Create a fulltsack application

Context

A web application has several parts, a *frontend* on the client side (ex: Angular) and a *backend* on the server side (ex: Java EE). The objective of this project is to create an application with a backend and a frontend, to have a fullstack application

Goal

The objective of this project is to deepen the knowledge acquired in Java EE and Angular while making the link between Front and Back.
You are going to create a web application allowing to display and add users (users) in a database via a Spring Boot and Angular project.

Prerequisites

- Java 1.8
- Maven
- Spring Boot
- MySQL
- Angular 4

Installation

Here are links to install:

- Java: https://www.java.com/fr/download/help/linux_x64_install.xml

- Maven: https://maven.apache.org/install.html

- Spring Boot: https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-installing-spring-boot.html

- Mysql: https://doc.ubuntu-fr.org/mysql

- Angular: https://angular.io/cli

Tool

The project is developed in Java EE (Java 1.8) with Maven and Spring Boot.
Pages are developed in HTML / CSS / Typescript with Angular.
We will develop a JSON REST API with SPRING to recover data from MYSQL

JAVA

The application will be developed in Java EE 1.8, the java program will allow you to manipulate your database and control the data.

Maven

Maven uses an XML file called Project Object Model (POM) to describe a software project, its dependencies on external modules and the order to follow for its production.
The dependencies are added to the pom.xml file.

With Maven many things will be done automatically, there is no need to know how it works but only how to use it (example: compilation).
Maven allows:
automate certain tasks: compilations, unit tests and deployment of the applications that make up the project
manage library dependencies required for the project
generate documentation concerning the project

Example of dependency for the Joda Time library in the pom file:

```
<dependencies>
            <dependency>
                  <groupId>joda-time</groupId>
                  <artifactId>joda-time</artifactId>
                  <version>2.9.2</version>
            </dependency>
</dependencies>
```

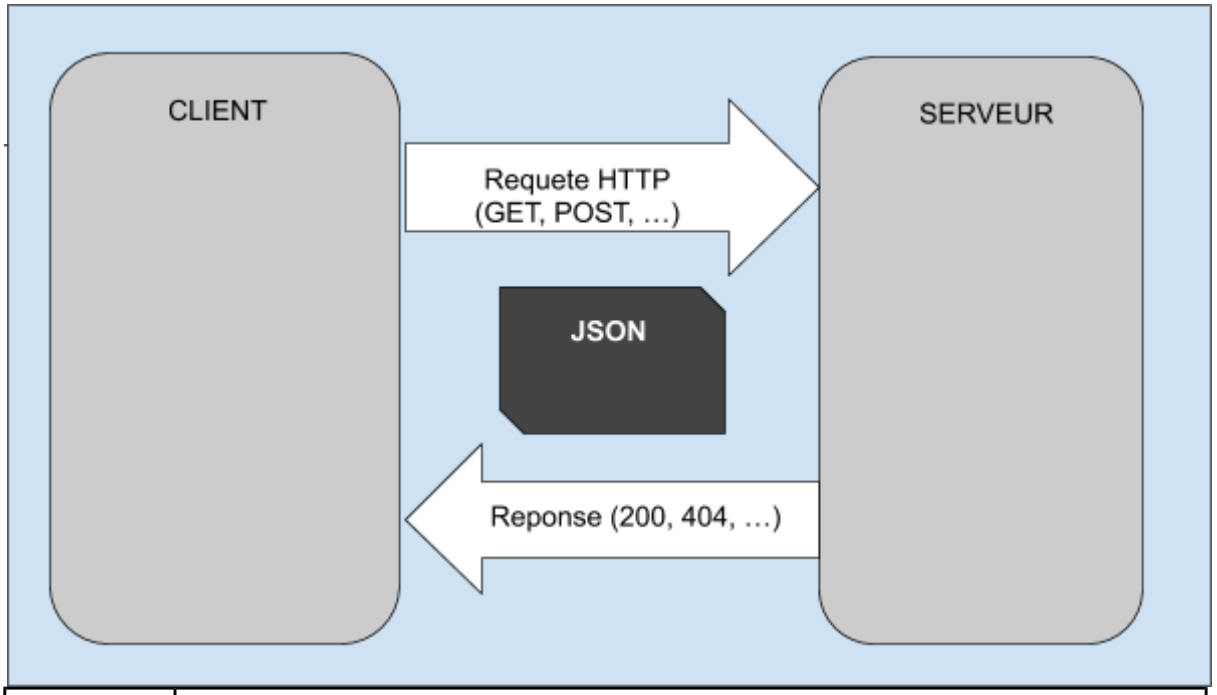
Among other things, this pom.xml file provides information on the name of the project, its version and the dependencies on external libraries.

Spring Boot

Spring is a framework to facilitate the development of a Java EE application. It allows you to quickly start developing applications.

API REST

Data transfers will be done using a REST API.
API means for Application Programming Interface, REST means for Representational State Transfer. The REST APIs are used for exchanging information and imitate exchanges between client-servers.
The data are called resources and are identified by URLs respecting a particular format.



GET:	GET http://site.com/users?parametre=exemple&autre_parametre=exemple2 to retrieve data from a resource.
POST:	If you are using an API to leave a comment on an article on another site the POST request may look like the following: POST /page.php HTTP/1.1 Host: site.com Content-Type: application/x-www-form-urlencoded Content-Length: 36
	titre=titreici&corps=icimoncommentaire Allows you to send data in a request and often to add it to the resource specified in the URI part of the first line of the request.

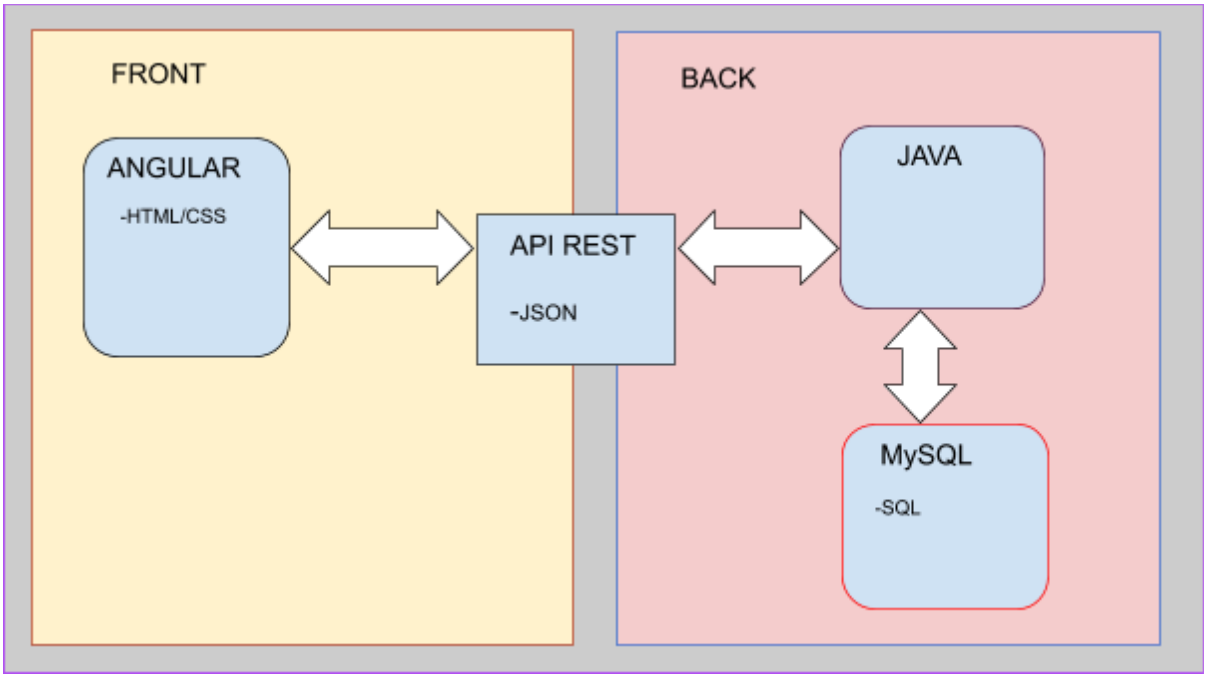
PUT:	PUT /users/10 HTTP.1.1 name=jessica This request does two things: it asks to create a new user with ID 10 if there is none or to replace the user with this ID! It's not a good idea. It is generally better to let the server decide which ID to give to a new object and what to do with it. As you are the client in this case, it is not really you who knows better, especially if you work with an external REST API.
DELETE	By using DELETE you are saying that you want to delete the resource given in the URI.

For this project, data is sent using this API as a JSON file. So the Java program sends a JSON file that Angular retrieves, reads and then displays the data. (See section Diagram).
In a simplified way, this creates a service that accepts HTTP requests and responds with a JSON file.
Here is how to identify resources (fictitious links):
To recover all the players:
https://api.playerplanet.com/greeting
Corresponding JSON file:
`{“id”:1, “content”: “Hello, World!”}`
We can customize with for example a **name** parameter
https://api.playerplanet.com/greeting?name=User
Corresponding JSON file:
`{“id”:1, “content”: “Hello, User!”}`

The server response is made up of three digits indicating success or not as well as the type of problem encountered (200 OK, 404 page not found, ...).

Project creation

Here is a tutorial to explain how to create a fullstack application using the REST API. Below a diagram to illustrate the different parts that we will implement:



Before starting, put your MySQL server online with this command:

```
sudo systemctl start mysql
```

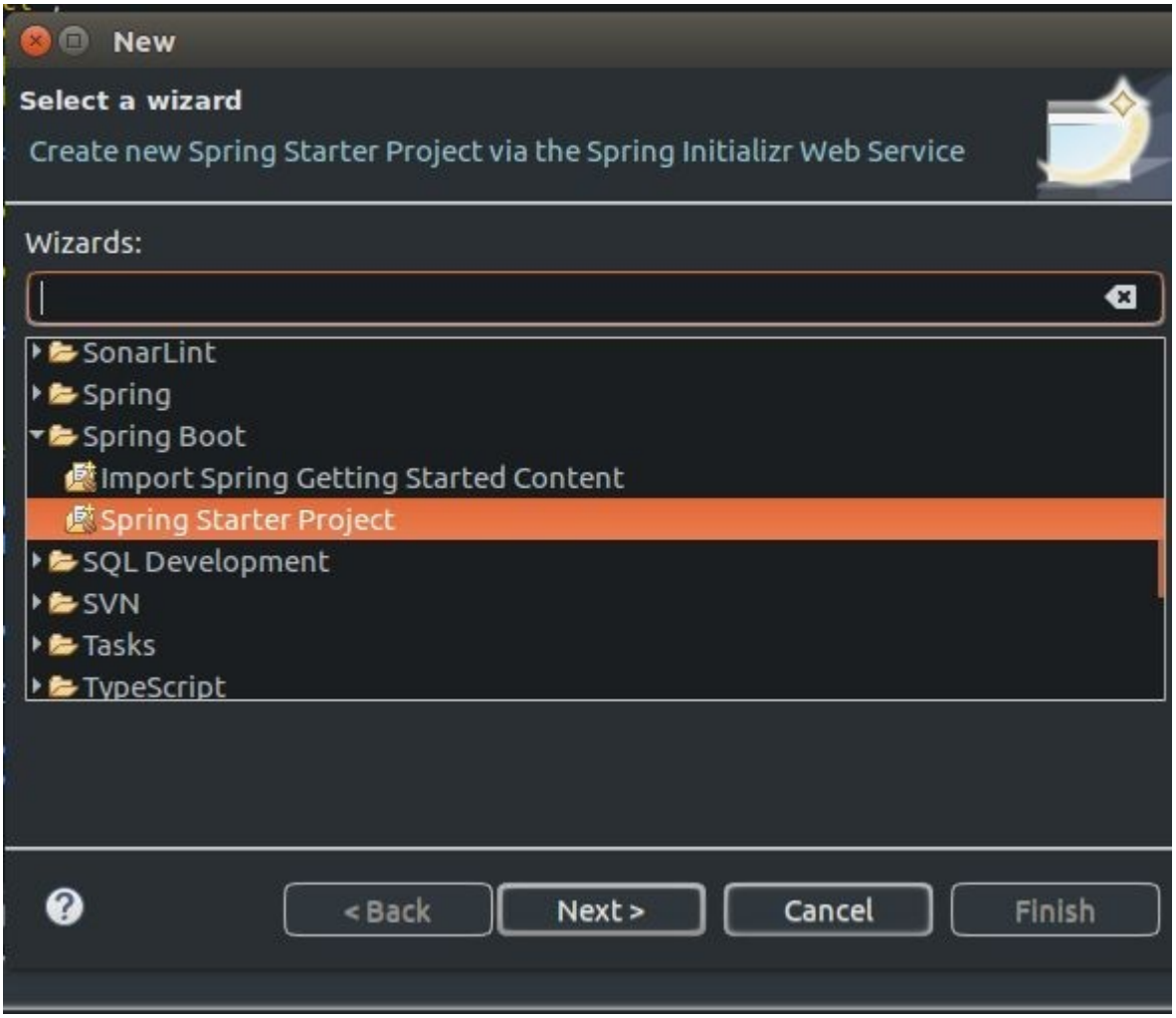
Then create a database:

```
CREATE DATABASE IF NOT EXIST name;
```

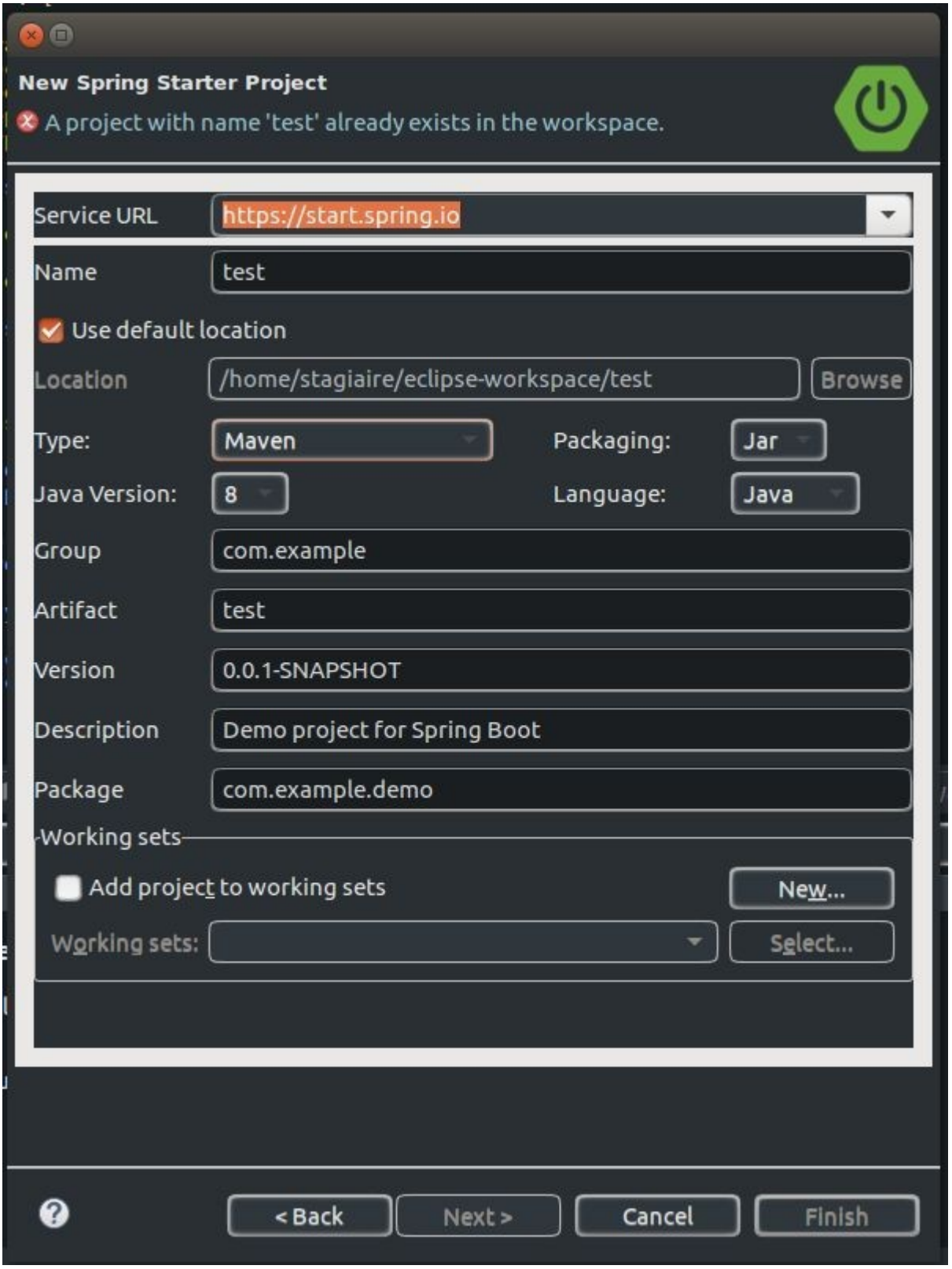
Replace “name” with the name of your choice and keep this name in memory.

Now let's tackle this project, this time we are not going to create a classic Java EE web project but a **Spring Boot** project.

To do this, start Eclipse, Click on **New> Other**
open **Spring Boot** then **Spring Starter Project**

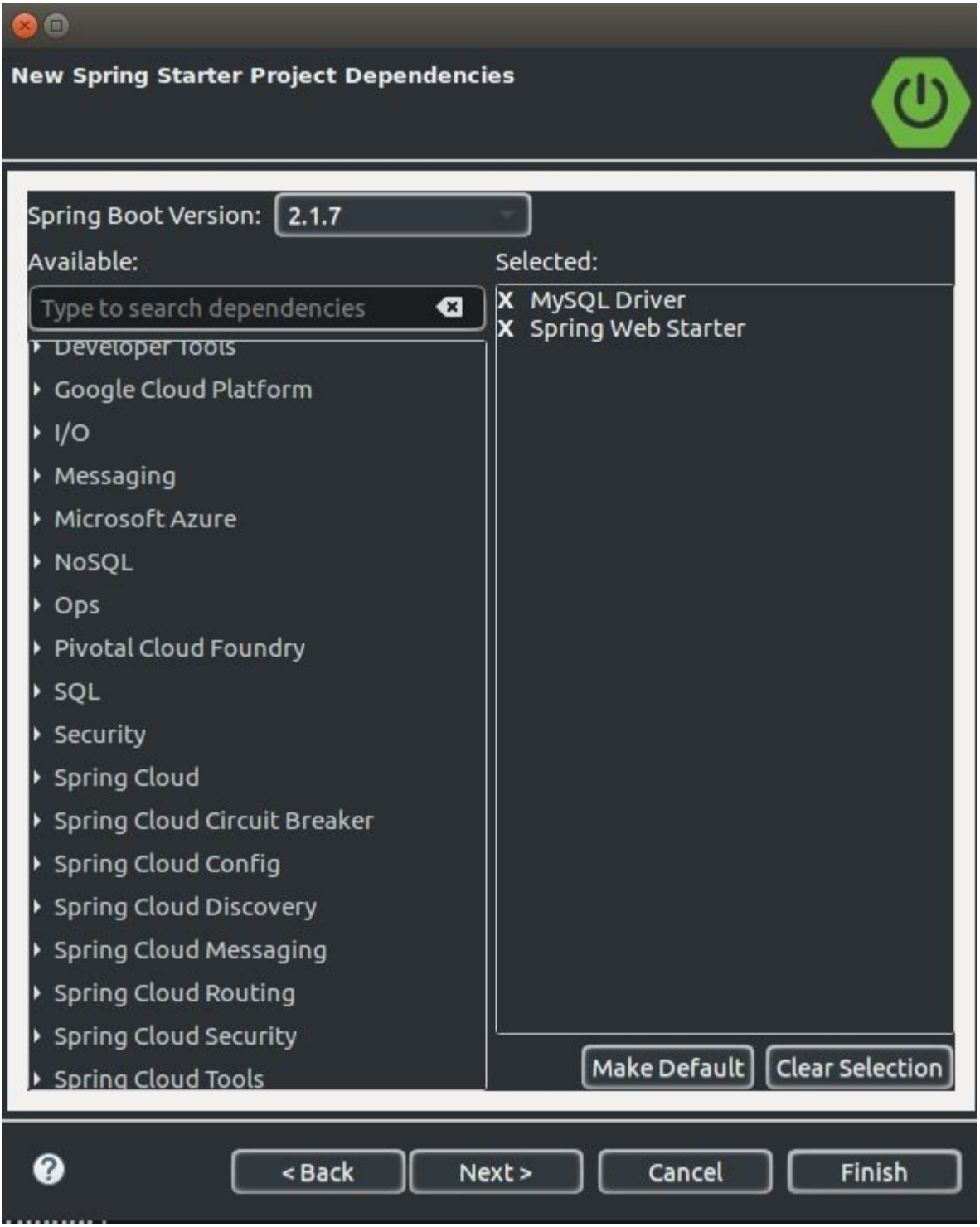


Click **Next**, give a name to your project, make sure to set the **Type** on **Maven** and **Java Version** on **1.8**

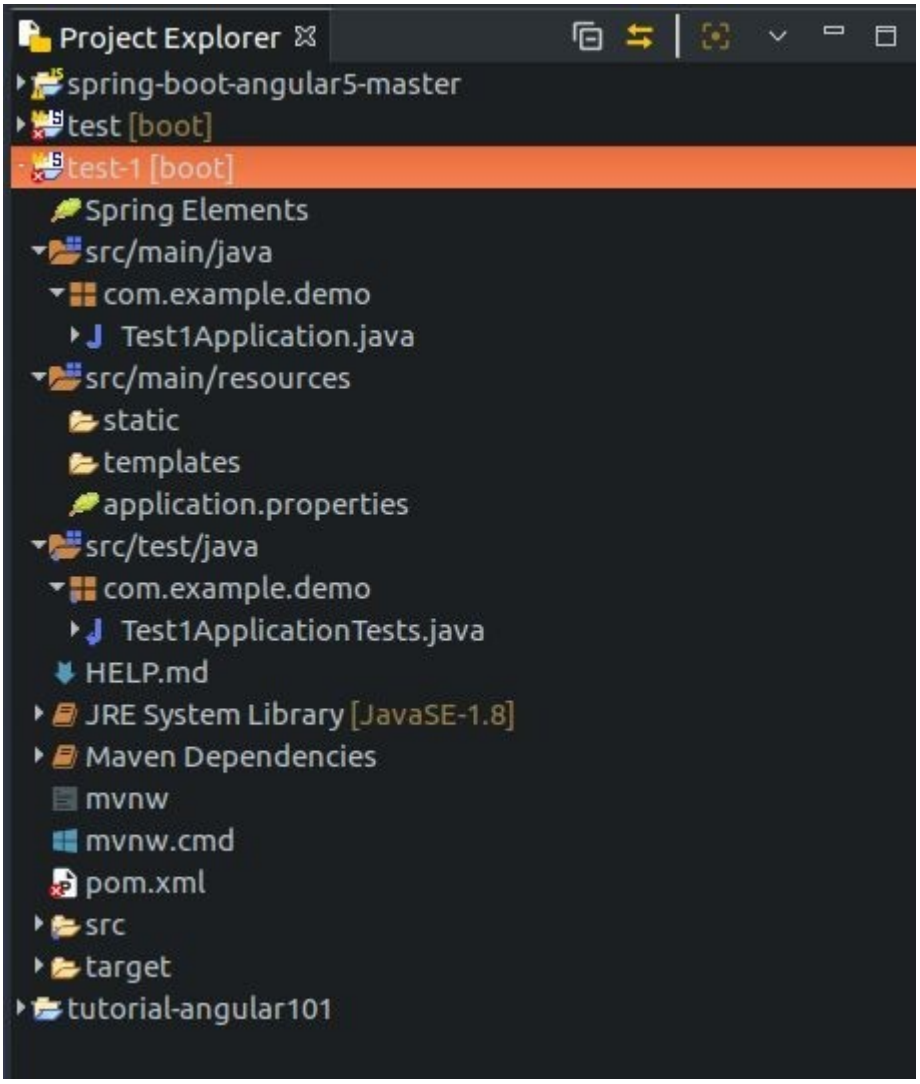


Click on **Next**, you will arrive on the dependencies page, this step is not currently required since you can modify the file whenever you want later by hand. However, this can come in handy if there are dependencies that you know will be necessary for your project.

Our goal is to create a web application using a REST API and an SQL database, you can already add the corresponding dependencies here, for that click on the **MySQL** and **Spring Boot** tabs.



Click on **Finish**



Your project is created, on **Project Explorer** you should have this tree structure (here test-1).
Your Java files will be in **src/main/java** and the resources/properties files in **src/main/resources**.

The dependencies are located in the **pom.xml** file, which can also be modified by hand. It may be useful to browse the Maven Repository site to find new dependencies to add to the project.

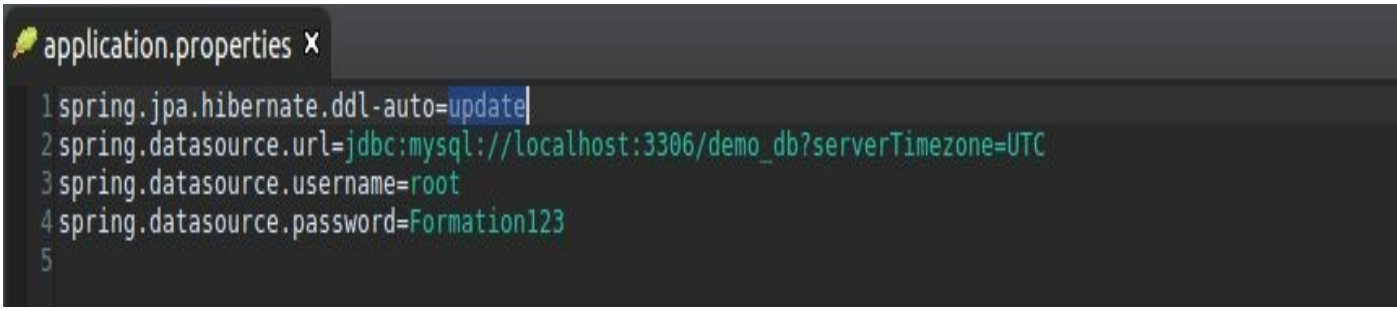
The **pom.xml** file should look like this:

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.springframework</groupId>
    <artifactId>gs-mysql-data</artifactId>
    <version>0.1.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.6.RELEASE</version>
    </parent>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- JPA Data (We are going to use Repositories, Entities, Hibernate, etc...) -->

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <!-- Use MySQL Connector-J -->

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Looking closer, we can notice the dependencies **spring-boot-starter-web** and **mysql-connector-java**, corresponding to **Spring Web Starter** and **MySql Driver** when choosing dependencies. The **spring-boot-starter-data-jpa** dependency was added later.

We will now configure the access to the database, for that open the file application.properties Put the attribute **spring.jpa.hibernate.ddl-auto** on the **update** value, as implied on this value, the database will be updated (example: addition of column, constraints, etc ...) and will not delete the existing tables during the application shutdown.
This also allows, when the application starts, to automatically create a table in the database based on the annotated class *Entity*, if it has not already been created before.
For the attributes *url*, *username*, *password*, they correspond to the name of the database you are using (here *demo_db*), to the user name (here *root*) and then to its password.
Be careful to add the *serverTimezone* parameter (here set to UTC) to the URL of your database, otherwise an error will occur when connecting to the database.



Now let's go to Java code, to start, we need to create a JPA Entity type class which in our case will be a user with an id (primary key auto-incremented in the database), a name and an email address. First, let's create a **User** class:

User class

This class will contains 3 fileds:

- an **integer** *id*
- a **String** *name*
- a **String** *email*

Add *setters* et *getters*

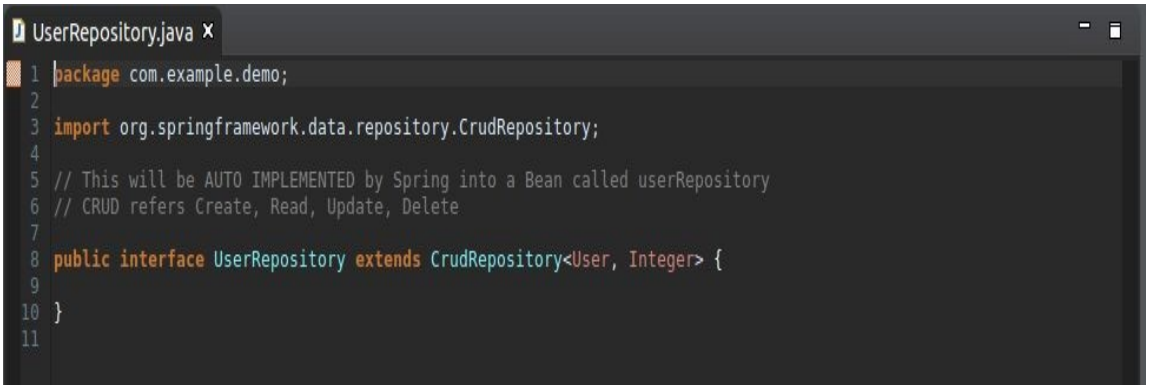
Just above the line “public class User”, type the annotations @**Entity** then at the line @**Table** (name = ”User”)
The @**Entity** annotation therefore makes it possible to define the **User** class as a JPA entity so that it is stored in a table of the database, which will be created automatically as we have just seen. The name of this table can be modified via the @**Table** annotation (optional), here for the example the name remains set to user.

Now above the id attribute, add the annotations @**Id** then @**GeneratedValue** (strategy = GenerationType.AUTO)
The @**Id** and @**GeneratedValue** annotations make it possible to define the id field as a primary key whose value is auto-incrementing.

Interface UserRepository

We must then create an interface for the **User** class, this will simply allow us to use the **CRUD** (Create, read, update, delete) functions already written in the **CrudRepository** interface on our user table.

Here is a link to the CrudRepository documentation:
https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html



MainController class

Now we need a controller class for the **REST Controller**, for that create a **MainController** class:

MainController.java

```java
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;

@RestController
@CrossOrigin(origins = "http://localhost:4200")
public class MainController {
    @Autowired // This means to get the bean called userRepository
               // Which is auto-generated by Spring, we will use it to handle the data
    private UserRepository userRepository;

    @PostMapping("/users")
    void addUser(@RequestBody User user) {
        userRepository.save(user);
    }

    @GetMapping("/users")
    public @ResponseBody Iterable<User> getAllUsers() {
        // This returns a JSON or XML with the users
        return userRepository.findAll();
    }
}
```

We can note several things in this code, first of all the annotations @**RestController** makes it possible to indicate that the class is a “controller” and @**CrossOrigin** makes it possible to indicate that only “http://localhost:4200” can carry out cross-origin requests. We also note that the origins parameter refers to the default URL of an Angular project, which will also be used in our project.

CORS

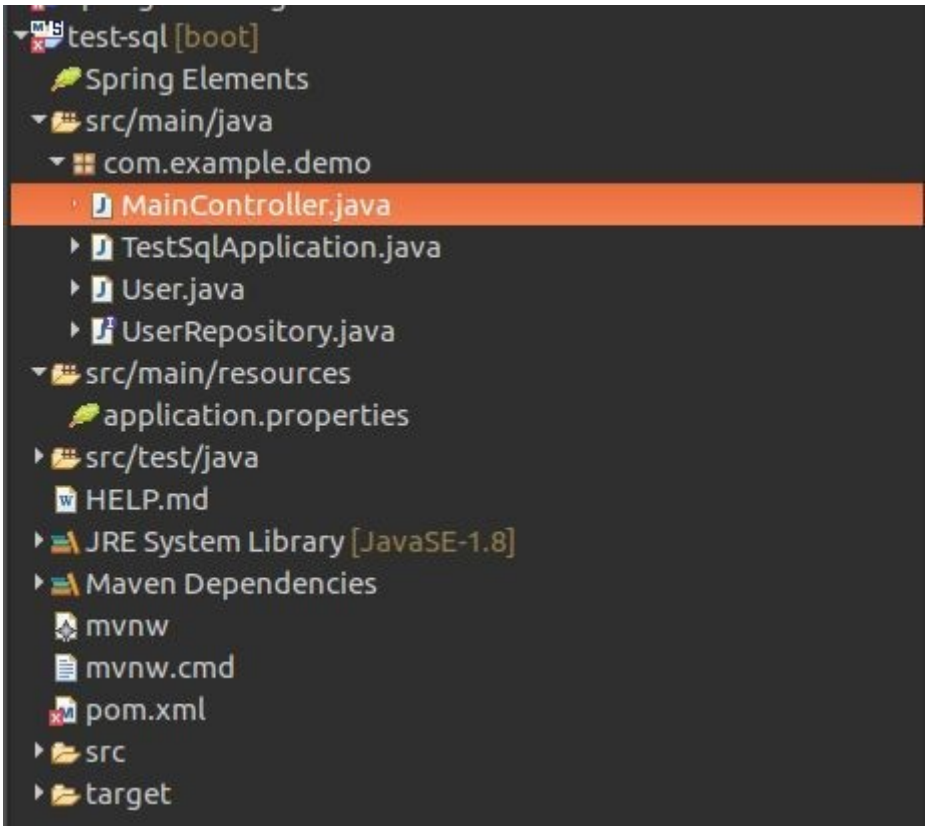
CORS (Cross-Origin Resource Sharing) is a security mechanism that allows a web page coming from a domain or other origin to access a resource with a different domain (cross-domain request).
Example: http://localhost:9000 and http://localhost:8080 are two different origins, let's imagine that a small app on localhost: 9000 needs to make an AJAX request on localhost: 8080, CORS allows to perform cross-domain requests.

Then, the getAllUsers() and addUser() methods will be called during HTTP requests on the URL “/users”. The annotation used for getAllUsers() is @GetMapping (“/users”) which at the time of a Get HTTP request makes it possible to list all the Users via the findAll() method of the UserRepository directory previously set up. In the same way the addUser method makes us a Post request thanks to the save() method.

Finally, the Application class is created automatically by Spring Boot and simply implements the main() method. That’s it for the *backend* part.

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class TestSqlApplication {

    public static void main(String[] args) {
        SpringApplication.run(TestSqlApplication.class, args);
    }

}
```

At the end of all this, you should have the following tree structure:



Angular

JSP files will not be used for client-side display, instead we will use the Angular framework. Data transfer between Java and Angular will be done using the REST API.

Open the terminal, then in the folder of your choice type this command to create an Angular project (here called angularclient):

```
ng new angularclient
```

The application entry point is the **index.html** file:

App Component

Let's go to the file ***app.component.ts***

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string;

  constructor() {
    this.title = 'Spring Boot - Angular Application';
  }
}
```

You can edit ***title.***

Let's go to the file ***app.component.html***:

```html
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <div class="card bg-dark my-5">
        <div class="card-body">
          <h2 class="card-title text-center text-white py-3">{{ title }}</h2>
          <ul class="text-center list-inline py-3">
            <li class="list-inline-item"><a routerLink="/users" class="btn btn-info">List Users</a></li>
            <li class="list-inline-item"><a routerLink="/adduser" class="btn btn-info">Add User</a></li>
          </ul>
        </div>
      </div>
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

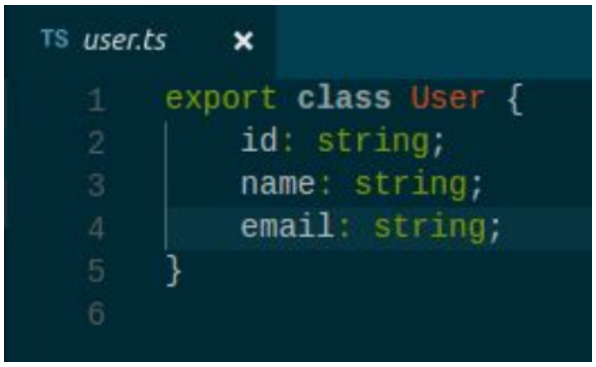
Add two links, one to list the Users and another to add a new User, the **routerLink** attribute for each will be set to "/user" for the list and "/adduser" to add.

The **routerLink** attribute of our buttons allows you to query either / users or / adduser. The corresponding component will be called and displayed via the *<router-outlet></router-outlet>* tag.

Now type this command on the console:

```
ng generate class user
```

This command will create an empty User class, edit it so that this class has the same attributes of the User class in Java (an id, a name and an email):



We must now create a service that performs GET and POST requests on: http://localhost:8080/users.

Now type this command on the console:

```
ng generate service user-service
```

Open **user-service.service.ts**:

This is where the functionality for using the REST API you implemented in Spring Boot is.

The **findAll()** method performs an HTTP GET request on http://localhost:8080/users (remember that we defined this URL to make our SQL requests in the Java **MainController** class) via Angular and returns an observable type User[].

The **save()** method will take a User as a parameter and will save it in the database via a POST request. Be careful to change the type of response to ‘text’ to avoid an error.

We are now going to implement a component which will allow us to display the list of users present in the database and which will therefore use the findAll() method of our *user-service*.

User-list

Now type this command on the terminal:

```
ng generate component user-list
```

TS user-list.component.ts

```ts
import { UserServiceService } from './../../services/user-service.service';
import { User } from './../user';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.css']
})
export class UserListComponent implements OnInit {

  users: User[];

  constructor(private userService: UserServiceService) { }

  ngOnInit() {
    this.userService.findAll().subscribe(data => {
      this.users = data;
    });
  }

}

```

So we get an instance of our service, then in the NgOnInit() method, we will subscribe to the observable returned by the findAll() method and simply get a list of User from it.

Open the file user-list.component.html and modify it so that the page displays the list of Users. Remember to use *ngFor to iterate over the list .

<> user-list.component.html

```html
<div class="card my-5">
  <div class="card-body">
    <table class="table table-bordered table-striped">
      <thead class="thead-dark">
        <tr>
          <th scope="col">#</th>
          <th scope="col">Name</th>
          <th scope="col">Email</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let user of users">
          <td>{{ user.id }}</td>
          <td>{{ user.name }}</td>
          <td><a href="mailto:{{ user.email }}">{{ user.email }}</a></td>
        </tr>
      </tbody>
    </table>
  </div>
</div>

```

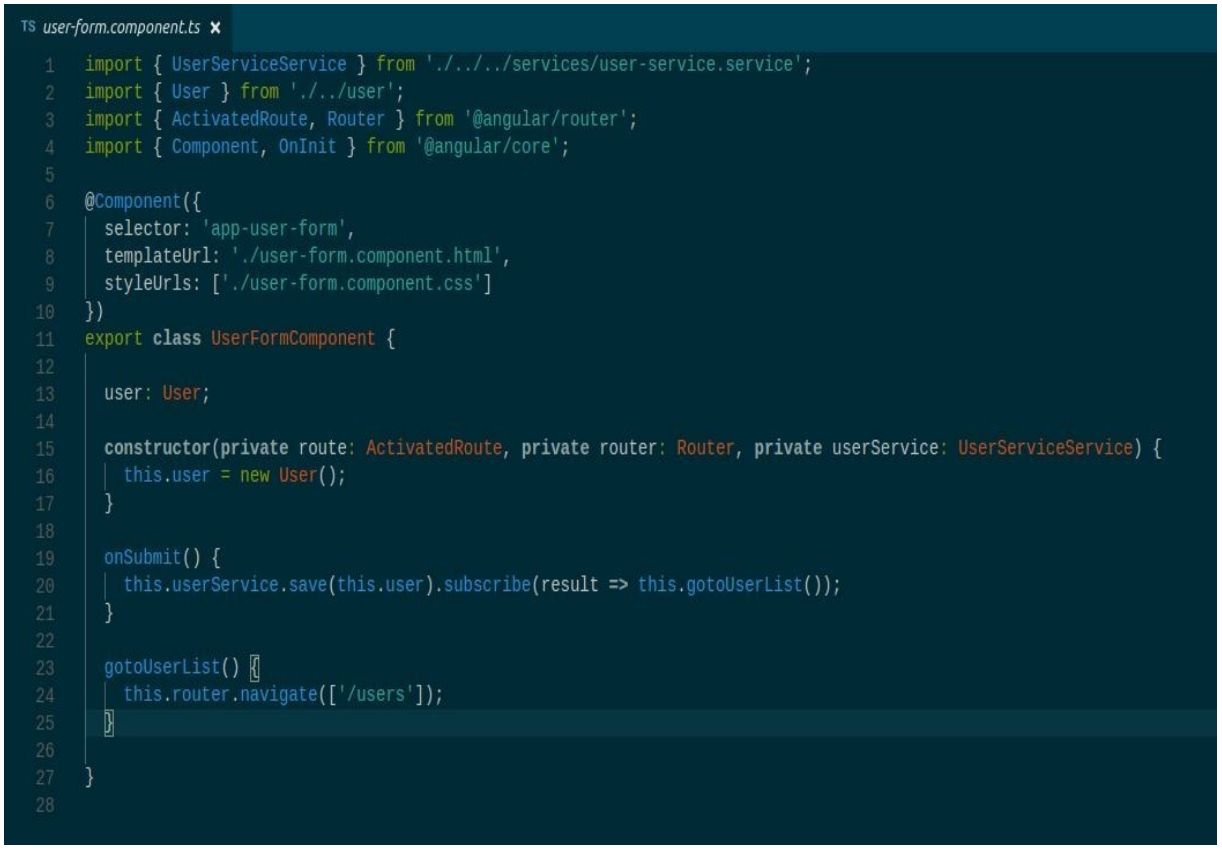
So much for the display, we now need to take care of the form to add a user to our database.

User-form

Type this command on the terminal:

```
ng generate component user-form
```

Edit the ***user-form.component.ts*** file.



```ts
import { UserServiceService } from './../../services/user-service.service';
import { User } from './../user';
import { ActivatedRoute, Router } from '@angular/router';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-user-form',
  templateUrl: './user-form.component.html',
  styleUrls: ['./user-form.component.css']
})
export class UserFormComponent {

  user: User;

  constructor(private route: ActivatedRoute, private router: Router, private userService: UserServiceService) {
    this.user = new User();
  }

  onSubmit() {
    this.userService.save(this.user).subscribe(result => this.gotoUserList());
  }

  gotoUserList() {
    this.router.navigate(['/users']);
  }

}
```

This component is used to implement the onSubmit() function, which calls the save() function of the *user-service* when the form is validated. As soon as this is done, you can call the gotoUserList() function to simply navigate to the list of users.

You must set up the form allowing the addition of a User with the file **user-form.component.html**.

user-form.component.html

```html
<div class="card my-5">
  <div class="card-body">
    <form (ngSubmit)="onSubmit()" #userForm="ngForm">
      <div class="form-group">
        <label for="name">Name</label>
        <input type="text" [(ngModel)]="user.name"
          class="form-control" id="name" name="name" placeholder="Enter your name"
          required #name="ngModel">
      </div>
      <div [hidden]="!name.pristine" class="alert alert-danger">Name is required</div>
      <div class="form-group">
        <label for="email">Email</label>
        <input type="text" [(ngModel)]="user.email"
          class="form-control" id="email" name="email" placeholder="Enter your email address"
          required #email="ngModel">
        <div [hidden]="!email.pristine" class="alert alert-danger">Email is required</div>
      </div>
      <button type="submit" [disabled]="!userForm.form.valid" class="btn btn-info">Submit</button>
    </form>
  </div>
</div>

```

Nothing in particular to report for this form, which calls the component's onSubmit () function at the time of submission.

App-routing

Now you need to set up the navigation, create the **app-routing.module.ts** module with the following command:

```
ng generate module app-routing
```

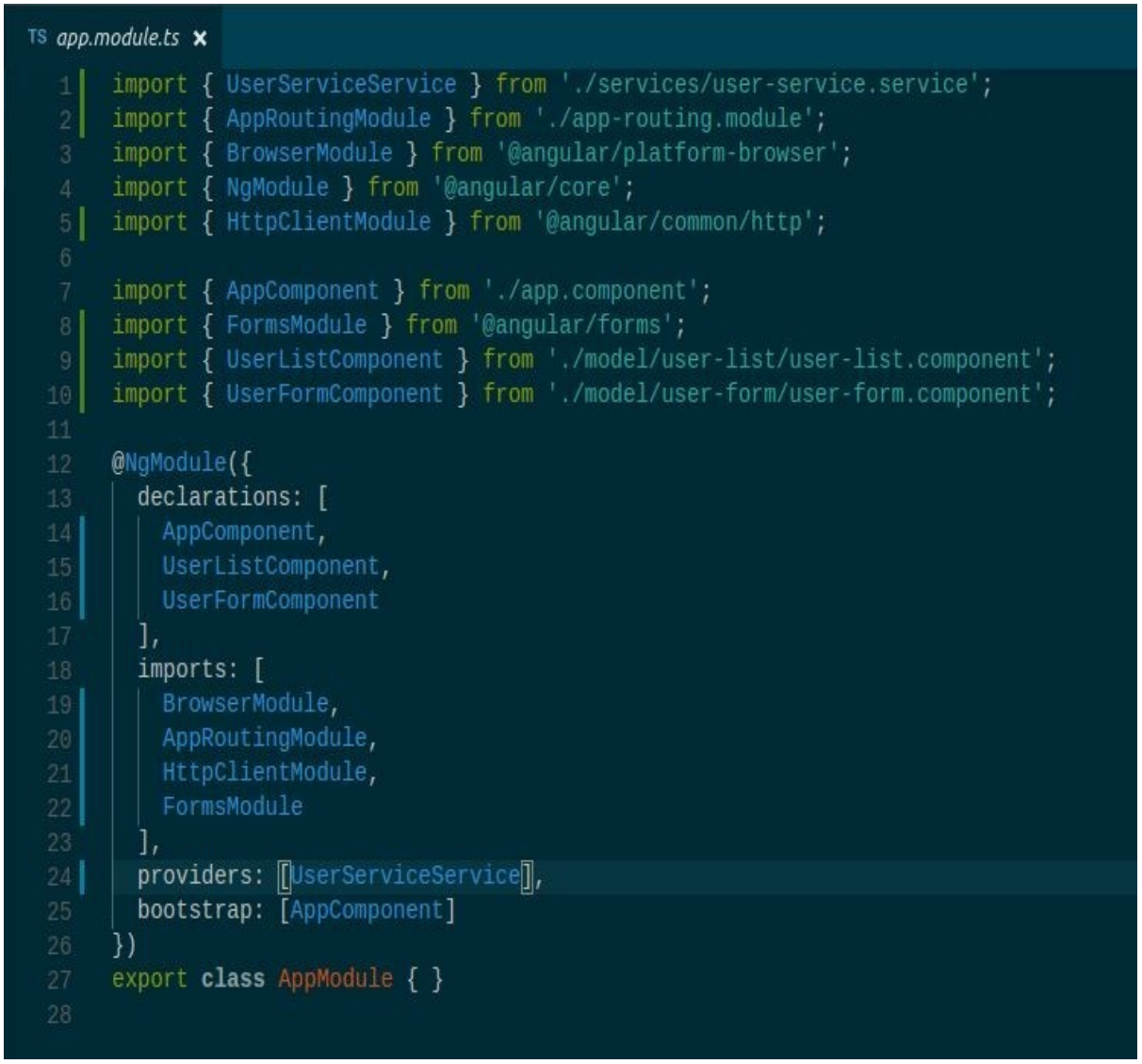
app-routing.module.ts

```typescript
import { UserFormComponent } from './model/user-form/user-form.component';
import { UserListComponent } from './model/user-list/user-list.component';
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  { path: 'users', component: UserListComponent },
  { path: 'adduser', component: UserFormComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  declarations: []
})
export class AppRoutingModule { }

```

The **routes** table contains the associations of URLs with the corresponding *components*, do not forget to import it into the **NgModule** with forRoot (routes).

App module

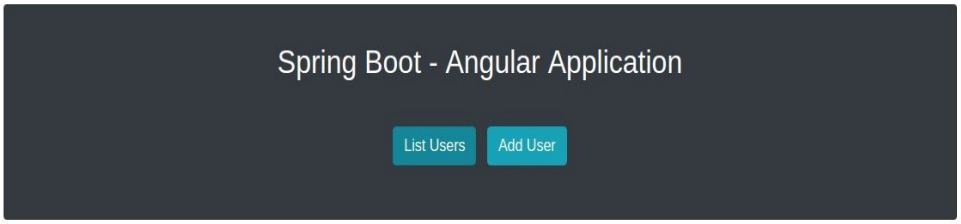
You must now modify the file **app.module.ts**, to import all the necessary modules, components and services, if this has not been done automatically.



Now you can launch the application, on the terminal launch type the following command:

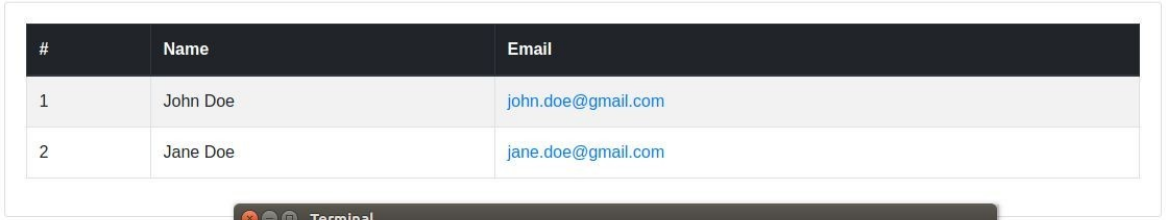
```
ng serve --open
```

You should arrive at the following home page:

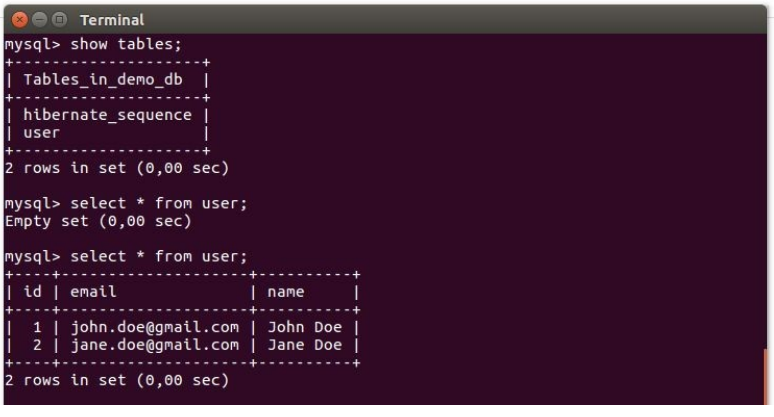


Click on the Add User button, you can add a new User

Click the Submit button, which returns you directly to the list of users:



#	Name	Email
1	John Doe	john.doe@gmail.com
2	Jane Doe	jane.doe@gmail.com



On the terminal, you can see the addition to your SQL table of the users you have just added, which confirms that the application is working properly. After restarting the app, if you click the List Users button directly, the users who are already in the table should be displayed correctly.