but this is sometboth powerful and complex.

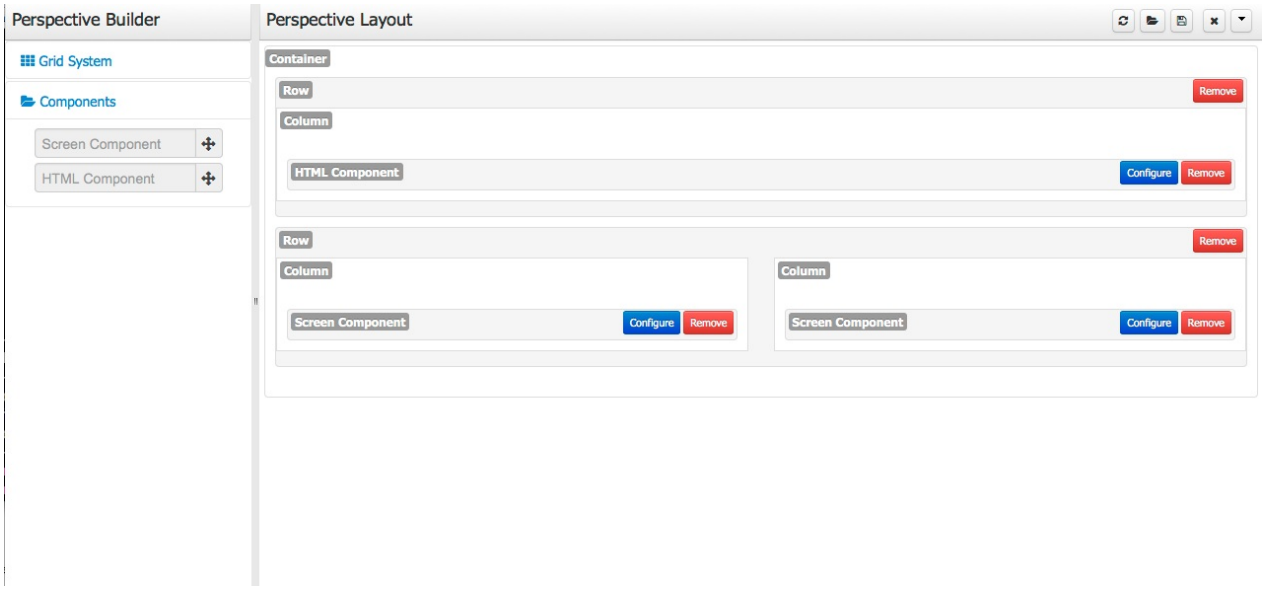
Plugin Management comes to solve this issue, making our users able to create new perspectives, screens, editors, splash screens and menus via an UI inspired by JSFiddle.



All the UI features are based on JS APIs detailed in previous sections.

Perspective Editor

Based on Bootstrap grid system, we also created a new dynamic grid builder, were our users will define their own perspectives.



Apps

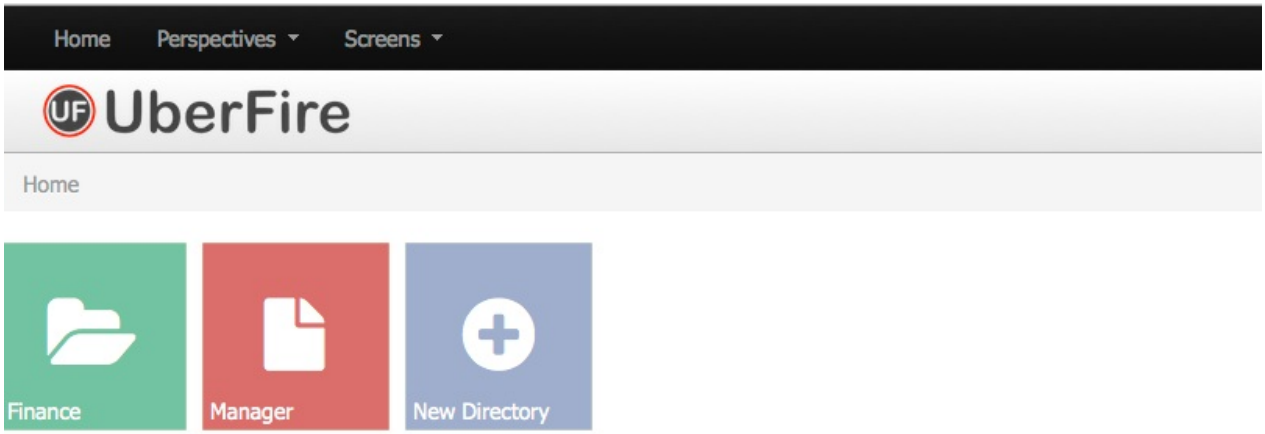
Uberfire Apps will be detailed in the next section.

There is also a a video showing this features working with DashBuilder.

App Directory

Part of Uberfire Configurable Workbench, Apps directory is a way to organize your own components, instead, or as well as, top menu entries.

When you save a new perspective, you can add labels for them and these labels are used to associate a perspective with an App's directory.



Here is a video showing our app builder working with DashBuilder.

Third Part Extensions

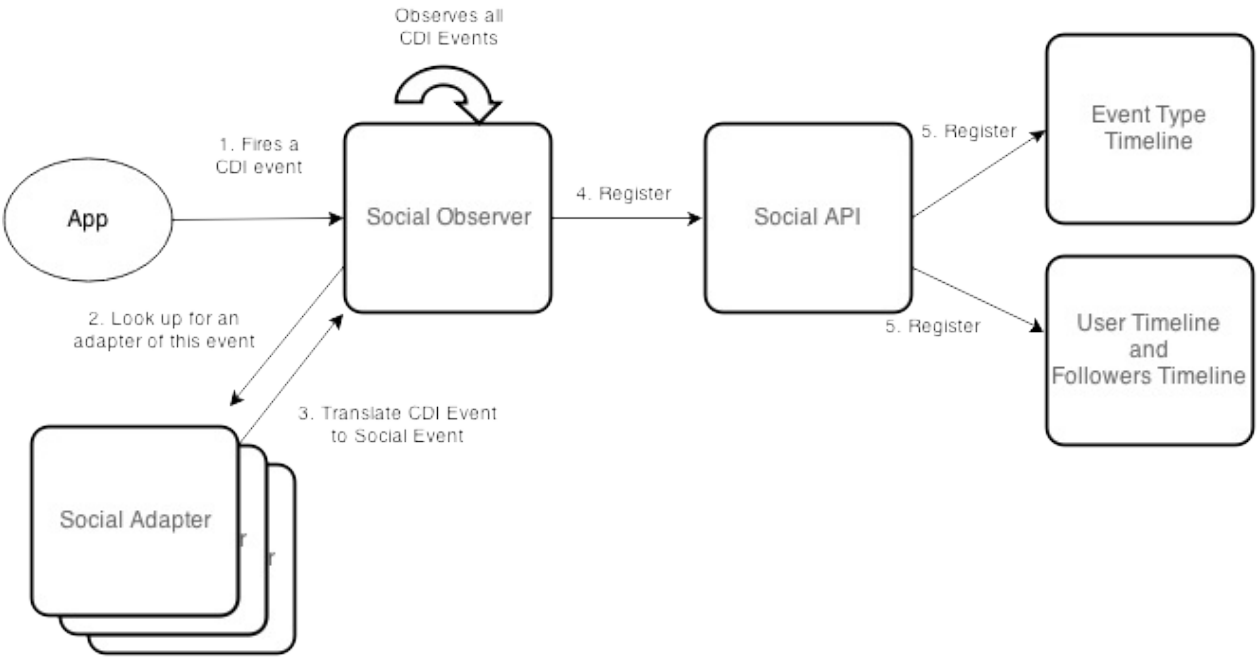
Uberfire Social Activities

Uberfire has a extension called Kie Uberfire Social Activities that provides an extensible architecture to capture, handle, and present (in a timeline style) configurable types of social events.

Basic Architecture

An event is any type of "CDI Event" and will be handled by their respective adapter. The adapter is a CDI Managed Bean, which implements SocialAdapter interface. The main responsibility of the adapter is to translate from a CDI event to a Social Event. This social event will be captured and persisted by Kie Uberfire Social Activities in their respectives timelines (basically user and type timeline).

That is the basic architecture and workflow of this tech:

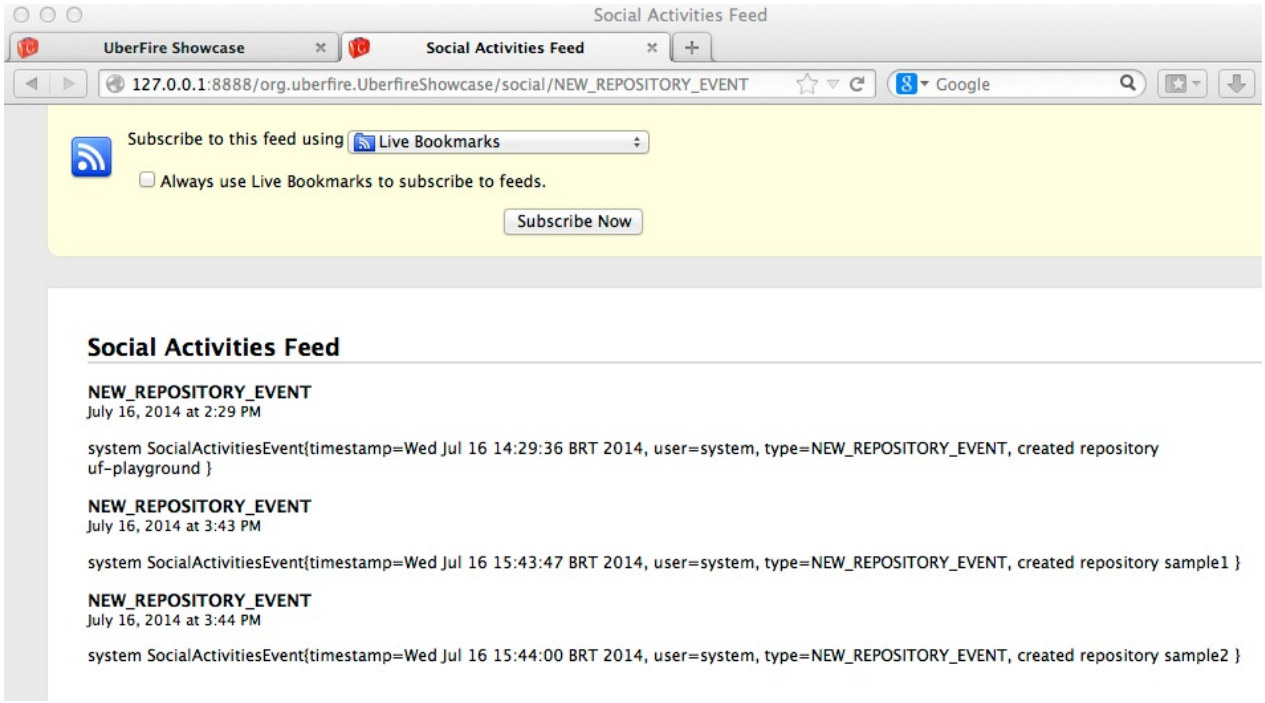


Timelines

There is many ways of interact and display a timeline. This session will briefly describe each one of them.

- Atom URL

Social Activities provides a custom URL for each event type. This url is accessible by: http://project/social/TYPE_NAME.

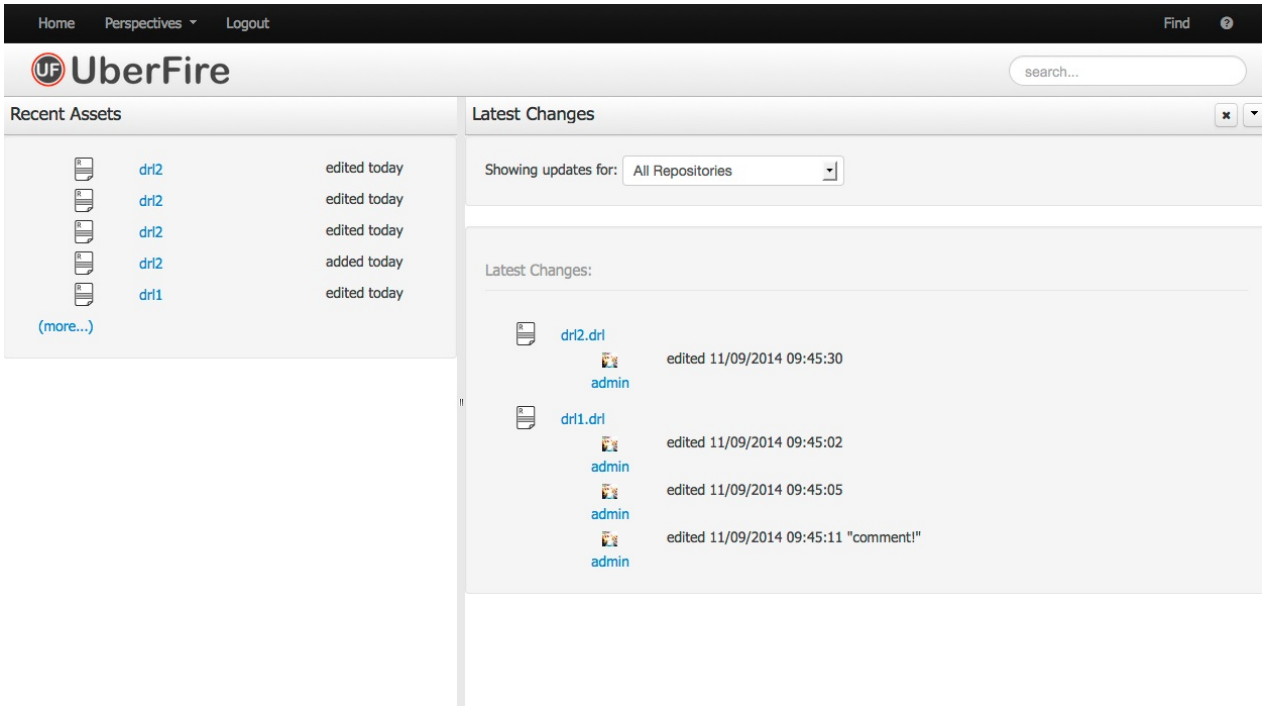


The users timeline works on the same way, being accessible by http://project/social-user/USER_NAME .

Another cool stuff is that an adapter can provide his pluggable url-filters. Implementing the method getTimelineFilters from SocialAdapter interface, he can do anything that he want with his timeline. This filters is accessible by a query parameter, i.e. http://project/social/TYPE_NAME?max-results=1 .

Basic Widgets

Social Activities also includes some basic (extendable) widgets. There is two type of timelines widgets: simple and regular widgets. You can see a example of the usage of these widgets on Recent Asssets (Simple) and Latest Changes widget.



The "more..," symbol on 'Simple Widget' is a pagination component. You can configure it by an easy API. With an object SocialPaged(2) you creates a pagination with 2 items size. This object helps you to customize your widgets using the methods canIGoBackward() and canIGoForward() to display icons, and forward() and backward() to set the navigation direction.

The Social Activities component has an initial support for avatar. In case you provide an user e-mail for the API, the gravatar image will be displayed in this widgets.

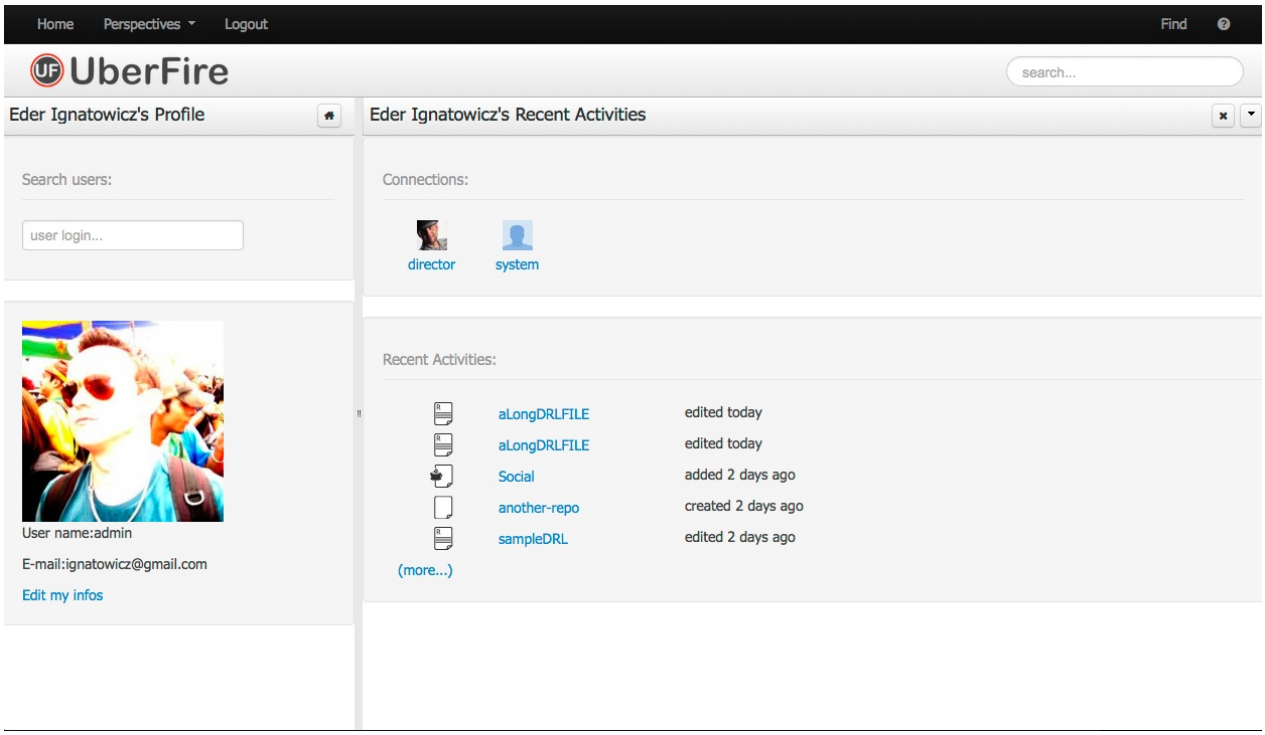
Drools Query API

Another way to interact with a timeline is throught the Social Timeline Drools Query API. This API executes one or more DRLs in a Timeline in all cached events. It's a great way to merge different types of timelines.

```
package org.uberfire.social.activities.drools
import org.uberfire.social.activities.model.SocialActivitiesEvent;

global org.uberfire.social.activities.drools.SocialTimelineRulesQuery queryAPI;
global java.util.List socialEvents;
rule "Get Type Cached"
when
  $result:SocialActivitiesEvent() from
queryAPI.getTypeCached("FOLLOW_USER","NEW_REPOSITORY")
  eval( false )  // put any condition here
then
  socialEvents.add($result);
end
```

User Profile, Followers/Following Social Users

Each user has his own profile. A user can follow another social user. When a user generates a social event, this event is replicated in all timelines of his followers. Social also provides a basic widget to follow another user, show all social users and display a user following list.

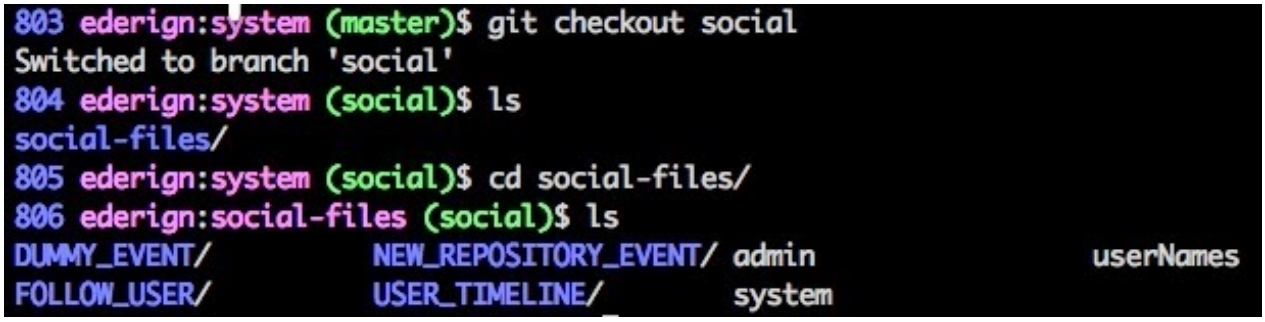


A user can follow another social user. When a user generates a social event, this event is replicated in all timelines of his followers. Social also provides a basic widget to follow another user, show all social users and display a user following list.

Persistence Architecture

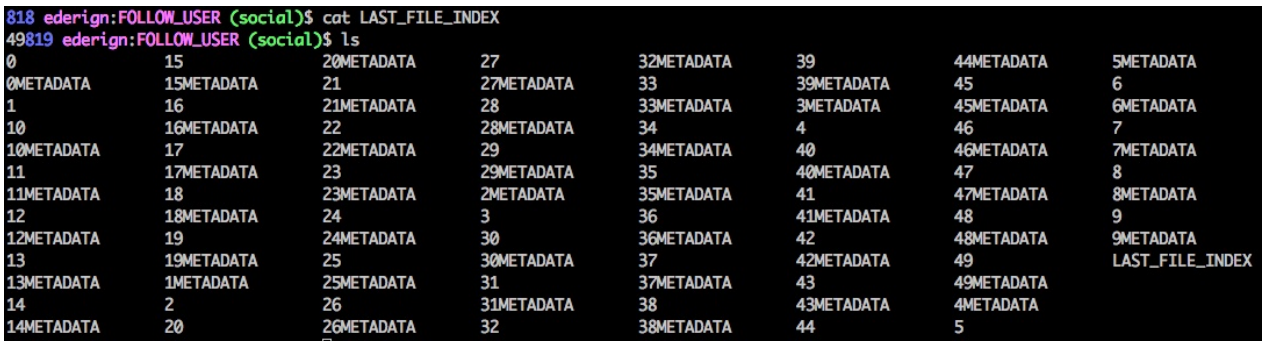
The persistence architecture of Social Activities is build on two concepts: Local Cache and File Persistence. The local cache is a in memory cache that holds all recent social events. These events are kept only in this cache until the max events threshold is reached. The size of this threshold is configured by a system property org.uberfire.social.threshold (default value 100).

When the threshold is reached, the social persist the current cache into the file system (system.git repository - social branch). Inside this branch there is a social-files directory and this structure:



- userNames: file that contains all social users name
- each user has his own file (with his name), that contains a Json with user data.
- a directory for each social type event .
- a directory "USER_TIMELINE" that contains specific user timelines

Each directory keeps a file "LAST_FILE_INDEX" that point for the most recent timeline file.



Inside each file, there is a persisted list of Social Events in JSON format:

```
({"timestamp":"Jul16,2014,5:04:13PM","socialUser":{"name":"stress1","followersName":[],"fo
```

Separating each JSONs there is a HEX and the size in bytes of the JSON. The file is read by social in reverse order.

The METADATA file current hold only the number of social events on that file (used for pagination support).

It is important to mention that this whole structure is transparent to the widgets and pagination. All the file structure and respective cache are MERGED to compose a timeline.

Clustering

In case that your application is using Uberfire in a cluster environment, Kie Social Activities also supports distributed persistence. His cluster sync is build on top of UberfireCluster support (Apache Zookeeper and Apache Helix).

Each node broadcast social events to the cluster via a cluster message SocialClusterMessage.NEW_EVENT containing Social Event data. With this message, all the nodes receive the event and can store it on their own local cache. In that point all nodes caches are consistent.

When a cache from a node reaches the threshold, it lock the filesystem to persist his cache on filesystem. Then the node sends a SOCIAL_FILE_SYSTEM_PERSISTENCE message to the cluster notifying all the nodes that the cache is persisted on filesystem.

If during this persistence process, any node receives a new event, this stale event is merged during this sync.

Roadmap

This is an early version of Kie Uberfire Social Activities. In the nexts versions we plan to provide:

A "Notification Center" tool, inspired by OSX notification tool; (far term)

Integrate this project with dashbuilder KPI's;(far term)

A purge tool, able to move old events from filesystem to another persistence store; (short term)

In this version, we only provide basic widgets. We need to create a way to allow to use customized templates on this widgets.(near term)

A dashboard to group multiple social widgets.(near term)

If you want start contributing to Open Source, this is a nice opportunity. Fell free to contact us!

Guvnor

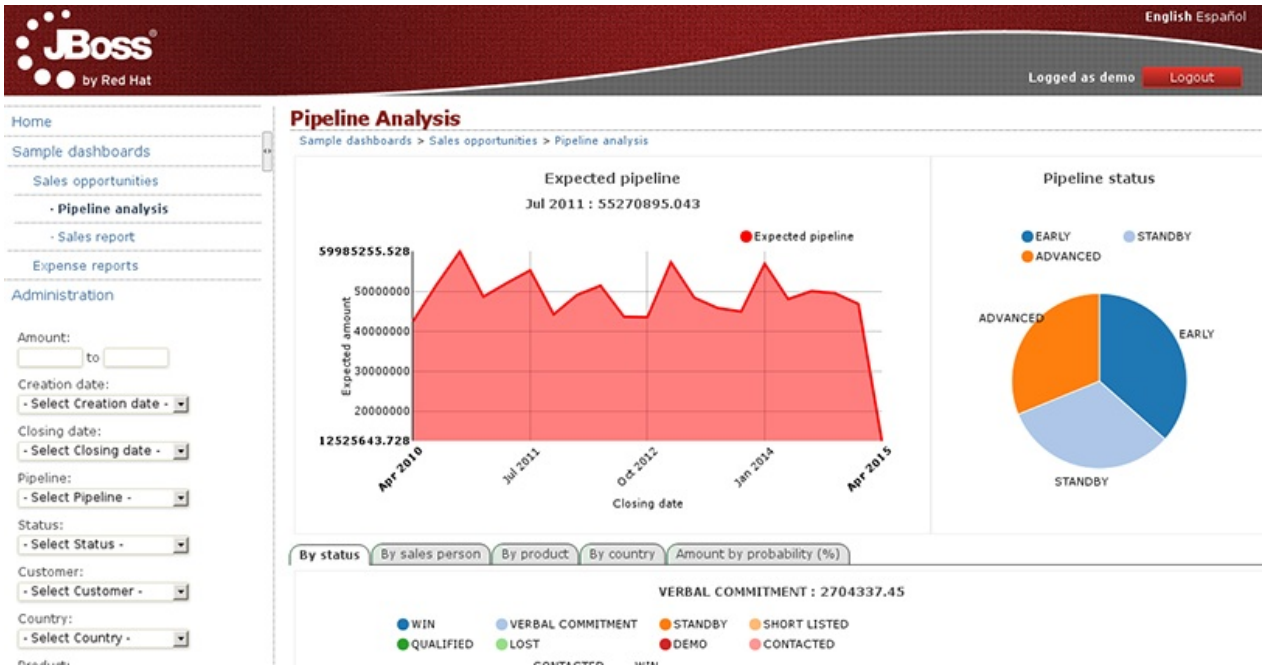
Dashbuilder

Based on Uberfire, Dashbuilder is a full featured web application which allows non-technical users to visually create business dashboards.

Dashboard data can be extracted from heterogeneous sources of information such as JDBC databases or regular text files.

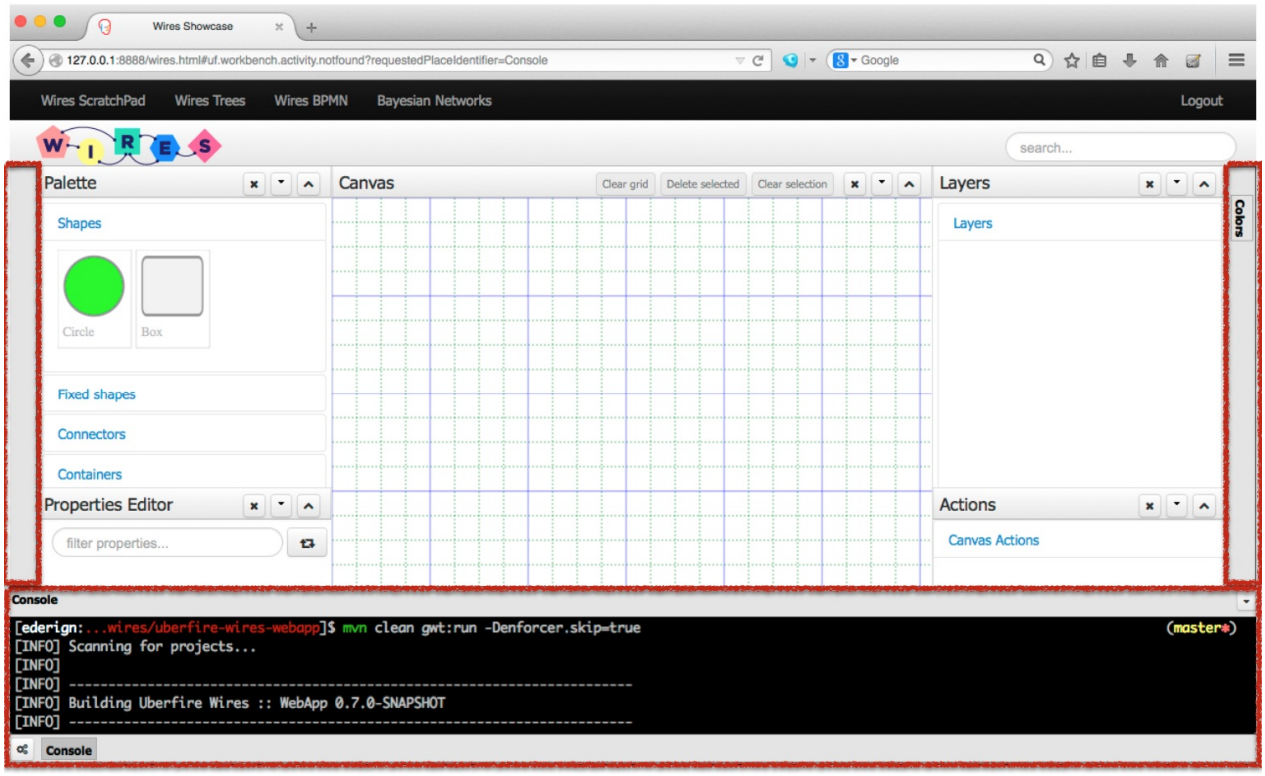
Key features

- Visual configuration of dashboards with Drag'n'drop.
- Inline creation and editing of KPIs (Key Performance Indicators).
- Interactive report tables.
- Data export to Excel and CSV format.
- Connectors to existing systems with standard protocols.
- Fine-grained access control for different users and roles.
- Look'n'feel customization tools.
- Integration the jBPM (Business Process Management) platform.
- Import and export.
- Ready for deployment.



Simple Docks

Simple Docks is an implementation of Uberfire Docks support.



In order to create a dock using Uberfire Extensions Simple Dock, you should add the dependency to your pom.xml and gwt.xml.

pom.xml

```
    <dependency>
      <groupId>org.uberfire</groupId>
      <artifactId>uberfire-simple-docks-client</artifactId>
      <version>0.7.0-SNAPSHOT</version>
    </dependency>
    ...
     <compileSourcesArtifact>org.uberfire:uberfire-simple-docks-client</compileSourcesArtifa
```

gwt.xml

```
    <inherits name="org.uberfire.UberfireDocksClient"/>
```

The Uberfire Docks is a CDI bean. So you should inject it in your web app:

```
@Inject
UberfireDocks docks;
```

Creating docks

A UberfireDock has two parameters: a Position (SOUTH, WEST, EAST) and a Place Request associated with the dock (usually a Uberfire Screen). The third optional constructor parameter is the perspective associated with the dock.

```
UberfireDock dock = new UberfireDock(UberfireDockPosition.EAST,
    new DefaultPlaceRequest("bla2"), "WiresScratchPadPerspective");
```

All the available docks are displayed in the available docks button (left side of south dock).

You also can set a different size of the dock open. You can define the size of a dock:

```
dock.withSize(500.0);
```

Registering docks

In order to register a dock, callregister method:

```
docks.register(dock);
```

There is also a way to temporarily disable/enable a dock inside a perspective:

```
docks.disable(UberfireDockPosition.EAST,"WiresScratchPadPerspective");
docks.enable(UberfireDockPosition.EAST,"WiresScratchPadPerspective");
```

Integration

From Angular

To Angular

Deploy on Tomcat

This guide will help you install an UberFire demo application on your own computer. This will help let you try out an UberFire application in Tomcat, and prove the UberFire apps work with your setup.

Building our App

This demo will be based on UF tasks app from previous section. If you didn't work on this App, fell download it and build it from source code:

```
git clone https://github.com/uberfire/uberfire-tutorial.git
cd uftasks
mvn clean install
```

Get an app server

UFTasks was generated from Uberfire Archetype. So inside uftasks-showcase, there is a directory called uftasks-distributions-wars that has inside target directory, WAR files for JBoss EAP 6.4, Tomcar 7.0 and Wildfly 8.1. Let's install this app on Tomcat 7.0

Get an app server

If you don't already have Tomcat 7.0 installed on your computer, you can download and instalk it. Installing is as easy as downloading and unzipping.

Start the app server

Now start the app server using a command line terminal. Use the cd command to change the working directory of your terminal to the place where you unzipped the application server, then execute one of the following commands, based on your operating system and choice of app server:

*nix, Mac OS X	Windows

bin/startup.sh	bin\startup.bat

Then visit the URL http://localhost:8080/ and you should see a webpage confirming that the app server is running.

Deploy the WAR

Rename the UFtasks tomcat WAR file to uftasks.war and copy it into the auto-deployment directory for your app server. In example on Unix/Linux/Mac:

```
mv uftasks-showcase-1.0-SNAPSHOT-tomcat7.0.war /~/bin/apache-tomcat-7.0.57/webapps/uftasks.w
```

See it work!

Now visit http://localhost:8080/uftasks/ and sign in with username admin, password admin.

Deploy on Wildfly

This guide will help you install an UberFire demo application on your own computer. This will help let you try out an UberFire application in Wildfly, and prove the UberFire apps work with your setup.

Building our App

This demo will be based on UF tasks app from previous section. If you didn't work on this App, fell download it and build it from source code:

```
git clone https://github.com/uberfire/uberfire-tutorial.git
cd uftasks
mvn clean install
```

Get an app server

UFTasks was generated from Uberfire Archetype. So inside uftasks-showcase, there is a directory called uftasks-distributions-wars that has inside target directory, WAR files for JBoss EAP 6.4, Tomcar 7.0 and Wildfly 8.1. Let's install this app on Tomcat 7.0

Get an app server

If you don't already have Wildfly 8.1 installed on your computer, you can download and install it. Installing is as easy as downloading and unzipping.

Start the app server

Now start the app server using a command line terminal. Use the cd command to change the working directory of your terminal to the place where you unzipped the application server, then execute one of the following commands, based on your operating system and choice of app server:

*nix, Mac OS X	Windows

bin/standalone.sh	bin\standalone.bat

Then visit the URL http://localhost:8080/ and you should see a webpage confirming that the app server is running.

Deploy the WAR

Rename the UFtasks wildfly WAR file to uftasks.war and copy it into the auto-deployment directory for your app server. In example on Unix/Linux/Mac:

```
mv uftasks-showcase-1.0-SNAPSHOT-wildfly8.1.war /~/bin/wildfly-8.1.0.Final/standalone/deploy
```

See it work!

Now visit http://localhost:8080/uftasks/ and sign in with username admin, password admin.

WAS

Web logic

EAP

Clustering

Who Uses Uberfire

jBPM

Drools Workbench

Troubleshooting & FAQ

This section explains the cause of and solution to some common problems that people encounter when building applications with Uberfire.

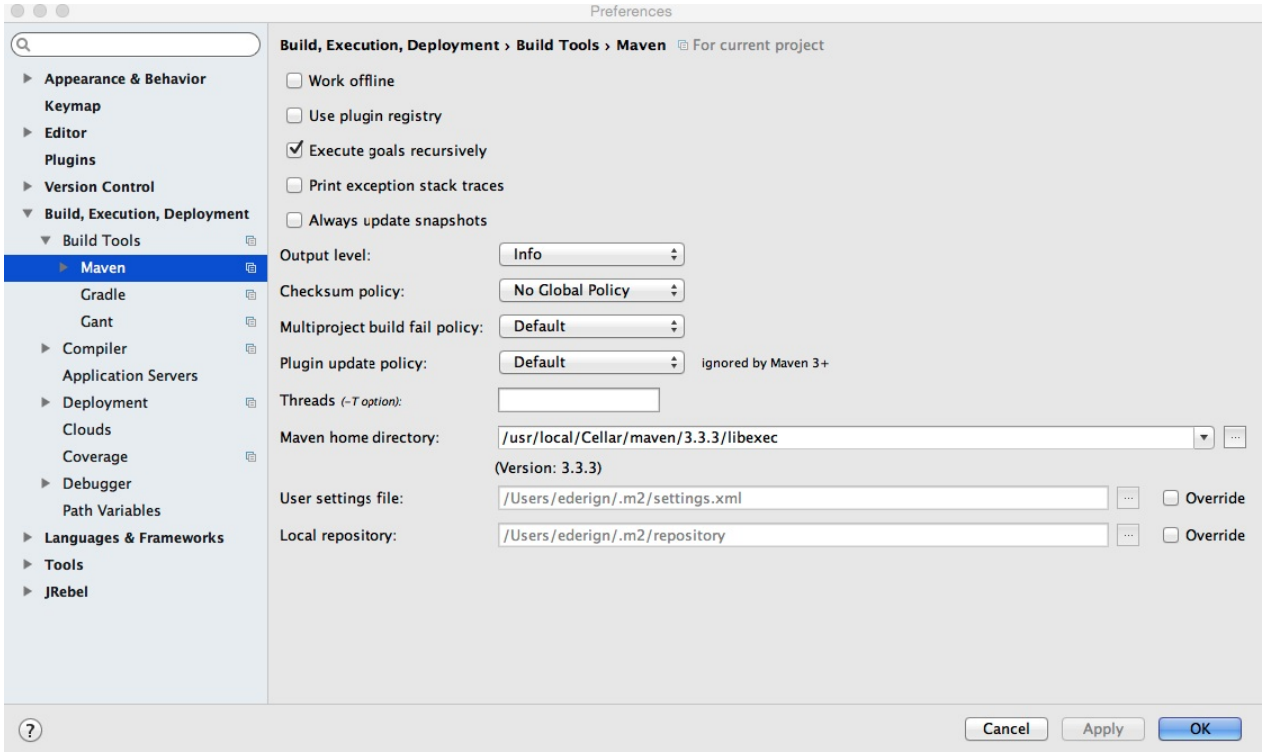
Of course, when lots of people trip over the same problem, it’s probably because there is a deficiency in the framework! A FAQ list like this is just a band-aid solution. If you have suggestions for permanent fixes to these problems, please get in touch with us: file an issue in our issue tracker, chat with us on IRC, or post a suggestion on our forum.

But for now, on to the FAQ:

1-) After import my Uberfire App in IntelliJ IDEA, the GWT module are not displayed.

Possible symptoms: build of your web-app was broken on IDEA, and it is running on command line.

Answer: Uberfire requires Maven 3.3. Make sure that your IDEA is using Maven 3.3.



How to Contribute

UberFire is an open-source, community-driven project and we'd love your help! Here's how:

- Help to improve this docs;
- Report (or fix!) a bug;
- Hang out in #uberfire on Freenode;
- Ask and answer questions in our discussion forum;
- Improve this website;
- Build from Source;

Build Uberfire from Source

```
$ git clone git@github.com:uberfire/uberfire.git
$ cd uberfire
$ mvn clean install
```