Application Setup Guide

Contents

[1 First-time IDE and DB Setup 1](#_Toc414694701)

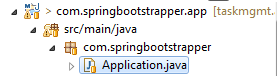
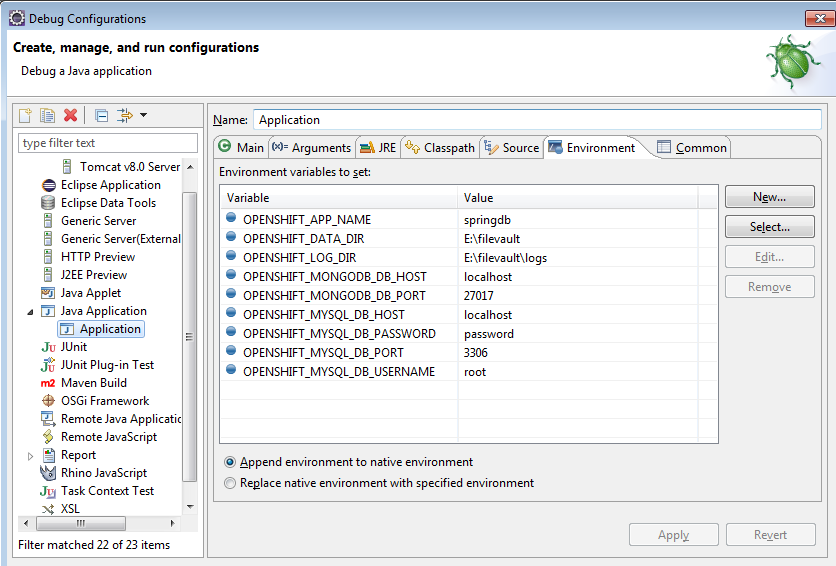
[2 Defining New Application Modules 2](#_Toc414694702)

[2.1 Defining Entities and Relationships on Server Side 3](#_Toc414694703)

[2.2 Defining UI Meta Data for Client-Side Rendering 4](#_Toc414694704)

[2.3 Defining UI Modules 5](#_Toc414694705)

# First-time IDE and DB Setup

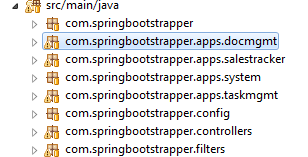
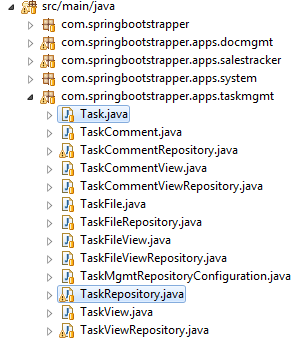
1. Application depends on MongoDB server, MySql server and a File-vault which is nothing but a folder location where uploaded files get stored.
2. MongoDB server can be downloaded from [here](https://www.mongodb.org/downloads). This is used for storing NoSql type entities. There is no need to setup authentication on the mongo-db, just run it after downloading by ensuring that the service that gets installed shows running in the service-manager.  
     
   MongoDb stores JSON documents in database->collections hierarchy. This sort of relates to database->schema notion of RDBMS type databases. If you like TOAD like admin interface to see what gets stored in the MongoDB, RoboMongo is an excellent application that can be downloaded from [here](http://robomongo.org/download.html).
3. MySql server can be downloaded from [here](http://dev.mysql.com/downloads/mysql/). MySql workbench, which is again, TOAD like admin interface that is useful for peeking inside MySql and administering it, can be downloaded from [here](http://dev.mysql.com/downloads/workbench/).
4. For file-vault, just designate some folder on your file-system, i.e., E:\filevault and make sure that is configured as the file-vault.
5. The application project is basically Java project based on Maven, so importing it as Existing Maven project should automatically download all the dependencies that are defined in pom.xml file.
6. The main-file of the application is Application.java located under com.springbootsrapper project.  
   
7. Running the main file will launch embedded tomcat server and application should then be accessible at localhost:8080 address. Ensure that no other process is running on port 8080.
8. When running the main file, application will try to connect to MySql and MongoDB databases based on what you configured in application.properties file. Notice that the application.properties file uses certain environment variables to configure specific mysql and mongodb databases, this is required because OpenShift.com platform requires us to not use hard-coded values for the database-connectivity, everything must be configured via environment variables. So, to make it consistent, you’d need to setup environment variables in the Run configuration of the main Application.java launcher as shown in the following image:
9. Notice that this is where the mysql as well as mongodb parameters are configured.

# Defining New Application Modules

This is two part process primarily.

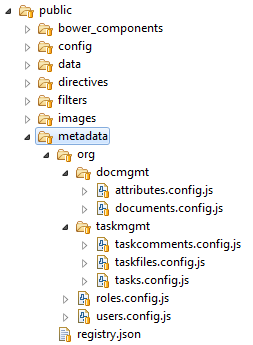
1. You’d need to define the REST backend by writing entity-relationships on the server side. You DO NOT need to create MySql or MongoDB schema to store the data, this will be done by spring automatically. Once server-back-end is running, ensure that you are able to validate REST backend with [Chrome PostMan plugin](https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm?hl=en) by performing CRUD operations using the PostMan plugin.
2. Second part is defining the UI related meta-data on the client-side. This is described in details below. This tells client how to render the data coming via REST end points.
3. Third and final part is to write the module by assembling the available widgets and writing bit of wiring code – which should be very minimal most of the times. This is where AngularJS knowledge will be required, studying existing modules should help understand how the wiring is done.

## Defining Entities and Relationships on Server Side

1. Create application specific folder parallel to existing folders. For example: diy\src\main\java\com\springbootstrapper\apps\**mynewapp**
2. Define entity relationships using JPA specification. It will help to look at how taskmgmt app’s entities are defined. For each entity that you intend to expose over REST end-point, you’d also need to define Repository interface.
3. The following image shows taskmgmt app’s entities.  
   
4. After you are done defining the entities and relationships, as well as repositories, you should be able to run the app and validate that the end points are working from Chrome post-man plug-in. Tables will be autogenerated in underlying mysql database, you should be able to confirm this using MySql workbench like tools(or, phpMyAdmin)
5. After confirming the REST end-points, swtich gears to define meta data on the client-side.

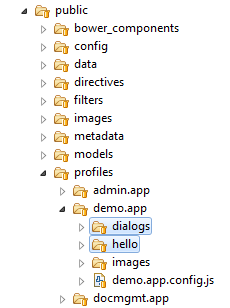
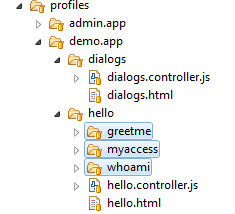
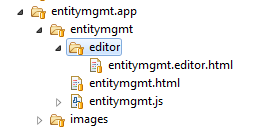
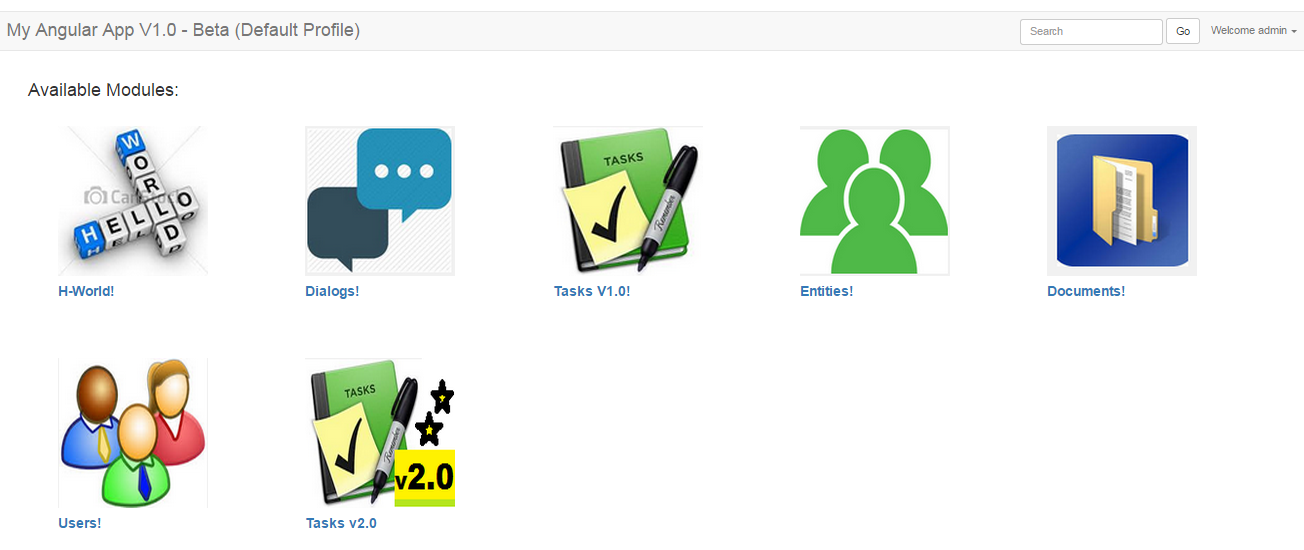
## Defining UI Meta Data for Client-Side Rendering

1. There is public/metadata folder where all the metadata for each of the entities that we defined on the server-side is defined for the client-side visualization.



1. The meta-data files for each type of entity is organized in java-class/packages-like hierarchy. The registry.json provides single file that lists down all entities defined by the system.
2. For each “.config.js” file, there will be an entry in registry.js file. The package-like hierarchy makes it possible to refer to the meta-data files by java-class-name syntax. For example, org.taskmgmt.tasks meta data contains the all the meta-data about tasks entities.
3. Each meta-data file in turn defines some basic things about the entity. It primarily defines editor and list-view related meta-data for the entity. Additionally some meta-data files also define custom actions (like ‘markDone’ action for tasks!) related code.
4. The editor part of the meta-data can be perceived like a table definition that helps UI render the editor widget and it tells UI where to post data to store/retrieve the entity.
5. The listview part of meta-data tells the UI how to render the grid-view for entity. Grid view may come from a ‘view’ based REST-entity defined on the server-side.
6. Copying existing application and start modifying it may be easier approach to define new modules that expose CRUD behaviour for newly defined entities on the server-side.
7. For any application programmer using this platform, server-side JPA entities, public/metadata and the profiles/xyz.app folder should be the only three places where all the work needs to be done.

## Defining UI Modules

1. diy\public\profiles contains various folders, which represent the client-side applications. Each sub-folder is an application that contains one or more modules. For example, the demo.app application folder has two modules, namely, ‘dialogs’ and ‘hello’. It can also have some other common folders such as images, templates etc. The ‘dialog’ and ‘hello’ folders represent the two modules exposed by the demo.app application.  
     
   
2. So, it would make sense to create an xyz.app folder (i.e., for xyz application), and then create another ‘xyz’ folder underneath it, which will basically represent default ‘module’ of xyz app.
3. The hierarchy is, **application -> module -> actions**. You can see below that hello module of hello.app application has three folders that have code for each of the three actions that the hello module has.  
   
4. Here is example of entitymgmt.app application, which has primary module called ‘entitymgmt’, which in turn has one ‘action’ called ‘editor’. You’d typically see xyz.js file for xyz module, i.e., entitymgmt.js has controller code for entitymgmt module, the .html file then has the actual view template.  
   
5. It may be easier to study or copy-over existing xyz.app folder and then start re-naming it to whatever app name you want to give.
6. The modules in the “application->modules->actions” hierarchy are the components that show up on the UI when you log-into the app.  
     
   
7. For authoring the UI template, some directives (i.e., widgets) can be used, these are defined at public/directives folder. entitymgmt.html file makes use of grid and editor widgets. The more elaborate documentation will need to be written to cover each UI widget that can be integrated while writing new applications.

## 