Contents

[1. Create new Maven Project 2](#_Toc447393145)

[2. Create a new Tomcat Server 6](#_Toc447393146)

[3. Getting rid of javax.servlet.http.HttpServlet jsp problem 20](#_Toc447393147)

[3.1. Method A 21](#_Toc447393148)

[3.2. Method B 22](#_Toc447393149)

[4. Configuring our maven project 23](#_Toc447393150)

[4.1. Configuring pom.xml 23](#_Toc447393151)

[4.2. Configuring web.xml 23](#_Toc447393152)

[4.3. Adding our beans configuration file 23](#_Toc447393153)

[4.4. Adding source (java) folder 24](#_Toc447393154)

[4.5. Add your first package and first class 24](#_Toc447393155)

[4.6. Publish your application to the server 26](#_Toc447393156)

[5. Import a maven project 27](#_Toc447393157)

[6. Add User functionality 29](#_Toc447393158)

[7. Configurate Hibernate & JPA 38](#_Toc447393159)

[7.1. Pom.xml – updates 38](#_Toc447393160)

[7.2. Configuring application to use DB (db.properties file) 38](#_Toc447393161)

[7.3. Update applicationContext.xml 38](#_Toc447393162)

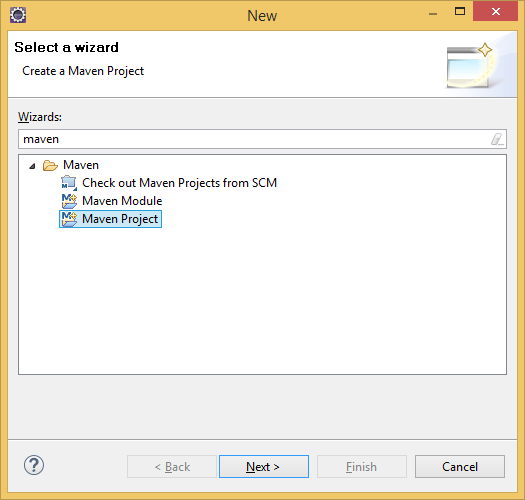
[7.3.1. Add application configuration in order to use the external config file 38](#_Toc447393163)

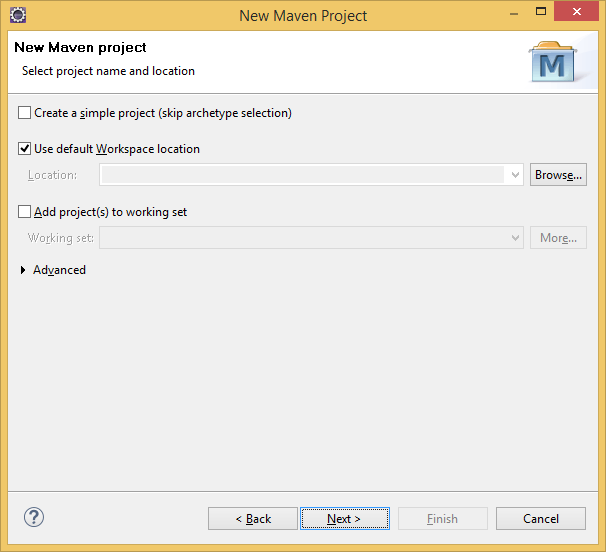
[7.3.2. Configuring data source (connection to DB) 39](#_Toc447393164)

[7.3.3. Configuring the Entity Manager 39](#_Toc447393165)

[7.3.4. Configuring the Transaction Manager 39](#_Toc447393166)

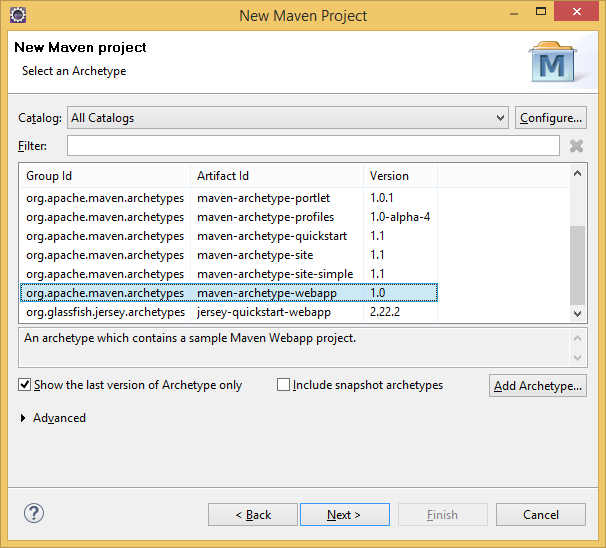
# Create new Maven Project



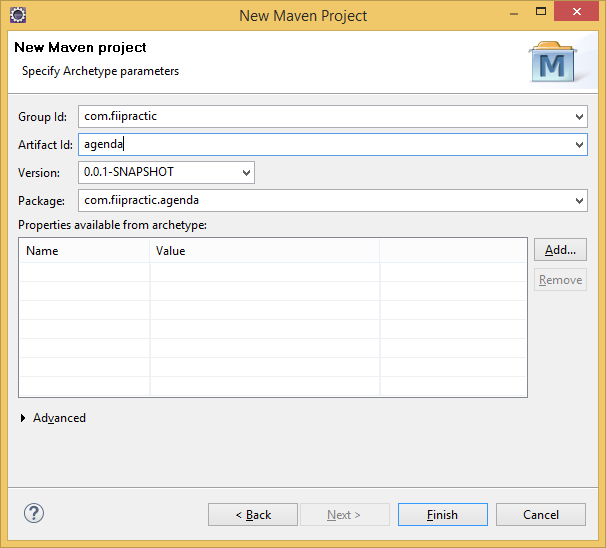
Next  


Next

Select artifact Id: maven-archetype-webapp (version 1.0)



Fill project details:

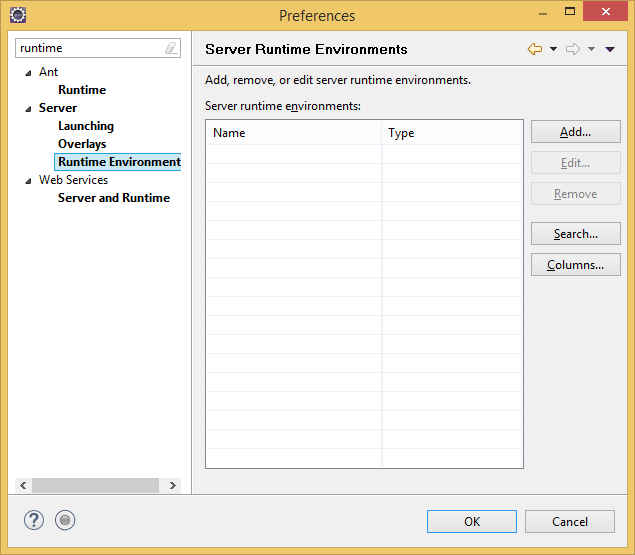


Group Id: com.fiipractic  
Artefact Id: agenda

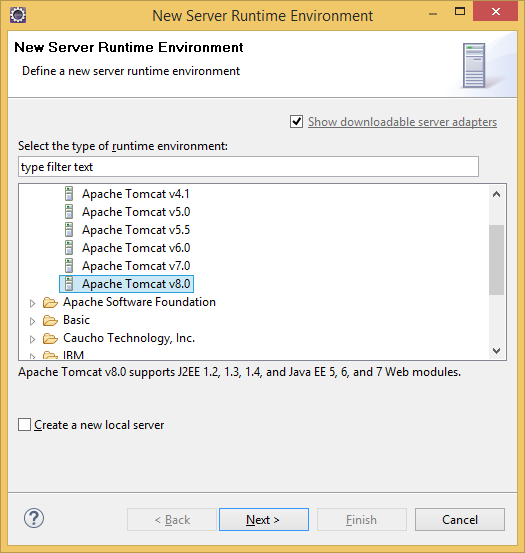
And click Finish ☺

# Create a new Tomcat Server

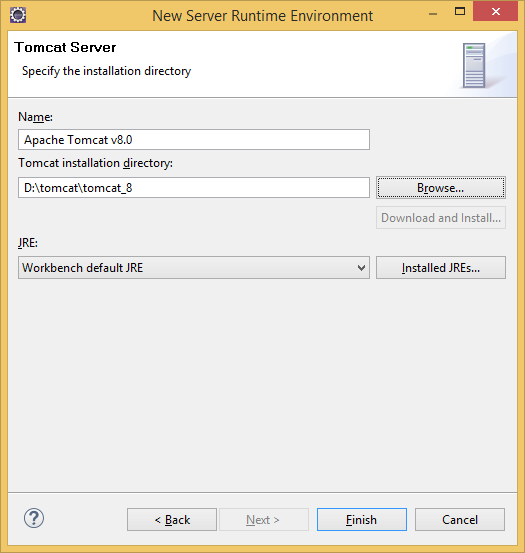
Go to Window -> Preferences and write runtime in ‘type filter text’ box, and go to Server -> Runtime Environment as below:



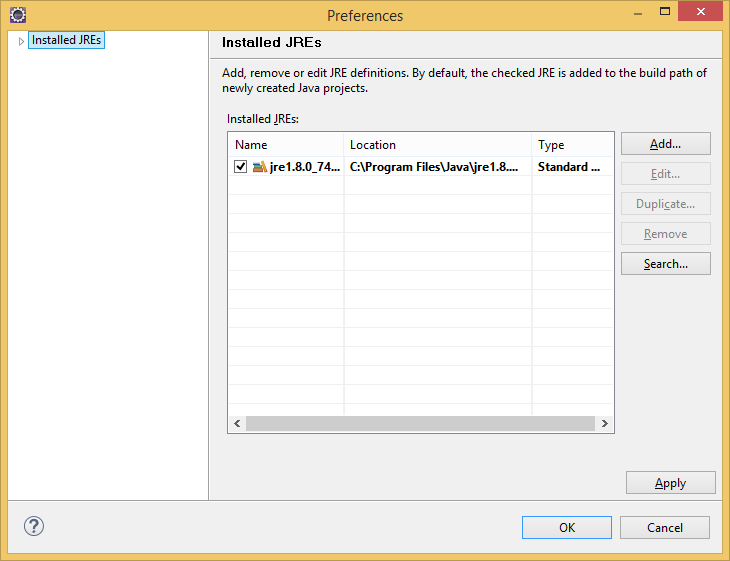
Click on Add and select Apache Tomcat v8.0:



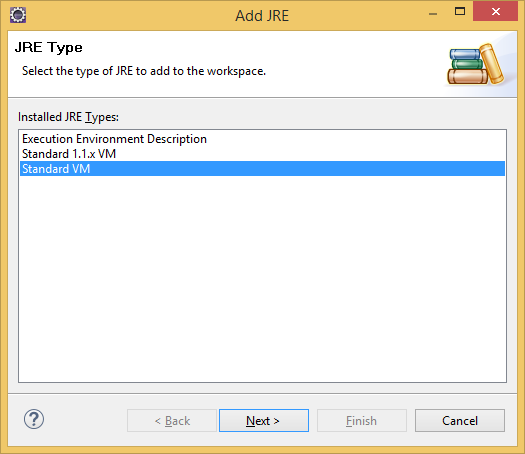
On the next screen browser for your server disk location:



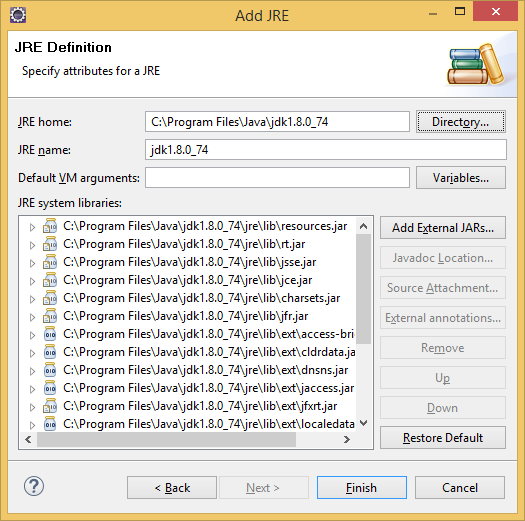
Update the JREs you have on your workspace click on Installed JREs:

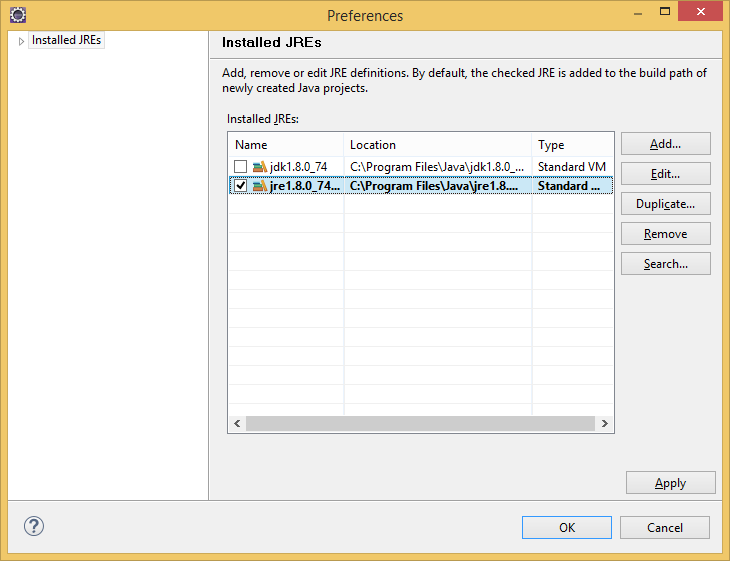


Click on Add and select Standard VM



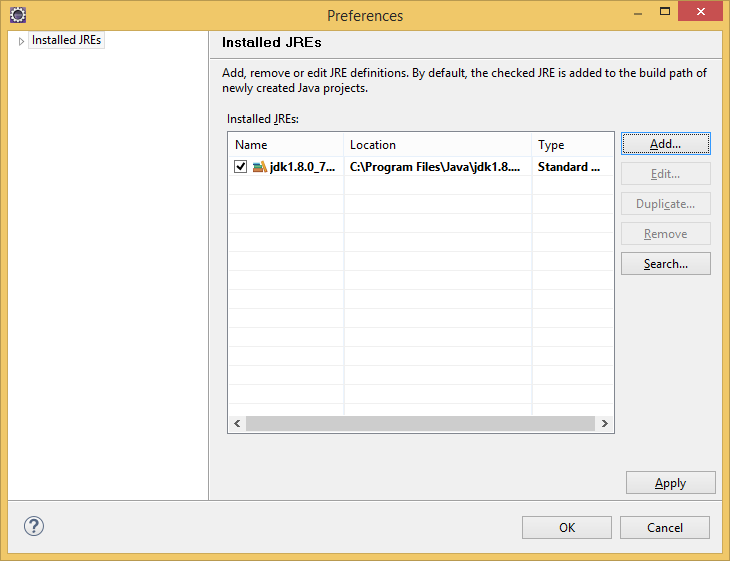
Next and browse for your JDK installation directory (click directory):



Finish ☺  
Remove the current jre and   


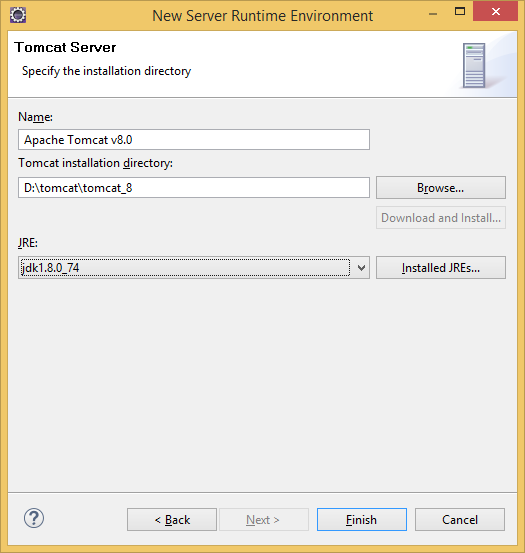
Click remove ☺

You should have something like this until now:

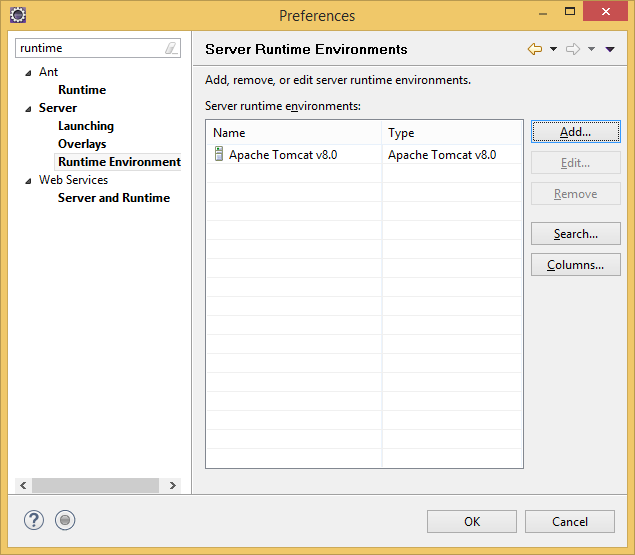


Click OK

Change the JRE from runtime environment definition window:

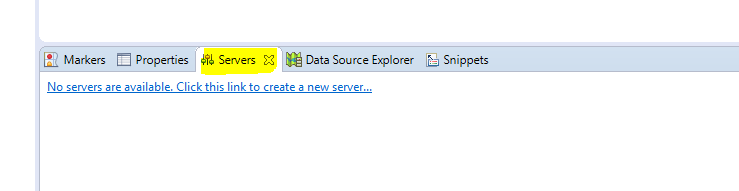


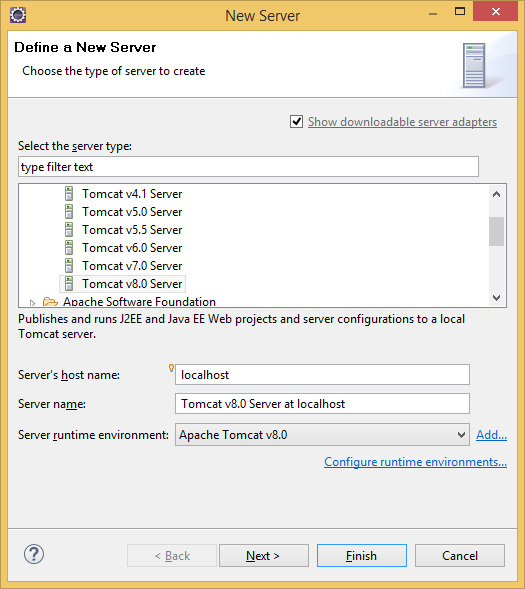
And Finish (now you have a template runtime defined).

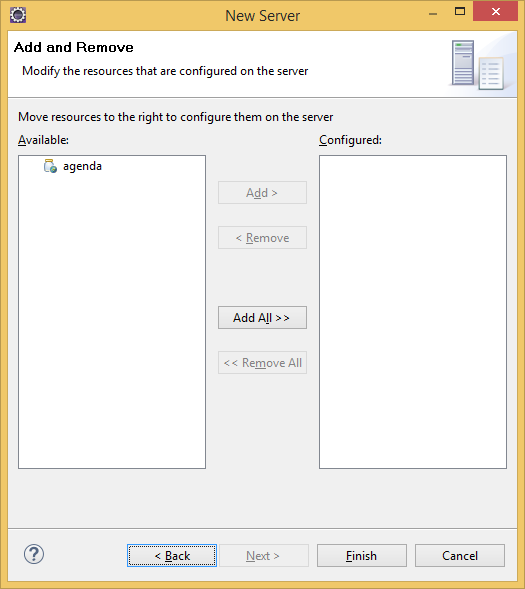


OK and you are done.

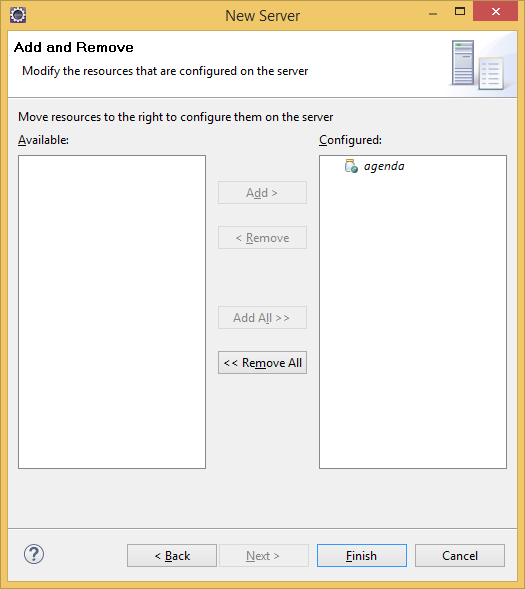
Go to Servers tab:



Click on the blue link to create a new instance of the server:  


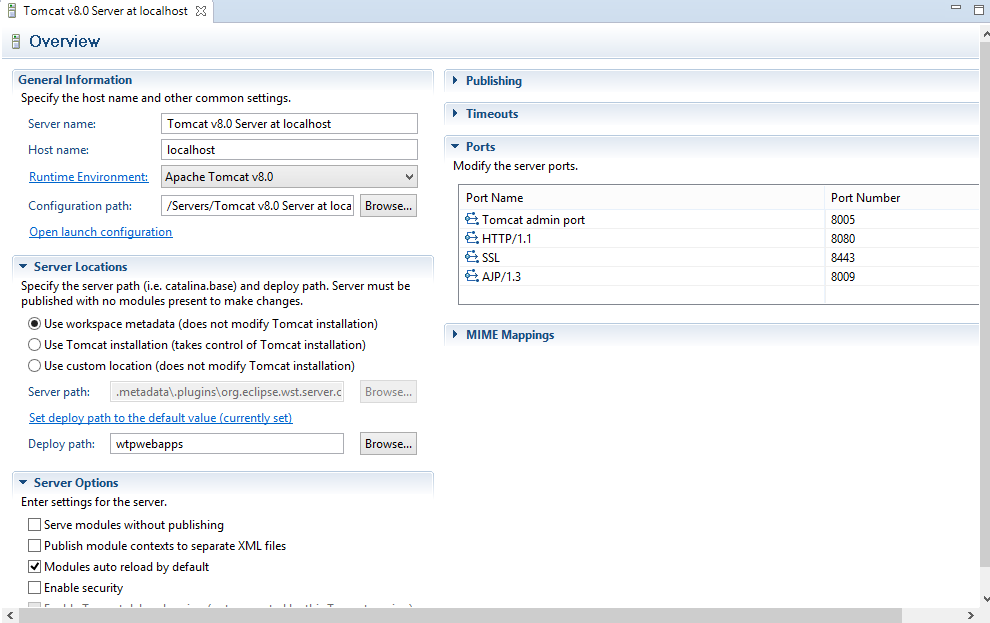
Select Tomcat v8.0 Server, click Next.  


Click on Add All to have your agenda project deployed on the server.

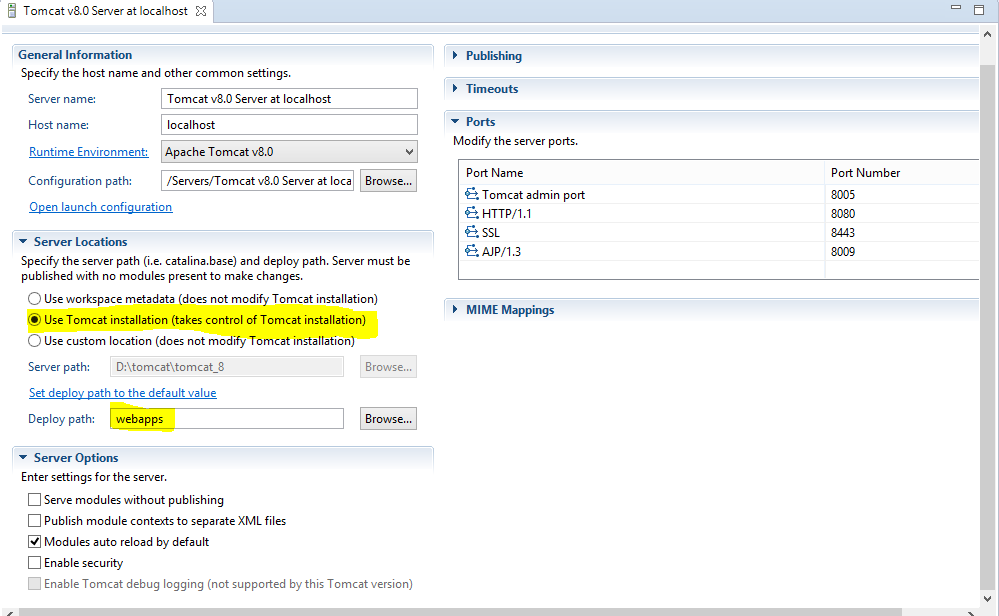


And then click finish.

Now let’s do some basic settings to our server. Double click on your server. You should get to this window:



In Server Locations select de second radio button and change from wtpwebapps to webapps:



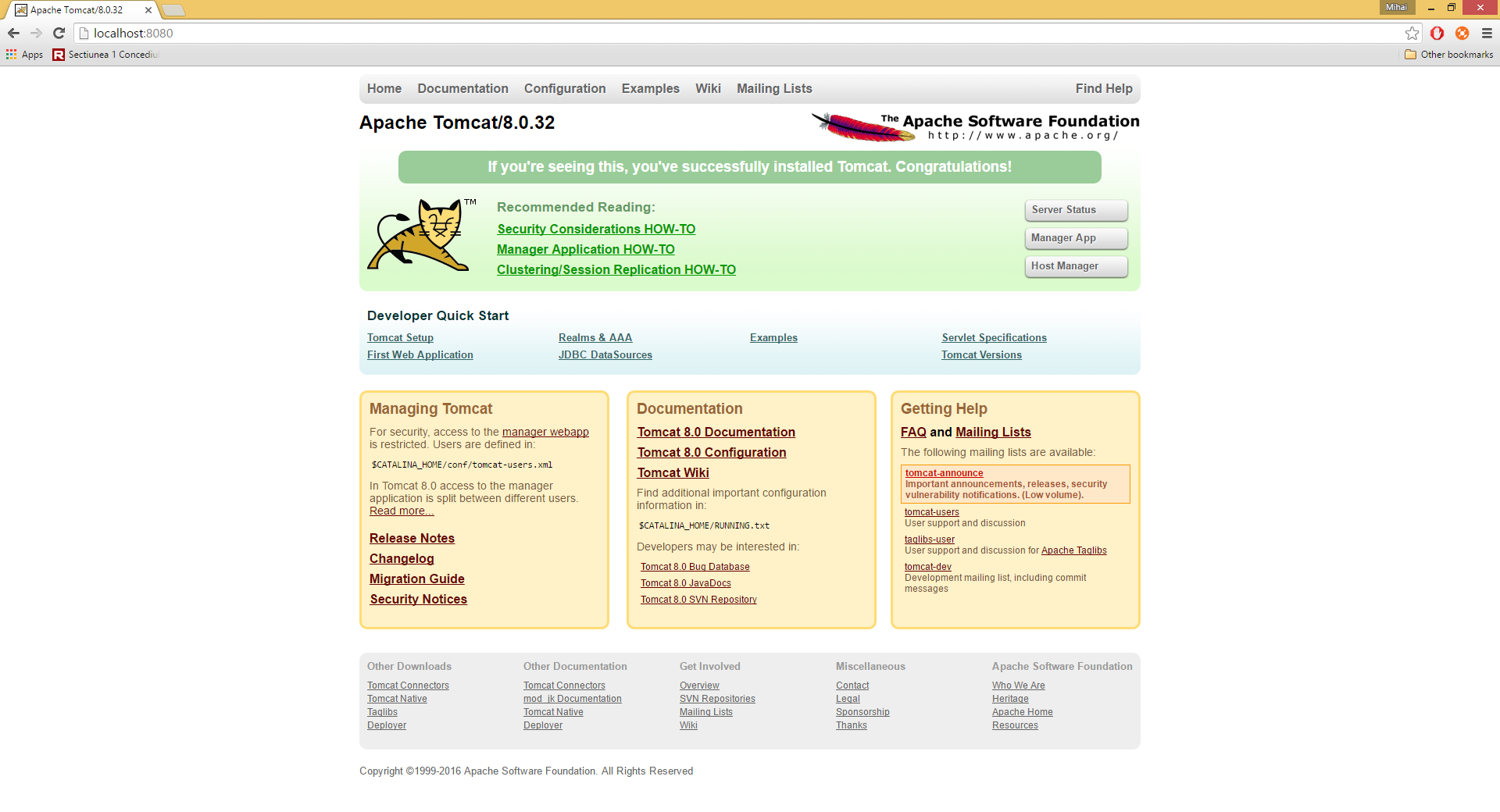
If some of you want to change the port of the application feel free to do it in the right side by changing the port with Port Name HTTP/1.1 and the value of it ☺

Save the modifications (Ctrl+s) you server should now work.

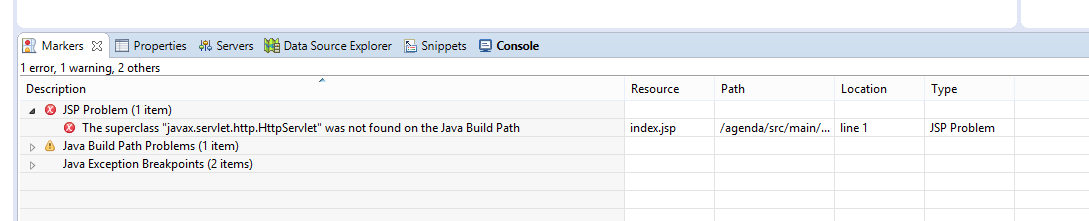
Click on the little green bug in order to start the server in debug mode and your server should display the following info: [Debugging, Synchronized]



And if you go to your browser to <http://localhost:8080/> you should get the following details:

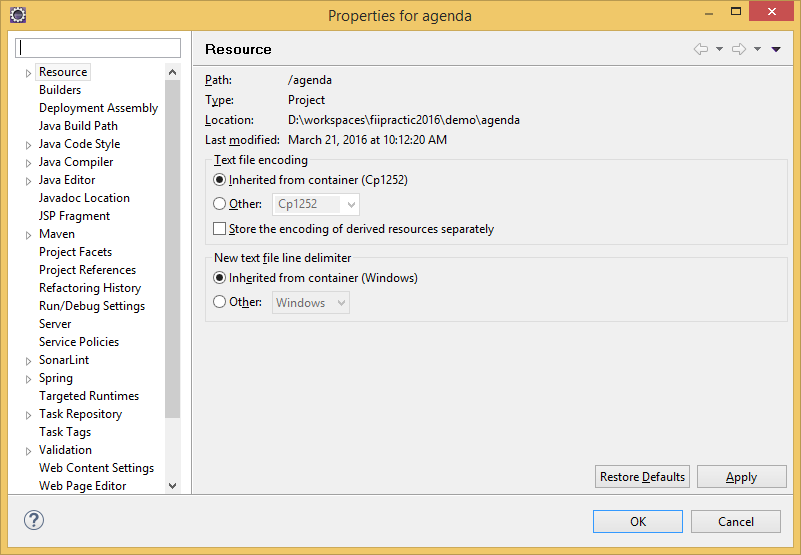


# Getting rid of javax.servlet.http.HttpServlet jsp problem

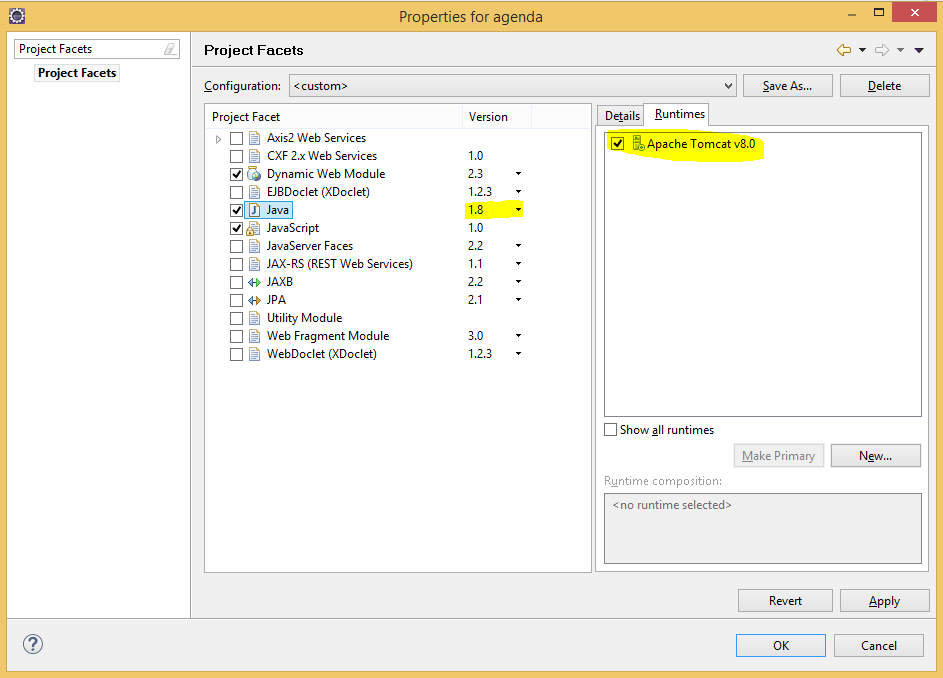


## Method A

In Project explorer right click on the agenda project and select properties:



In the type filter text write Project Facets and then click on it. Go to Runtimes and select Apache Tomcat v8.0, also if you are here you can select the java version to be 1.8.



## Method B

Go to your pom.xml file.

And add the following dependency:

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

The provided scope means that the Application Server (in our case Tomcat) provides the javax.servlet.http.HttpServlet.

I usually go with method B if I know that I’m going to import a lot the project.

# Configuring our maven project

## Configuring pom.xml

Open your pom.xml file and add dependency for spring-webmvc  
Go to <http://mvnrepository.com/artifact/org.springframework/spring-webmvc/4.2.5.RELEASE> and copy the maven dependency group and paste it to your dependencies group in your pom.xml. You should have this info in your pom.xml:

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.2.5.RELEASE</version>

</dependency>

</dependencies>

## Configuring web.xml

Now let’s take care of our web.xml configuration file.

1. Add a new servlet

<servlet>

<servlet-name>rest</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

1. Add a new servlet mapping

<servlet-mapping>

<servlet-name>rest</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

By doing this we are telling our application that any request that are being sent to <http://localhost:8080/agenda> are going to be handled by the servlet defined by the class: org.springframework.web.servlet.DispatcherServlet. This way we are giving the control of requests to the Spring framework.

## Adding our beans configuration file

I would suggest to add to your eclipse the Spring IDE tools (easy to write bean definitions and others). Get it from [here](https://marketplace.eclipse.org/content/spring-ide).

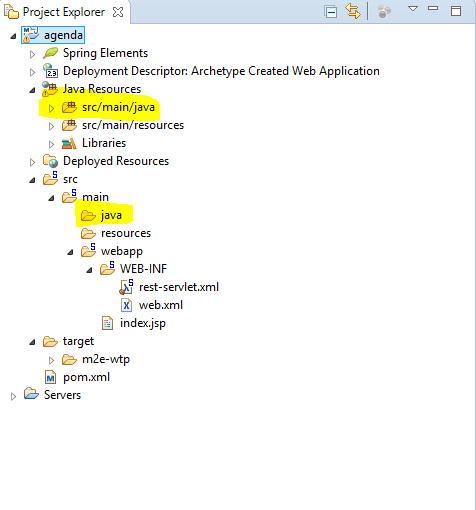
Add rest-servlet.xml configuration



## Adding source (java) folder

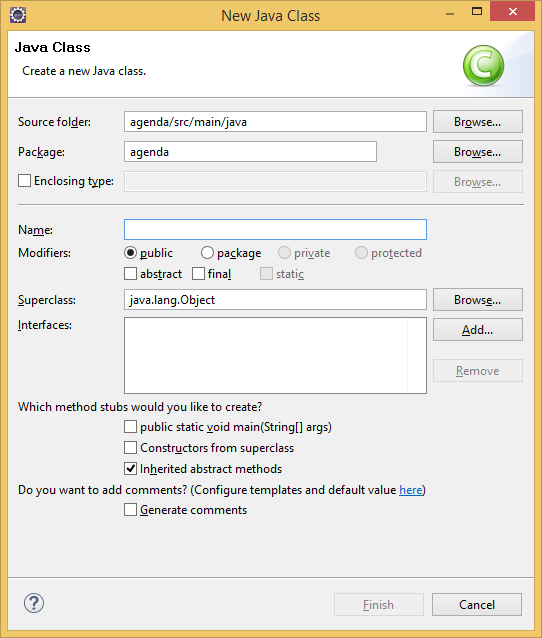
Right click on src/main and add a new folder called java.

You must have the following structure on your project:



## Add your first package and first class

Right click on the src/main/java from Java Resources and select New -> Class (see image below)

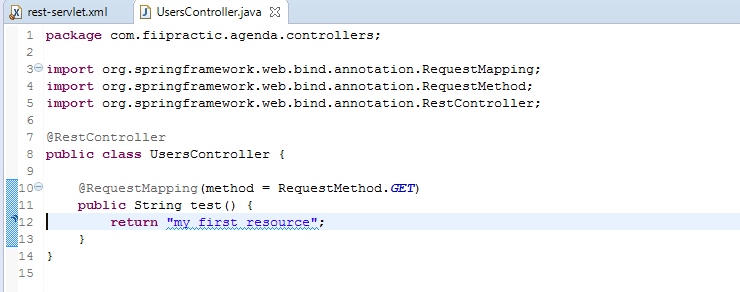


Fill with the following data:

Package: com.fiipractic.agenda.rest.controllers

Name: UsersController

You should get a new java file open in which you should fill like in the image below:



## Publish your application to the server

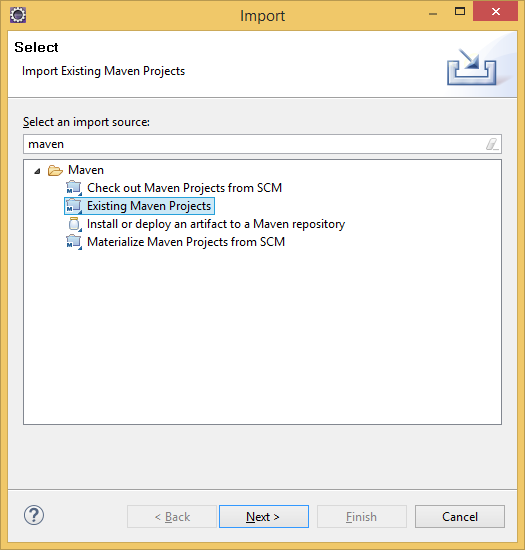
Go to Servers tab and right click on your server and select Publish (Ctrl+Alt+P). Start the server in debug mode (the little green bug button or Ctrl+Alt+D).

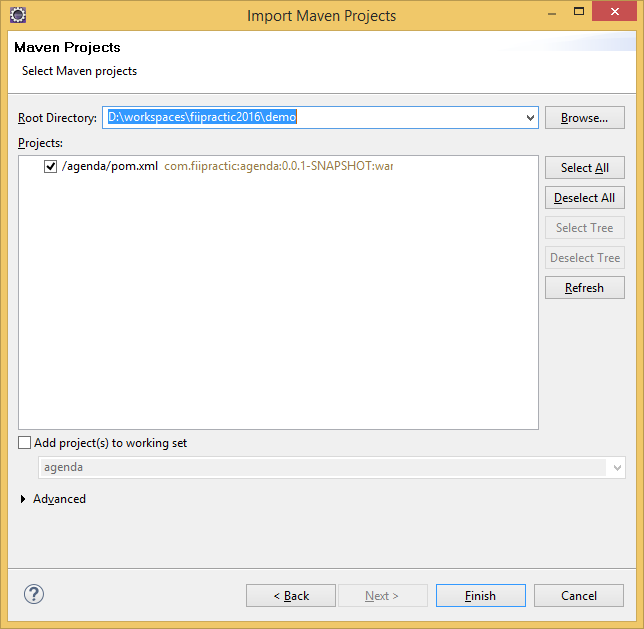
Go to <http://localhost:8080/agenda> to access your first resource.

# Import a maven project

File -> Import.

In the Select an import source write maven and select Existing Maven Projects.



Click next and browse for your pom.xml file. (E.g. mine is located at: D:\workspaces\fiipractic2016\demo)  


Click Finish and your project is imported.

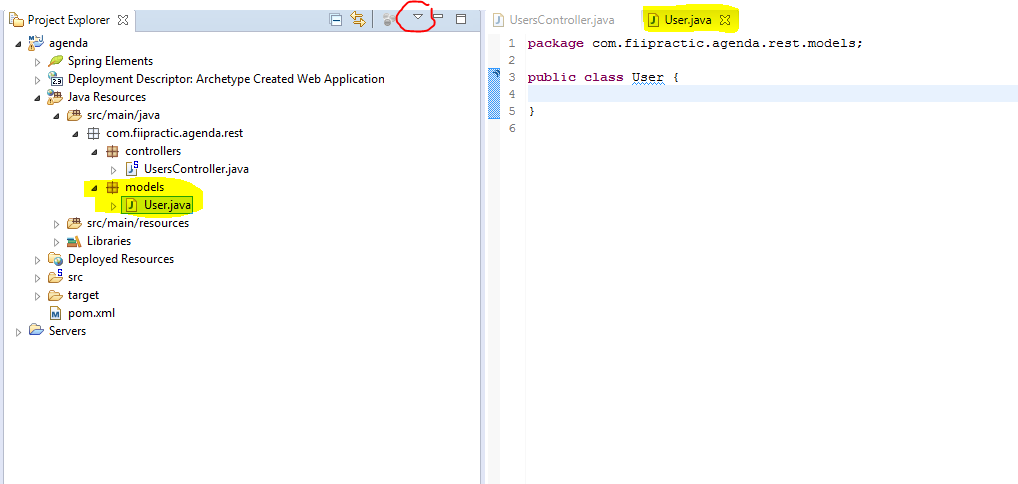
In order to add it to your server you need to go to Server tab and right click on your server and select Add and Remove … and move agenda from available to configured location, the same way you did it when you defined the server.



Start the server and your application should be up and running.

# Add User functionality

Add new class User in package com.fiipractic.agenda.rest.models. The same way you did for UsersController. In order to get the same view as me click on the circled arrow and select Package Presentation Hierarchical.



Let’s think a bit how we will encapsulate a user. What fields do we need?

* id – Long
* username – String
* password – String
* email – String
* and a flag to indicate if the user is enabled in our application or not: enabled – boolean

We are going to generate getters and setters for all our fields.

Now that we encapsulated our info we want to expose it to the real world (via resources) using our UsersController.

We had our first method that was mapped on GET HTTP verb. So we want to add the interaction with our users by mapping CRUD (Create, Read, Update & Delete) to the HTTP verbs.

C (Create) – POST

R (Read) – GET

U (Update) – PUT

D (Delete) – DELETE

Basically we are going to have 5 methods that will interact with our resources getUsers for retrieving all users, getUser for retrieving a particular user, createUser for adding a new user in our application, updateUser for modifying existing users in our application and deleteUser for removing a user from our application.

Let’s remember how did we thought our application URL will look like:

[http://localhost:8080/agenda/users/{username}/contacts/{contactId}](http://localhost:8080/agenda/users/%7busername%7d/contacts/%7bcontactId%7d)

How do you think the above URL should be mapped to the methods?

getUsers – GET /users

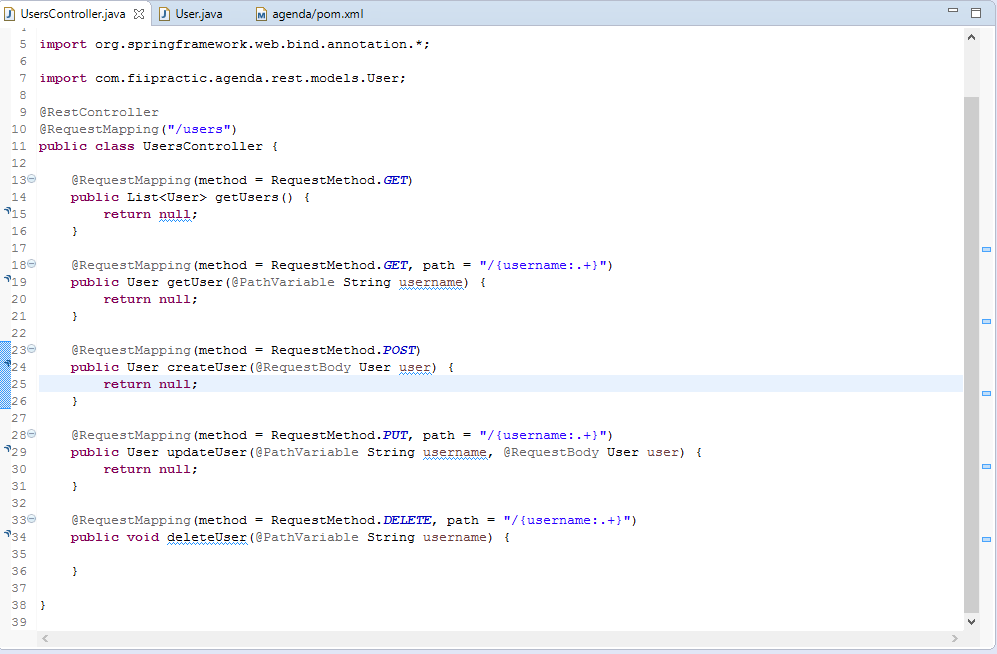
getUser – GET /users/{username}

createUser – POST / users

updateUser – PUT /users/{username}

deleteUser – DELETE /users/{username}

We see that all this methods have /users in common so we are going to annotate UsersController with this path. See below also how our methods are going to look, they are not yet implemented.



Now we need to implement our business logic (the methods that are not yet implemented).

Ok, now you may ask what’s with the annotation @PathVariable and @RequestBody.

Well that’s pretty simple by using @PathVariable we tell spring that the specified variable will be injected with the value from the path in the @RequestMapping annotation.

@RequestBody is used to send data via content, in HTTP you can send your data using two ways:

* via URL – usually used with GET requests (data is seen by anyone that has the URL)
* via request body – sending the data encapsulated in our request body – see below example:

POST /users/m.balaniscu

User-Agent: HTTPTool/1.0

Content-Type: application/json

{

“username”:“m.balaniscu”

“password”:“1234”

“email”:“[m.balaniscu@levi9.com](mailto:m.balaniscu@levi9.com)”

“enabled”:“true”

}

In order to respect basic programming principles we are not going to write business logic in the resource class (UsersController), instead we are going to have a service class that will handle this for us.

We are going to create an interface UserService that will describe how our service interacts with the outside world. In order to use the service in the UserController we need to a private member of type UserService and to annotate it with @AutoWired, this way we are letting Spring know what implementation of service to inject at runtime in our object.

In order for Spring to know what are the classes that needs to be instantiated a new beans configuration file is added called applicationContext.xml. It’s similar to rest-servlet.xml, it only tells Spring where to look for implementation of the interfaces from below.



Because we added this file we need to add a new line in web.xml:

<context-param>

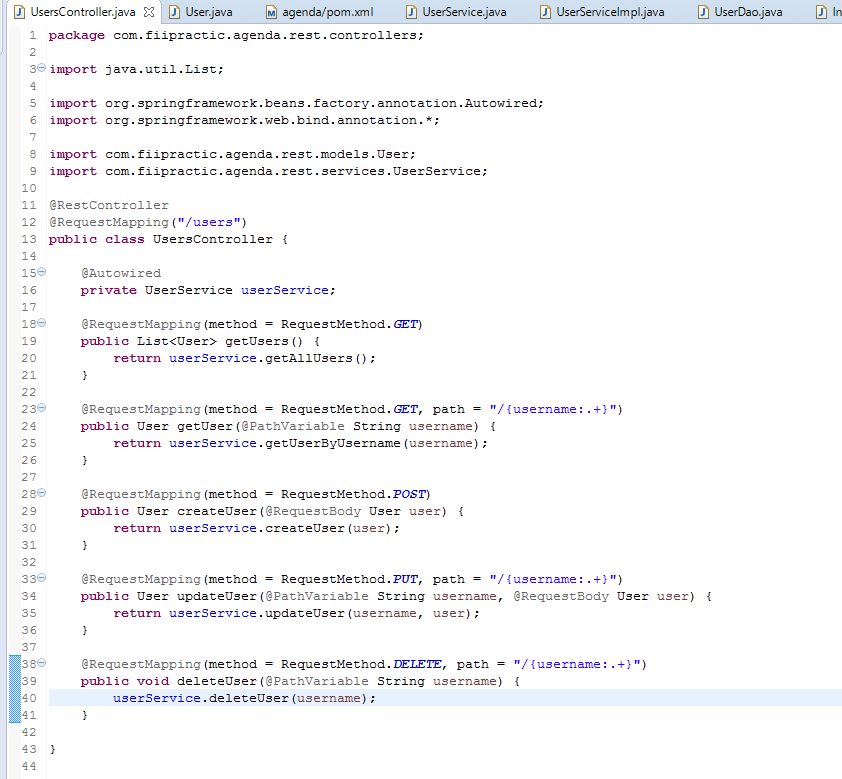
<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/applicationContext.xml</param-value>

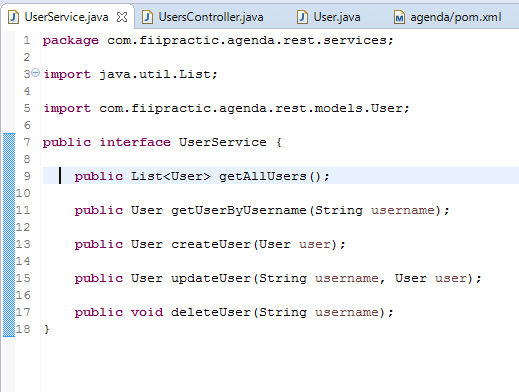
</context-param>

This way we tell Spring that we added a new configuration file.

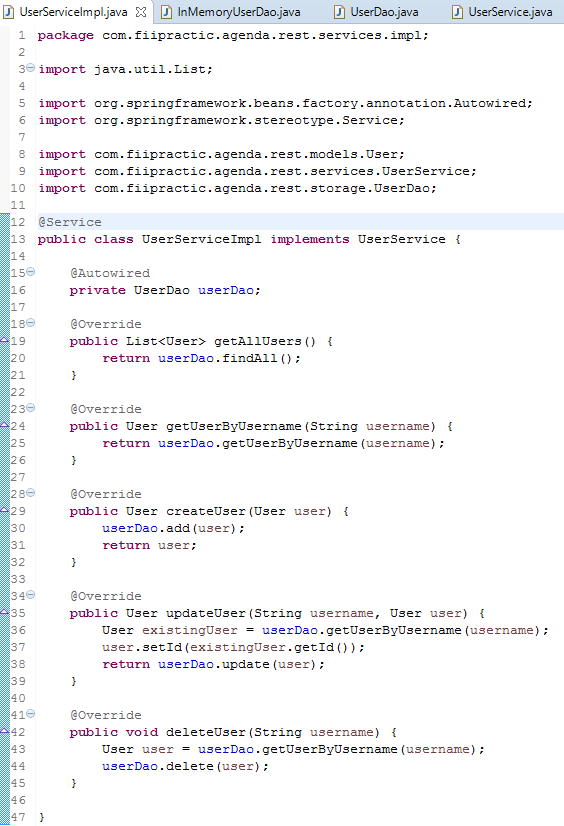
UserController:



UserService:

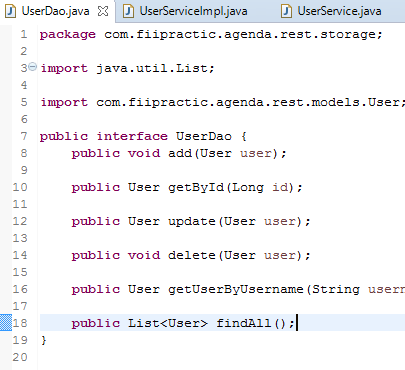


UserServiceImpl – the effective implementation of the service

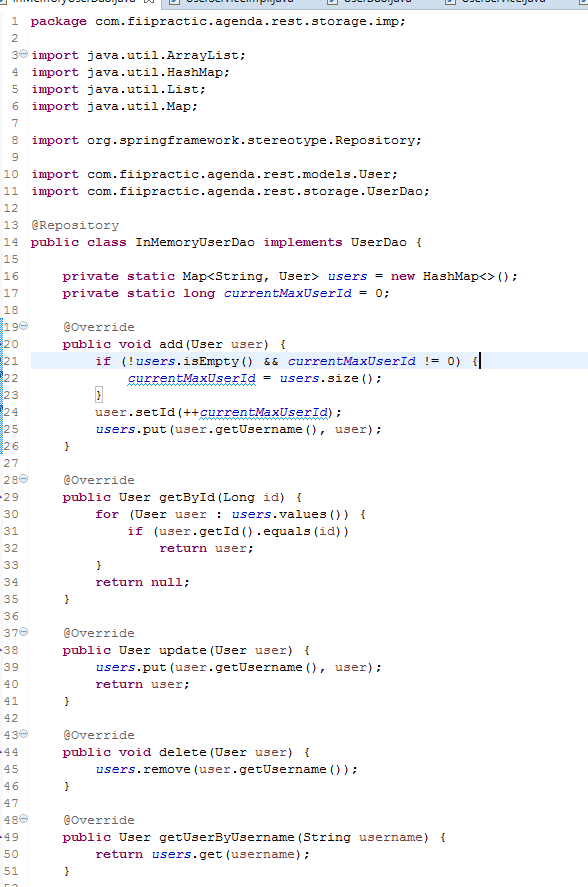


We did the business logic in another class (operating on User object according to our application logic) now we need to store it somewhere, for now we are just going to store it in memory, but we are going to do another layer in our application in order to be easy to maintain in the future, let’s say we are going to change from in memory to a give database then we will need to rewrite the UserService because we modified the database and we do not want to do that.

UserDao – Dao stand for Data Access Objects basically this objects are services for accessing the data layer.



And the current implementation using in memory data storing:



In order to let Spring know what class from the package are services we use:

@Service – for business logic services

@Repository – for DAO services

# Configurate Hibernate & JPA

## Pom.xml – updates

<!-- Hibernate -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>4.3.11.Final</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>4.2.2.RELEASE</version>

</dependency>

<!-- DB driver -->

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>9.4-1206-jdbc4</version>

</dependency>

## Configuring application to use DB (db.properties file)

jdbc.driverClassName=org.postgresql.Driver

jdbc.url=jdbc:postgresql://localhost:5432/agendademo

jdbc.username=agendademo

jdbc.password=agendademo

hibernate.dialect=org.hibernate.dialect.PostgreSQL9Dialect

hibernate.hbm2ddl.auto=update

hibernate.show\_sql=true

hibernate.format\_sql=true

The first part defines JDBC connection settings (in our case we have one database called agendademo) and a DB user identified by username = agendademo and password agendademo (never do this in production☺)

The second part defines the hibernate configuration input, first one says what DB we are using (PostgreSQL), the second one says that we want to create the tables if they are not there, and the two last ones are just for logging purposes (to show the SQL in console and to format it in a nice way – never enable this in production or you will have a bunch of logs with SQL)

## Update applicationContext.xml

### Add application configuration in order to use the external config file

<!-- loading properties file -->

<context:property-placeholder location=*"classpath\*:db.properties"* />

Basically we tell spring that db.properties file is in our classpath and should be loaded in a predefined bean.

### Configuring data source (connection to DB)

Next we configure the bean that will connect to the DB and will have as input the parameters defined in db.properties file ☺.

<!-- DB connection setup -->

<bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"* />

<property name=*"url"* value=*"${jdbc.url}"* />

<property name=*"username"* value=*"${jdbc.username}"* />

<property name=*"password"* value=*"${jdbc.password}"* />

</bean>

### Configuring the Entity Manager

<!-- Entity Manager Bean Setup -->

<bean id=*"entityManagerFactory"*

class=*"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"* />

<property name=*"persistenceUnitName"* value=*"myPersistenceUnit"* />

<property name=*"persistenceProviderClass"*

value=*"org.hibernate.jpa.HibernatePersistenceProvider"* />

<property name=*"packagesToScan"* value=*"com.fiipractic.agenda.rest.models"* />

<property name=*"jpaProperties"*>

<props>

<prop key=*"hibernate.dialect"*>${hibernate.dialect}</prop> <prop key=*"hibernate.hbm2ddl.auto"*>${hibernate.hbm2ddl.auto}</prop>

<prop key=*"hibernate.show\_sql"*>${hibernate.show\_sql}</prop>

<prop key=*"hibernate.format\_sql"*>${hibernate.format\_sql}</prop>

</props>

</property>

</bean>

Entity manager is the one that will ensure that our data is going to be stored in the DB. It uses the bean defined above to handle the connection and it uses the hibernate configuration the db.properties file.

### Configuring the Transaction Manager

<!-- Transaction Manager setup -->

<bean id=*"transactionManager"* class=*"org.springframework.orm.jpa.JpaTransactionManager"*>

<property name=*"entityManagerFactory"* ref=*"entityManagerFactory"* />

</bean>

<tx:annotation-driven transaction-manager=*"transactionManager"* />

Transaction manager will ensure that the data is being stored in a transactional manner – respecting the [ACID](https://en.wikipedia.org/wiki/ACID) properties.